Scalable Graph-Based Learning Applied to Human Language Technology

Andrei Alexandrescu

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2009

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Andrei Alexandrescu

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of the Supervisory Committee:

_____

Katrin Kirchhoff

Reading Committee:

_____

Katrin Kirchhoff

_____

Oren Etzioni

_____

Jeffrey A. Bilmes

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, 1-800-521-0600, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

Scalable Graph-Based Learning Applied to Human Language Technology

Andrei Alexandrescu

Chair of the Supervisory Committee:
Associate Research Professor Katrin Kirchhoff
Electrical Engineering

Graph-based semi-supervised learning techniques have recently attracted increasing attention as a means to utilize unlabeled data in machine learning by placing data points in a similarity graph. However, applying graph-based semi-supervised learning to natural language processing tasks presents unique challenges. First, natural language features are often discrete and do not readily reveal an underlying manifold structure, which complicates the already empirical graph construction process. Second, natural language processing problems often use structured inputs and outputs that do not naturally fit the graph-based framework. Finally, scalability issues limit applicability to large data sets, which are common even in modestly-sized natural language processing applications. This research investigates novel approaches to using graph-based semi-supervised learning techniques for natural language processing, and addresses issues of distance measure learning, scalability, and structured inputs and outputs.

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I hope that the committee and the occasional reader will allow me to use a considerably less formal tone in here than through the rest of my dissertation. If you prefer formal language, math, and the passive voice, there's a good amount of those to come after this page (though I did try to not overuse the latter). This is the one place where I can be "I" and where I can extend my personal thanks to the people who helped me the most with my work.

Doctoral work is much more of a team effort than most of those involved would realize. In spite of the many long solitary hours I have spent on my research, I have been the lucky beneficiary of a vast support network.

In mentioning those I'd like to extend my gratitude to, finding the appropriate order or even a starting point is difficult. We carry the integral over time of our influencing experiences and interactions on our back, and it's difficult to assess and compare an early but decisive interaction against a relatively late but intense collaboration. I can only proceed in an order that I myself consider arbitrary.

I'd like to thank my advisor, Katrin Kirchhoff, for taking, starting back in 2005, an insecure grad student with a huge impostor syndrome from the "NLP seems to be cool" level all the way to doctoral level. Looking back, I see many ways in which I could have done better, and I'd almost wish her that I was one of her worst students if my sense of honor wouldn't spur me into wishing I was one of her best. I also thank my other committee members, Jeff Bilmes and Oren Etzioni. We didn't get to interact as much, but I've received very helpful and influential insights and feedback from both after my general exam. Thanks to Jim Augerot for graciously accepting the role of GSR.

I owe a great debt to Scott Meyers, who has been my friend and mentor for eleven years now. He has gently but unabatedly encouraged me to pursue a doctorate by using some odd inverse psychology, as witnessed by this quote from an old email of his: "Ask yourself: do I really enjoy doing research, which means putting in very long hours for next to no money on topics that are unlikely to be terribly practical? If not, stay away from a Ph.D." Our friendship won't go down in history like that of Hardy and Ramanujan, but that's just because I'm nowhere near the latter; I can only say Scott's done best he could with whatever material he got.

My wife Sanda has been incredibly loving, supportive, and motivating while we have navigated this time of our life together, which has also included her own career growth and our son's birth. It's been almost four years now since we first met, and they've been the best years of my life. Considering these were grad student years we're talking about, that's saying a world of good about her. I'm also grateful to our six-months-old Andrew. My dissertation has failed to smile back at me, but he did a great job at supplementing it in that regard. I wish that some time down the road he'll have no difficulty in understanding this dissertation, and also that he'll find my research quaint and obsolete in ways I can't even imagine.

I thank Andrei Ionescu-Zanetti, my high school physics teacher and now long-time friend, for his extraordinary positive influence during my formative years as a would-be scientist and as a person.

My friends Petru Mărginean, Mihail Antonescu, Ionut Burete, Walter Bright, and Bartosz Milewski also deserve much credit, among others too numerous to mention. Whenever I'm down, I like to take solace in thinking that I must be doing something right if I got to meet and befriend people who are so much better than myself.

The Signal, Speech and Language Interpretation (a.k.a. SSLI "sly") Lab has been a generous and friendly oasis for research and beyond. I know the atmosphere in a hierarchical environment is mostly determined by its leaders, so there is no doubt in my mind that the faculty Jeff Bilmes, Mari Ostendorf, and of course Katrin are to be credited for that. I'm grateful to everybody in the lab, and in particular I've enjoyed many fruitful conversations related to research and much more with Amittai Axelrod, Chris Bartels, Kevin Duh, Karim Filali, Sangyun Hahn, Dustin Hillard, Xiao Li, Jeremy Kahn, Jon Malkin, Arindam Mandal, Marius Alex Marin, Scott Otterson, Sarah Petersen, Sheila Reynolds, Karen Studarus, Amar Subramanya, and Mei Yang. (Special credit goes to Karim, with whom I've enjoyed many great conversations over about as many coffees. What didn't kill us certainly made us sleep less.)

My parents have inoculated me from an early age with a desire to learn and have always encouraged my creativity, which was not always easy given the rate with which I was breaking electronic appliances around the house. My sister and parents have been incredibly staunch supporters of mine, always believing I'll do something great... yet always happy with whatever I ended up doing.

I'd like to extend my gratitude to the faculty and students in the CSE department at University of Washington. This is a great place to pursue research and I found it very nurturing and collaborative. I've received good insights and advice (particularly while I was looking for a thesis topic) from graduate advisor Lindsay Michimoto and professors Richard Anderson, Brian Bershad, Gaetano Boriello, Luis Ceze, Craig Chambers (special thanks for guiding me through my anticlimactic quals), Carl Ebeling, Steve Gribble, Dan Grossman, Richard Ladner, David Notkin, Mark Oskin, Larry Snyder, Steve Tanimoto, Dan Weld, David Wetherall, and John Zahorjan.

Last but not least, many thanks for all interaction to fellow grad students Seth Bridges, Alex Colburn, Lubomira Dontcheva, Jon Froelich, Jeremy Holleman, Jonathan Ko, Keunwoo Lee, Wilmot Li, Todd Millstein, Andrew Petersen, Matthai Philipose, Maya Rodrig, Pradeep Shenoy, Aaron Shon, Adrien Treuille, Deepak Verma, Steve Wolfman, Tao Xie, and Alexander Yates.

I'm sure I've forgotten more than a few. My hope is that if you believe your name belongs here, you have also developed an understanding of and a tolerance to my proverbial forgetfulness.

Chapter 1

INTRODUCTION

Machine Learning methods based on global similarity graphs can be used successfully against realistically-sized Human Language Technology tasks addressing problems in Natural Language Processing, Automatic Speech Recognition, and Machine Translation.

There are good reasons for pursuing such an endeavor. We are applying Graph-Based Learning—a novel machine learning method sporting many desirable properties—to concrete problems in the vast, dynamic, and largely unsolved fields of Natural Language Processing, Automatic Speech Recognition, and Machine Translation. We will refer to these fields collectively as Human Language Technology, in short HLT. Although various algorithms for learning with similarity graphs have been proposed, they have been largely confined to highly problem-specific formulations and small data sets. This dissertation proposes generalized, systematic, and scalable applications of graph-based learning to a large variety of HLT tasks—and possibly beyond.

1.1 What is Human Language Technology?

Our daily lives are more structured, sophisticated, and informationally richer than probably at any time in history. We have become so used to the notions of rapid change and progress, it is hard to imagine that most previous generations of people lived through long periods of relative stagnation. Discussing whether that all is for our own good is beyond the scope of this dissertation, but one thing is clear—one fundamental cause of today's rate of progress is the advent of automated computing.

Computing has pervaded our daily lives in many ways, starting from the obvious such as personal computers and the Internet, and ending with the many small embedded systems residing in today's music players, telephones, and kitchen appliances. Clearly computers have matched and exceeded human capabilities at sheer numeric computation and information storage, and also at certain specialized tasks that were once considered the monopoly of human intelligence—such as planning, proving theorems, or playing chess. Many interesting and successful applications of automated computing, however, include the human as the essential participant in an asymmetric exchange: content on the hugely informative Internet is mostly generated by humans; popular systems such as the Web, email, instant messaging, social websites, or smart telephony, do little more than boringly brokering interaction between human beings, who do the "interesting" part. One key piece in expanding the capabilities of computers in such directions is having them understand and exchange information in natural language. This is the object of the vast field of Human Language Technology (HLT).

Improving on automated processing of human language is not only helping human-machine interfacing, but more importantly makes a wealth of human-produced information available for automated processing. Such processing would reinforce a learning cycle that further equips machines with the capability to acquire ever more detailed and subtler aspects of human culture. However,

priming this cycle poses a chicken-and-egg problem. Human language is as complex as the human psyche itself. Language is the main vehicle we use to understand the world, to conceptualize new ideas, and most often to convey them. Since to this day we have not seeded Artificial Intelligence, and since true human language understanding likely requires full-fledged human-like intelligence, Human Language Technology is one of the most formidable challenges that computing is facing today. HLT is colloquially called "AI-complete," hinting to the fact that achieving human-grade HLT is tantamount to achieving human-grade AI.

Due to its size and complexity, the field of HLT is divided in many highly specialized subfields. Within the main fields of Automatic Speech Recognition, Machine Translation, and Natural Language Processing, HLT subareas under active research today include parsing, speaker detection, document and speech summarization, speaker detection, word sense disambiguation, named entity detection, question answering, coreference resolution, part-of-speech tagging, information extraction, and more.

The initial research enthusiasm underestimated the size of the problem, but after a boom and bust cycle, the field of HLT is undergoing an accelerated evolution. Even the most skeptical observer would have to admit—sometimes with annoyance—that automated HLT systems are percolating through the fabric of our society. Speech interfaces for automated phone dialog systems not only make it more difficult to reach an actual human customer service representative, but act increasingly less distinguishable from one; combining speech recognition and automated translation has also led to early automated two-way telephone translation systems; myriads of automated systems connected to the Internet parse and process text pages, answer questions written in natural language, or produce intelligible translations of web pages and other texts (albeit sometimes with humorous results); and the list could continue. While we are far from anything like a true solution, these steps show that we do have an attack on the problem.

Several factors are conditioning this recent accelerated progress. The increased availability of computing power, the advent of the Internet, the ubiquity of broadband communication, and the exponential improvement of storage in both density and affordability [204] have enabled production of text data in enormous quantities [35], with speech data closely following suit [154]. In a concurrently-evolving trend in HLT, statistical methods outpaced symbolic/rule-based methods in applicability and performance [105, Ch. 1]. (Rules are, however, making a comeback, just not as whole systems, but as aides and complements to statistical systems [196, 225, 95].)

Such data abundance would bode well for the data-hungry statistical HLT approaches, except that many statistical HLT applications require model training with *labeled* (annotated) data. In contrast with the readily-available raw data, labeled data is labor-intensive, slow to produce, and expensive to obtain. Scarcity of labeled data is most acutely felt for less known languages, such as:

- languages without writing systems (purely spoken). Of the world's estimated 7000 languages, only one third have writing systems [74];

- languages without standardized writing systems. Scripts of such languages have a lot of variation, which requires extensive text normalization and therefore further slows down data acquisition;

- dialects and vernacular languages;

- non-mainstream languages (languages offering little economic or political incentive to HLT system builders).

Today's strong informational globalization trends warrant developing HLT systems that can work with languages and domains offering little annotated data relatively to the quantity of unannotated data. This setup is directly addressed by semi-supervised learning methods, which we briefly describe below.

Traditional statistical learning methods use a *supervised* approach, meaning that a model's parameters are adjusted (trained) by using labeled data, i.e., data for which both inputs (also known as *features*) and correct outputs (often referred to as *labels*) are known. After the model has been trained, it is able to predict correct labels when presented with formerly-unseen features, as long as there exists correlation between features and labels and the correlation is the same for both training and test data. Another statistical learning method, in a way converse to supervised learning, is *unsupervised* learning. In an unsupervised setup, labels are not known for neither training nor test data. The system, however, infers labels by discovering patterns and clusters in feature space. There is no formal distinction between training and test data. Finally, a third method called *semi-supervised* learning borrows traits from both supervised and unsupervised learning: like in supervised learning, labeled samples are present; and like in unsupervised learning, unlabeled (test) data is used during the learning process. Unlabeled data hints the learning system with information about density of data in feature space. If density of data is high around specific labels and relatively low around decision boundaries (assumption that is sometimes, but not always, applicable), then unlabeled samples may help the labeling process. Section § 2.2 includes a formal definition and an in-depth discussion of semi-supervised learning, including its enabling assumptions.

Semi-supervised learning methods include self-training [229], co-training [28], transductive Support Vector Machines [110], and graph-based methods [239, § 6]. Our work builds on the latter. Some properties of interest in Graph-Based Learning (GBL) include few parameters to tune, global consistency over train and test data, tractable global optimum, inherently adaptive modeling, solid intuition behind the learning process, and most importantly, excellent results with the setup of little train data and abundant test data (a situation common to many HLT applications, as discussed above). These advantages would make GBL an excellent match for many of today's challenging machine learning tasks in Human Language Technology, were it not for its disadvantages: the burden of choosing an appropriate similarity measure in complicated feature spaces, exacerbated scalability issues (quadratic time complexity in the total data size in a straightforward implementation), problems in addressing disparity of train and test data, the need to load the entire data set in-core prior to computation, and the difficulty to parallelize (an increasingly prominent requirement from basic algorithms in wake of today's serial computing crisis). This work first provides the appropriate background information and then addresses these difficulties from both theoretical and practical perspectives, with a focus on getting principled, theoretically sound solutions to work on realistic tasks in HLT—a heavily experimental field. Experiments conducted show how the proposed solutions properly tackle the respective challenges, and the obtained results illustrate how the improved graph-based algorithms perform significantly better than state-of-the art machine learning systems for HLT.

Organization The rest of this dissertation is structured as follows. Chapter 2 introduces semi-supervised learning as a general approach to learning and provides the necessary background for Graph-Based Learning, with an emphasis on the label propagation algorithm and its characteristics concerning HLT applicability. Chapter 3 discusses in detail the problem of graph construction. Chapter 4 discusses applications of graph-based learning to structured learning. Chapter 5 discusses scalability issues in graph-based learning, and Chapter 6 concludes by assessing the intended impact of the proposed research.

Summary of Contributions We provide in Chapter 2 an alternative proof of convergence for iterative label propagation. Compared to the original proof by Zhu [238], our proof rigorously uses only the minimal requirements for convergence, while remaining simple and terse. Chapter 3 proposes a data-driven approach to graph construction. That approach uses a supervised classifier that provides features for the graph-based learner. We illustrate data-driven graph construction with experiments on lexicon learning and word sense disambiguation. On the latter task we obtain significantly better results than the comparable state of the art (the former experiment has no baseline). In Chapter 4 we propose a framework for applying graph-based learning to structured inputs and outputs, in a formalization that is applicable to a large variety of tasks. We then instantiate that framework for machine translation and apply it to a real-world translation task, improving on a state-of-the art baseline. Finally, we introduce several contributions in Chapter 5 dedicated to scalability:

- an in-place label propagation algorithm that is always faster than the original iterative algorithm (experimentally converges in roughly one third of the number of steps);
- a multicore label propagation algorithm that uses parallel processing and benign data races to distribute work on label propagation;
- a graph reduction algorithm that reduces the size of the graph by orders of magnitude without affecting the result of label propagation (we use label propagation as proposed by Zhu [238] throughout this dissertation);
- experiments with a real-world speech corpus that yield accuracy significantly better than state-of-the art results on the Vocal Joystick speech corpus, while also being scalable by using kd-trees for fast nearest neighbors computation; and
- an algorithm called DYNTRIE that optimizes string kernel computations over a set of strings, which experimentally is three times faster than existing approaches.

Two appendices mention theoretical results that we believe are interesting and potentially useful, but that we have not used in our experiments. One appendix introduces two upper bounds for the number of steps to convergence of the label propagation algorithm, and the other defines an alternate algorithm that converges in fewer steps than the version we use, at the cost of requiring a more expensive matrix squaring operation.

Chapter 2

BACKGROUND

This chapter introduces the reader to the fundamentals of semi-supervised learning, in particular graph-based learning and label propagation, with a focus on Human Language Technology applicability.

2.1 Notational Aid

For convenience in understanding the equations presented in this work, Table 2.1 defines the most important notations used throughout. By necessity some of the terms have not been defined at this point yet, so the reader may want to skip this section for the moment and return to it whenever the definition of a symbol is unclear from context.

Table 2.1: Main notations used throughout this document.

Notation	Description
a, b, c	Real numbers or sequences
$\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$	Row vectors (e.g., feature vectors)
A, B	Matrices or sets
$[a, b)$ etc.	Classic interval notation, "(" and ")" for open, "[" and "]" for closed
$\mathbb{R}_+$	The interval $[0, \infty)$
$\mathbb{R}^*$	$\mathbb{R} \setminus \{0\}$ (also $\mathbb{R}^*_+$ is $(0, \infty)$ and $\mathbb{N}^*$ is $\mathbb{N} \setminus \{0\}$)
$\{e_1, \ldots, e_n\}$	Finite set
$\langle\langle e_1, \ldots, e_n \rangle\rangle$	Finite ordered set a.k.a. row vector (unlike in a set, the order does matter and equal elements may occur multiple times)
$\mathbf{a}_{[i]}$	The i^{th} component of vector $\mathbf{a}$ (notation chosen to avoid confusion with $\mathbf{a}_i$, the i^{th} vector in an ordered set $\langle\langle \mathbf{a}_1, \ldots, \mathbf{a}_n \rangle\rangle$)
B^A	For sets A and B, B^A is the set of functions defined on A with values in B: $B^A \triangleq \{f \mid f : A \to B\}$
A^n	The set of row vectors of length $n \in \mathbb{N}^*$ with elements in A (A stands in for any set, e.g. $[0, 1]^n$ is the set of row vectors of length n with elements in $[0, 1]$)
$A^{m \times n}$	The set of $m \times n$ matrices with elements in A
$\mathbb{1}^n$	The identity matrix of size $n \times n$ (n may be missing if clear from the context)
$c^{m \times n}$	A matrix of size $m \times n$ with all elements equal to c

(continued)

Table 2.1 *(continued)*

Notation	Description
$\delta_n(b)$	The Kronecker vector of length $n \in \mathbb{N}^*$ with 1 in position $b \in \{1, \ldots, n\}$ and 0 everywhere else: $\langle\!\langle \overbrace{0, 0, \ldots, 0}^{b-1}, 1, \overbrace{0, 0, \ldots, 0}^{n-b} \rangle\!\rangle$
$\log x$	Logarithm in base 2
$\ln x$	Natural-base logarithm
$\ell \in \mathbb{N}^*$	Label count (number of distinct labels in an unstructured classification problem)
$\mathtt{t} \in \mathbb{N}$	Number of labeled (training) data samples
$\mathtt{u} \in \mathbb{N}$	Number of unlabeled data samples
$\mathtt{X} = \langle\!\langle \mathtt{x}_1, \ldots, \mathtt{x}_n \rangle\!\rangle$	Train and test features
$\mathcal{X}$	The (possibly infinite) set to which train and test features belong in a learning problem
$\mathtt{Y} = \langle\!\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\!\rangle$	The training labels
$\mathcal{Y}$	The set that labels belong to (for unstructured labels $\mathcal{Y} = \{1, \ldots, \ell\}$, for structured labels $\mathcal{Y}$ is an elaborate, potentially infinite set that depends on the problem)
$\mathrm{card}(X)$	The number of elements in discrete set X (for infinite sets, $\mathrm{card}(X) = \infty$)
$\mathtt{W} \in \mathbb{R}_+^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$	Symmetric matrix holding pairwise similarities between samples, with labeled samples coming in the top-left corner
$\mathtt{P} \in [0,1]^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$	Matrix holding *row-normalized* pairwise similarities between samples, in the same order as $\mathtt{W}$
$\mathtt{P}_{\mathtt{UL}} \in [0,1]^{\mathtt{u}\times\mathtt{t}}$	Bottom-left sub-matrix of $\mathtt{P}$ holding unlabeled-labeled similarities
$\mathtt{P}_{\mathtt{UU}} \in [0,1]^{\mathtt{u}\times\mathtt{u}}$	Bottom-right sub-matrix of $\mathtt{P}$ holding unlabeled-unlabeled similarities
$\mathtt{f} \in \mathbb{R}^{(\mathtt{t}+\mathtt{u})\times\ell}$	Matrix holding the (temporary) solution in a label propagation iteration
$\mathtt{f}_\mathtt{L} \in \mathbb{R}^{\mathtt{t}\times\ell}$	The top $\mathtt{t}$ lines of $\mathtt{f}$
$\mathtt{f}_\mathtt{U} \in \mathbb{R}^{\mathtt{u}\times\ell}$	The bottom $\mathtt{u}$ lines of $\mathtt{f}$
$\gamma \in \mathbb{R}_+^*$	Lower bound for convergence speed in iterative label propagation
$\tau \in \mathbb{R}_+^*$	Tolerance for fixed point convergence
$\leftarrow$	Mutation, e.g. $\mathtt{f} \leftarrow \mathtt{Pf}$ replaces $\mathtt{f}$ with $\mathtt{Pf}$ (only valid in algorithms)
$\triangleq$	Introduction of notation, e.g. $\|\mathbf{a}\| \triangleq \sqrt{d(\mathbf{a}, 0)}$
$[\![a]\!]$	The indicator function: 1 if Boolean predicate a is true, 0 otherwise
$\overrightarrow{f}\,(A)$	Given $A \in \mathbb{K}^a$ and $f : \mathbb{K} \to \mathbb{K}'$, $\overrightarrow{f}\,(A)$ creates a vector $A' \in \mathbb{K}'^a$ containing the element-wise application of f to A

The expression $p \log p$ occurs frequently in this text with $p \geq 0$ (usually p is a probability). Although the function $p \log p$ is undefined for $p = 0$, we define by convention $0 \log 0 = 0$. This is a continuous extension justified by the fact that $\lim_{p \searrow 0} p \log p = 0$.

2.2 Semi-Supervised Learning

Machine learning techniques for supervised classification use labeled data to train models that learn an input-output mapping function. A supervised model takes as its training input a sample collection represented by feature vectors $\mathtt{X} = \langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{t} \rangle\rangle$, where $\mathtt{x}_i$ are vectors belonging to a feature space $\mathcal{X}$. Also, in a typical unstructured classification task, discrete labels are available for these samples: $\mathtt{Y} = \langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\rangle$ with $\mathtt{y}_i \in \{1, \ldots, \ell\}\ \forall i \in \{1, \ldots, \mathtt{t}\}$. The goal of the training stage is to obtain a system that provides a good approximation of the probability $p(\mathtt{y}|\mathtt{x})$. When presented with previously-unseen (test) samples in $\mathcal{X}$, the system is able to attribute estimated labels to them. The commonly-made enabling assumption is that both train and test samples belong to the same distribution—i.e., they are assumed to be independently and identically distributed.

One problem is that sometimes—and frequently in Human Language Technology (HLT) applications—obtaining labeled data is a slow, expensive, and error-prone process that requires expert human annotators to tag data manually. In contrast, unlabeled data (such as raw text, speech, or images) is often abundant and easily obtainable. The need is therefore to approximate $p(\mathtt{y}|\mathtt{x})$ from only few labeled samples and many unlabeled samples. Semi-supervised learning (SSL) is designed to exploit such situations by systematically using a small amount of labeled data in conjunction with a relatively large amount of unlabeled data in the learning process.

The typical SSL model takes as input a sample set represented by features $\mathtt{X} = \langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\rangle$, where $\mathtt{x}_i$ are again vectors in $\mathcal{X}$. Discrete labels are available for the first $\mathtt{t}$ samples: $\mathtt{Y} = \langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\rangle$ with $\mathtt{y}_i \in \{1, \ldots, \ell\}\ \forall i \in \{1, \ldots, \mathtt{t}\}$. The goal is to obtain a classifier that minimizes classification errors on the test set. Depending on subsequent use, two kinds of SSL classifiers can be distinguished:

- *transductive:* the test data is $\langle\langle \mathtt{x}_{\mathtt{t}+1}, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\rangle$;
- *inductive:* the test data consists of samples $\langle\langle \mathtt{x}_{\mathtt{t}+\mathtt{u}+1}, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}+m} \rangle\rangle \in \mathcal{X}^m$, unseen during training, to which the original unlabeled set $\langle\langle \mathtt{x}_{\mathtt{t}+1}, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\rangle$ may or may not be added.

Distinguishing between SSL and transduction can be subtle, partly because the literature tends to use slightly different definitions for each. A simple definition proposed by Zhu [239] is: SSL is transductive if the resulting model is defined only on $\mathtt{X}$; in contrast, if the model is defined on $\mathcal{X}$ (i.e., it can predict a label for any point in the feature space), then SSL is transductive. For example, although transductive support vector machines (TSVMs) [111] assume a transductive setup, they define a model that, in spite of its name, supports inductive inputs naturally. On the other hand, traditional graph-based approaches [27, 234] are unable to handle unseen inputs, although recent work has extended graph-based frameworks to handle unseen inputs without modifying the model and in a computationally-efficient manner [64]. In short, in transductive learning all test data is available at the beginning of the training process, whereas in inductive learning the training proceeds without some of the test data (or even without any test data at all, in which case the semi-supervised effect is forgone and the process degenerates to simple supervised learning).

Like any machine learning technique, SSL builds on certain assumptions about the nature of the function to learn. All machine learning methods rely on some notion of continuity or *smoothness* of the function mapping inputs (features) to outputs (labels): if $\mathtt{x}$ and $\mathtt{x}'$ are similar, then the labels $\mathtt{y}$ and $\mathtt{y}'$ are likely to be similar (equal in the case of discrete labels). Semi-supervised learning

methods actively exploit unlabeled data in enforcing that assumption. Use of unlabeled samples can only help if $p(\mathtt{x})$ can be related to $p(\mathtt{y}|\mathtt{x})$, and to do that additional density assumptions are needed. Commonly-used assumptions used by SSL algorithms are [42, Ch. 1]:

- *The cluster assumption:* Data points in the same cluster have the same label. A converse formulation is that the decision boundary should span low-density spaces and avoid high-density spaces. Adding more unlabeled data helps defining clusters and identifying high-density and low-density regions.

- *The manifold assumption:* The high-dimensional samples lie on a low-dimensional manifold. This can be seen as a particular case of the cluster assumption. Adding unlabeled samples helps approximating the structure of the manifold and computing accurate geodesic distances.

If semi-supervised assumptions are not met, it is possible that unlabeled data actually harms the learning process [58, 56, 57]. As a simple example of unmet assumptions, consider two clusters in $\mathcal{X}$ belonging to distinct classes (i.e., bearing distinct labels). Some samples in each cluster are labeled, and many are not. If the clusters do not overlap significantly, the decision boundary goes through a low-density region. But as the clusters get closer to each other, the data density in the overlapping region grows and at a point will even surpass the maximum density of either or both clusters. In that case, a density-informed semi-supervised learner may conclude that the clusters belong to the same class. In such cases the class with a higher density of training samples "wins" and in fact the use of unlabeled data only hurts because it propagates the wrong decision deeper into the other cluster's region.

2.3 Graph-Based SSL

Graph-based SSL algorithms have received increasing attention in the recent years [27, 214, 235, 236, 237, 232, 29, 239]. In graph-based SSL, data points are arranged in a weighted undirected graph that reflects similarity among samples; the weight of an edge encodes the strength of the similarity between that edge's endpoints. Unweighted similarity graphs can be considered to have unit weights for all edges. The graph is characterized by its symmetric weight matrix $\mathtt{W} \in \mathbb{R}_{+}^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$, whose elements $\mathtt{w}_{ij} = \mathtt{w}_{ji}$ are similarity measures between vertices i and j, and by the ordered set $\langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\rangle$ that defines labels for the first $\mathtt{t}$ vertices. If no edge is linking nodes i and j, then $\mathtt{w}_{ij} \triangleq 0$. Other than that, applications have considerable freedom in choosing the edge set and the $\mathtt{w}_{ij}$ weights. For example, a simple approach to building a graph is to define $\mathtt{w}_{ij} = 1$ if $\mathtt{x}_i$ and $\mathtt{x}_j$ fall within each other's k nearest-neighbors, and zero otherwise. Another commonly-used weight matrix is defined by a Gaussian kernel of parameterized width:

$$\mathtt{w}_{ij} = \exp\left[-\frac{d(\mathtt{x}_i, \mathtt{x}_j)^2}{\alpha^2}\right] \tag{2.1}$$

where $d(\mathtt{x}_i, \mathtt{x}_j)$ is the (estimated) distance between feature vectors $\mathtt{x}_i$ and $\mathtt{x}_j$, and α is a hyperparameter to be chosen on a theoretical basis or optimized experimentally. Notice how similarity is quickly decaying with distance, reflecting a dependence of graph-based SSL on accurate estimates of high similarity, but not necessarily of low similarity. In practice, a host of distance measures

have been used, based e.g. on cosine similarity, Euclidean distance, Jeffries-Matusita distance, or Jensen-Shannon divergence. Often, applications use a blend of methods in defining $\mathtt{W}$, for example by layering a k nearest-neighbors or fixed-radius neighborhood on top of weights calculated by using Eq. 2.1. Choosing the appropriate similarity measure practically decides the graph construction and is the most important factor in successfully applying graph-based SSL.

Intuitively, in graph-based SSL, outputs can be computed by means of local graph neighborhood membership, even though the similarity of many unlabeled samples with actual labeled samples can be weak or even not defined. This is why graph-based semi-supervised learning often performs better than nearest-neighbor approaches, although both make similar assumptions.

2.3.1 Graph-Based Learning Algorithms

Blum and Chawla [27] formulated binary classification on a similarity graph as a mincut problem, i.e. finding the smallest total weight of edges that, when removed, cut the flow between the binary-labeled samples, modeled as sources and sinks. The nodes then are classified depending on whether they are on the source or sink side of the partitioned graph. The main problem with this approach is that it gives discrete results (does not provide a confidence of the labeling), which makes the method unsuitable for function regression. In follow-up work, Blum et al. [29] obtain confidence information in a manner reminiscent of the Monte Carlo method by performing multiple mincut calculations, each preceded by adding random noise to edge weights. Averaging over many mincuts lends confidence information to the classification.

Szummer and Jaakkola [214] proposed a random walk on a similarity graph. Their random walk has a maximum length that ensures termination, albeit not necessarily at a global optimum. Zhou et al. [232] described the label spreading algorithm, which is similar to label propagation but includes a regularization term in the cost function, thus yielding a smoother output. Belkin et al. focused on the regularization aspect and derived bounds on the generalization error [12], and also developed theoretical underpinnings for handling out-of-sample labels [13]. Agarwal [3] proposed an algorithmic framework for hierarchical ranking on graph data by means of regularization using a modified cost function. Zhu et al. defined SSL using Gaussian fields and harmonic functions [237] and defined the label propagation algorithm [234], proving that it always converges to a global optimum. Our research builds on the label propagation algorithm, which we describe in detail below.

2.3.2 Label Propagation

Once the $\mathtt{W}$ matrix is constructed, the basic label propagation algorithm [234] also constructs the matrix $\mathtt{Y_L}$ of size $\mathtt{t} \times \ell$, encoding the known labels as Kronecker vectors:

$$\mathtt{Y_L}_{(\text{row } i)} = \delta_\ell(\mathtt{y}_i) \tag{2.2}$$

where $\delta_\ell(\mathtt{y}_k)$ is a Kronecker row vector of length ℓ containing 1 in position $\mathtt{y}_k$:

$$\delta_\ell(\mathtt{y}_k) \triangleq \langle\!\langle \overbrace{0, 0, \ldots, 0}^{k-1}, 1, \overbrace{0, 0, \ldots, 0}^{\ell-k} \rangle\!\rangle \tag{2.3}$$

Algorithm 1 defines iterative label propagation. The definition usually found in literature [238] does not include the tolerance τ and only focuses on iteration to convergence without regard for speed of

convergence. We introduced $\tau > 0$ to ensure provable convergence in a finite number of steps, for which we will compute a bound in Chapter 5. Also, our definition provides more detail for practical implementations.

Algorithm 1: Iterative Label Propagation

Input: Labels $\mathtt{Y}$; similarity matrix $\mathtt{W} \in \mathbb{R}_+^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$ with $\mathtt{w}_{ij} = \mathtt{w}_{ji} \geq 0$ $\forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; tolerance $\tau > 0$.

Output: Matrix $\mathtt{f}_\mathtt{U} \in [0,1]^{\mathtt{u}\times\ell}$ containing unnormalized probability distributions over labels.

1 $\mathtt{w}_{ii} \leftarrow 0 \;\; \forall i \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$;

$\mathtt{p}_{ij} \leftarrow \dfrac{\mathtt{w}_{ij}}{\sum_{k=1}^{\mathtt{t}+\mathtt{u}} \mathtt{w}_{ik}} \;\; \forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$;

2

3 $(\mathtt{Y}_\mathtt{L})_{\text{row } i} \leftarrow \delta_\ell(\mathtt{y}_i) \;\; \forall i \in \{1, \ldots, \mathtt{t}\}$;

4 $\mathtt{f}'_\mathtt{U} \leftarrow 0^{\mathtt{u}\times\ell}$;

5 **repeat**

6 $\quad \mathtt{f}_\mathtt{L} \leftarrow \mathtt{Y}_\mathtt{L}$;

7 $\quad \mathtt{f}_\mathtt{U} \leftarrow \mathtt{f}'_\mathtt{U}$;

8 $\quad \mathtt{f}' \leftarrow \mathtt{P}\mathtt{f}$;

until $\sum_{i=\mathtt{t}+1}^{\mathtt{t}+\mathtt{u}} \sum_{j=1}^{\ell} \left|\mathtt{f}_{ij} - \mathtt{f}'_{ij}\right| < \tau$;

9

Step 1 eliminates self-similarities $\mathtt{w}_{ii}$, which are usually large relative to other similarities. This eliminates self-edges in the corresponding graph. The step is not required, but self-edges only delay convergence and may reduce numeric precision by forcing all other similarities to be small numbers after normalization.

After the algorithm terminates, the $\mathtt{f}$ matrix contains the solution in rows $\mathtt{t}+1$ to $\mathtt{t}+\mathtt{u}$ in the form of unnormalized label probability distributions; most applications need hard labels, obtainable by:

$$\hat{\mathtt{y}}_i = \arg\max_{j \in \{1,\ldots,\ell\}} \mathtt{f}_{ij} \;\; \forall i \in \{\mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u}\} \tag{2.4}$$

Zhu has shown [238] that the iteration converges. We provide the proof below for reference. Let us first split $\mathtt{P}$ into four sub-matrices:

$$\mathtt{P} = \begin{bmatrix} \mathtt{P}_{\mathtt{LL}} & \mathtt{P}_{\mathtt{LU}} \\ \mathtt{P}_{\mathtt{UL}} & \mathtt{P}_{\mathtt{UU}} \end{bmatrix} \tag{2.5}$$

With these notations, the following theorem applies.

Theorem 2.3.1 (Zhu 2003 [237]). *If* $\sum_{j=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ij} \leq \gamma < 1 \; \forall i \in \{1, \ldots, \mathtt{u}\}$, *then Algorithm 1 terminates regardless of the initial value of* $\mathtt{f}'_\mathtt{U}$ *chosen in step 4.*

Proof. The algorithm's core iteration becomes:

$$\mathtt{f}_{\mathtt{U}} \leftarrow \mathtt{f}'_{\mathtt{U}} \tag{2.6}$$

$$\mathtt{f}'_{\mathtt{U}} \leftarrow \mathtt{P}_{\mathtt{UU}}\mathtt{f}_{\mathtt{U}} + \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{2.7}$$

which is repeated until $\mathtt{f}_{\mathtt{U}}$ is equal element for element with $\mathtt{f}'_{\mathtt{U}}$, within accumulated tolerance τ. Unrolling the iteration yields

$$\mathtt{f}_{\mathtt{U}}^{\text{step } t} = \mathtt{P}_{\mathtt{UU}}^{t}\mathtt{f}_{\mathtt{U}}^{\text{step } 0} + \left(\sum_{i=1}^{t} \mathtt{P}_{\mathtt{UU}}^{i-1}\right)\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{2.8}$$

We multiply both sides of the equation by $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$ to the left, obtaining:

$$(\mathbb{1} - \mathtt{P}_{\mathtt{UU}})\mathtt{f}_{\mathtt{U}}^{\text{step } t} = (\mathbb{1} - \mathtt{P}_{\mathtt{UU}})\mathtt{P}_{\mathtt{UU}}^{t}\mathtt{f}_{\mathtt{U}}^{\text{step } 0} + (\mathbb{1} - \mathtt{P}_{\mathtt{UU}})\left(\sum_{i=1}^{t} \mathtt{P}_{\mathtt{UU}}^{i}\right)\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{2.9}$$

$$= (\mathbb{1} - \mathtt{P}_{\mathtt{UU}})\mathtt{P}_{\mathtt{UU}}^{t}\mathtt{f}_{\mathtt{U}}^{\text{step } 0} + (\mathbb{1} - \mathtt{P}_{\mathtt{UU}}^{t+1})\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{2.10}$$

We need to show that $\mathtt{P}_{\mathtt{UU}}^{t}$ converges when $t \to \infty$. In fact it does converge to the null matrix. We will show by induction that $\sum_{j=1}^{\mathtt{u}}(\mathtt{P}_{\mathtt{UU}}^{t})_{ij} \leq \gamma^{t}\ \forall t \in \mathbb{N}^{*}$. The base step for $t = 1$ is directly provided by the hypothesis. For the inductive step, we write an element of $\mathtt{P}_{\mathtt{UU}}^{t}$ as follows:

$$\sum_{j=1}^{\mathtt{u}}(\mathtt{P}_{\mathtt{UU}}^{t})_{ij} = \sum_{j=1}^{\mathtt{u}}\sum_{k=1}^{\mathtt{u}}\left(\mathtt{P}_{\mathtt{UU}}^{t-1}\right)_{ik}(\mathtt{P}_{\mathtt{UU}})_{kj} \tag{2.11}$$

$$= \sum_{k=1}^{\mathtt{u}}\left[\left(\mathtt{P}_{\mathtt{UU}}^{t-1}\right)_{ik}\sum_{j=1}^{\mathtt{u}}(\mathtt{P}_{\mathtt{UU}})_{kj}\right] \tag{2.12}$$

$$\leq \gamma\sum_{k=1}^{\mathtt{u}}\left(\mathtt{P}_{\mathtt{UU}}^{t-1}\right)_{ik} \tag{2.13}$$

$$\leq \gamma^{t} \tag{2.14}$$

The row-wise sums of elements in $\mathtt{P}_{\mathtt{UU}}$ converge to zero, and since all elements are positive, they all converge to zero. This nullifies the term involving $\mathtt{f}_{\mathtt{U}}^{\text{step } 0}$ and makes $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}^{t+1}$ converge to $\mathbb{1}$. □

Solving Eq. 2.10 for $\mathtt{f}_{\mathtt{U}}$ yields:

$$\mathtt{f}_{\mathtt{U}} = (\mathbb{1} - \mathtt{P}_{\mathtt{UU}})^{-1}\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{2.15}$$

which has a unique solution if $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$ is invertible, i.e., if all of graph's connected components have at least one labeled point in them. Notice that Theorem 2.3.1 imposes a stronger restriction, namely that *every* unlabeled node in the graph is connected to at least one labeled node. The theorem below lifts that restriction and clarifies that the non-singularity requirement alone guarantees convergence of the iterative solution.

Theorem 2.3.2. *If $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$ is non-singular, then Algorithm 1 terminates regardless of the initial value of $\mathtt{f}'_{\mathtt{U}}$ chosen in step 4.*

Proof. We first prove that each element of $\mathtt{P}^t_{\mathtt{UU}}$ decreases monotonically:

$$(\mathtt{P}^{t+1}_{\mathtt{UU}})_{ij} = \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} (\mathtt{P}^t_{\mathtt{UU}})_{kj} \tag{2.16}$$

$$\leq \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} (\mathtt{P}^{t-1}_{\mathtt{UU}})_{kj} \tag{2.17}$$

$$= (\mathtt{P}^t_{\mathtt{UU}})_{ij} \tag{2.18}$$

For the inequality we used $(\mathtt{P}_{\mathtt{UU}})_{kj} \leq 1$ and the fact that the exponential function is decreasing for bases smaller than or equal to 1. Since they are all positive, they all converge by the monotone convergence theorem [11], so there exists a matrix $\mathtt{P}^\infty_{\mathtt{UU}} \triangleq \lim_{t\to\infty} \mathtt{P}^t_{\mathtt{UU}}$. Then

$$\mathtt{P}^\infty_{\mathtt{UU}} = \mathtt{P}_{\mathtt{UU}} \mathtt{P}^\infty_{\mathtt{UU}} \tag{2.19}$$

$$\mathtt{P}^\infty_{\mathtt{UU}} - \mathtt{P}_{\mathtt{UU}} \mathtt{P}^\infty_{\mathtt{UU}} = 0 \tag{2.20}$$

$$(\mathbb{1} - \mathtt{P}_{\mathtt{UU}}) \mathtt{P}^\infty_{\mathtt{UU}} = 0 \tag{2.21}$$

By the hypothesis $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$ is invertible, so we can multiply Eq. 2.21 to the left by $(\mathbb{1} - \mathtt{P}_{\mathtt{UU}})^{-1}$ obtaining $\mathtt{P}^\infty_{\mathtt{UU}} = 0$. □

This theorem is related to perennial work on irreducible diagonal dominant matrices [208, 90, 216], and can in fact be interpreted as a converse of the Lévy-Desplanques theorem [90]. That theorem states that an irreducibly diagonally dominant matrix is nonsingular. Theorem 2.3.2 proves that a weakly diagonally dominant matrix (in this case $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$) with rows normalized to sum to 1, which is also invertible, is irreducibly diagonally dominant.

The hypothesis of Theorem 2.3.2 relaxes the restrictions imposed by the hypothesis of Theorem 2.3.1. Connectivity to at least one labeled node is not needed anymore; the requirement is that $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$ is invertible, which translates to a graph in which each connected component has at least one labeled node in it. This result is intuitively justified and also known from the methods of relaxations [68]. The class of partially-labeled graphs for which $\lim_{t\to\infty} \mathtt{P}^t_{\mathtt{UU}} = 0$ is larger than the class of graphs with non-singular $\mathbb{1} - \mathtt{P}_{\mathtt{UU}}$, but only the latter is of interest to us. Graphs including disconnected unlabeled components are not "grounded" and may receive any constant label across each such component because absence of labeled nodes brings no information to those components.

2.3.3 Illustration

Figure 2.1 shows a graph before and after the label propagation process. All edges have unit weight. Initially, there are two "+" labeled nodes and two "-" labeled nodes. To show information on confidence, we use nuance-coding as well. The label propagation algorithm pushes the labels into the test nodes, the result being a blend of "+" and "-" for each node.

Note that the graph as drawn is planar but actually could reside (as a slightly curved or "crumpled" manifold) in a high-dimensional space. Graph-based algorithms can detect and exploit the

lower-dimensional mesh defined by the graph. That is why defining a good distance measure is important—good edges reveal that data lies on a low-dimensional manifold (in this case two-dimensional) that in turn is situated in a high-dimensional feature space.

The shades of grey filling the nodes in the bottom graph in Fig. 2.1 are accurate proportional mixes of black and white computed from a real label propagation on the graph. Even after accounting for possible aberrations in the rendering process, it can be easily seen how test nodes closer to the white train nodes receive lighter shades than those closer to the black train nodes.

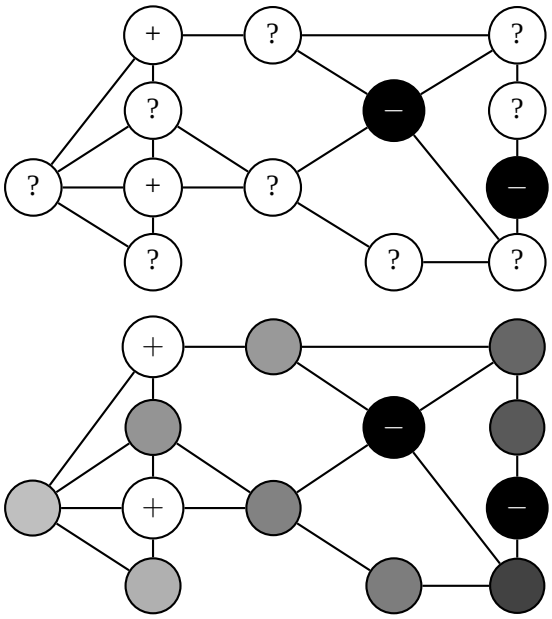


Figure 2.1: A graph for a binary classification problem before and after label propagation. The labels are encoded as white (+) and black (−), and all edges have unit weight. The process assigns labels to unlabeled nodes depending on their connections with neighboring nodes—shades of grey represent different probabilities for the label assignment. By virtue of the global connectivity properties, unlabeled nodes receive labels even when they are not directly connected to any labeled nodes.

Applications on real data lead of course to much larger graphs lying in higher-dimensional spaces. Fundamentally the desired effect in applying graph-based learning is the same: starting from points in a high-dimensional space, create a mesh defining a lower-dimensional manifold and operate on it instead of the original space.

2.3.4 Cost Function for Label Propagation

Convergence is interesting only if the convergence point has desirable properties, such as optimizing a goal useful in a learning process. The fixed point of the label propagation satisfies $\mathtt{f} = \mathtt{Pf}$, with values of $\mathtt{f}$ restricted to existing labels for all labeled data. For a given unlabeled point k in the

graph and a label c, we have

$$\mathtt{f}_{ic} = \sum_{j=1}^{\mathtt{t+u}} \mathtt{p}_{ij}\mathtt{f}_{jc} = \frac{\sum_{j=1}^{\mathtt{t+u}} \mathtt{w}_{ij}\mathtt{f}_{jc}}{\sum_{j=1}^{\mathtt{t+u}} \mathtt{w}_{ij}} \tag{2.22}$$

So along each column c, the value of $\mathtt{f}$ at each point is the weighted average of its values at neighboring nodes, with the restriction that values of $\mathtt{f}_{ic}$ at all labeled points is 1 if point i bears label c, and 0 if point i bears a different label. Functions satisfying Eq. 2.22 are called *harmonic functions*, and label propagation is in fact an application of the method of relaxations used to compute harmonic functions [68], with two notable differences: (a) label propagation uses matrix algebra to update function values at all points in the graph in one macro step; and (b) label propagation updates simultaneously function values for all ℓ labels, whereas traditionally the relaxation method computes a uni-dimensional function (akin to $\ell = 1$). Harmonic functions occur naturally in many physical and statistical phenomena (such as electric networks, thermal gradients, rigid solid physics, and random walks) [68] and enjoy a number of interesting properties. One of particular interest is *smoothness*. By Thomson's principle [1, Ch. 10], the harmonic function obtained through label propagation minimizes the following cost function:

$$\mathcal{S} = \sum_{\substack{i,j\in\{1,\ldots,\mathtt{t+u}\}\\ i>\mathtt{t}\,\vee\, j>\mathtt{t}\\ k\in\{1,\ldots,\ell\}}} \mathtt{w}_{ij}\left(\mathtt{f}_{ik} - \mathtt{f}_{jk}\right)^2 \tag{2.23}$$

The condition $i > \mathtt{t} \vee j > \mathtt{t}$ is present to clarify that values in $\mathtt{f}_{\mathtt{L}}$ are fixed and only values of $\mathtt{f}_{\mathtt{U}}$ are learned to minimize the cost function[1] $\mathcal{S}$. The cost measures the extent to which nearby nodes (as defined by $\mathtt{W}$) sport different values of $\mathtt{f}$; minimizing $\mathcal{S}$ favors globally-consistent values of $\mathtt{f}$ such that highly similar nodes are assigned highly similar values of $\mathtt{f}$. $\mathcal{S}$ is called smoothness (which is a mild misnomer, since $\mathcal{S}$ increases with "jerkiness," the opposite of smoothness).

If hard labels are needed, we must associate a label $\mathtt{y}_i \in \{1, \ldots, \ell\}$ with each unlabeled node i. The choice

$$\hat{\mathtt{y}}_i = \arg\max_{j\in\{1,\ldots,\ell\}} \mathtt{f}_{ij} \tag{2.24}$$

minimizes the discretized version of the smoothness function:

$$\mathcal{S}' = \sum_{\substack{i,j\in\{1,\ldots,\mathtt{t+u}\}\\ i>\mathtt{t}\,\vee\, j>\mathtt{t}\\ k\in\{1,\ldots,\ell\}}} \mathtt{w}_{ij}[\![\hat{\mathtt{y}}_{jk} \neq \hat{\mathtt{y}}_{ik}]\!] \tag{2.25}$$

where $[\![a \neq b]\!]$ (defined in Table 2.1) is 1 if $a \neq b$ and 0 otherwise. Given $\mathtt{f}$, the labeling choice in Eq. 2.24 minimizes $\mathcal{S}'$ because it zeroes the largest term in the partial sum (for node i)

$$\mathcal{S}_i = \sum_{j=1}^{\mathtt{t+u}} \sum_{k=1}^{\ell} \mathtt{w}_{ij}\left(\mathtt{f}_{ik} - \mathtt{f}_{jk}\right)^2 \tag{2.26}$$

[1] In fact "cost functional" would be a more precise term because in this case the cost is parameterized by a function, i.e. $\mathcal{S}(\mathtt{f})$. We use, however, an implicitly parameterized notation and the better-known phrase "cost function."

So a hard labeling obtained through label propagation finds a labeling that, to the extent possible and within the constraints established by the already-labeled nodes, assigns identical labels to nodes linked by high weights. This goal is consistent with the notion of similarity embodied by $\mathtt{W}$.

The fixed point of the label propagation algorithm has a number of equivalent interpretations leading to various methods of computing the harmonic function. An intuitive interpretation is that of a random walk. The random walk on the graph characterized by $\mathtt{W}$ and $\mathtt{Y}_\mathtt{L}$ is defined as starting with an unlabeled vertex, stopping as soon as a labeled vertex is reached, and making a step from vertex i to vertex j with probability:

$$\mathtt{p}_{ij} = \frac{\mathtt{w}_{ij}}{\sum_k \mathtt{w}_{ik}} \tag{2.27}$$

It has been shown [238] that upon convergence of the label propagation algorithm, the cell $\mathtt{f}_{ij}$ contains (after normalization) the probability that a random walk starting in unlabeled node i will terminate in a node carrying label j.

2.3.5 Previous HLT Applications

In HLT, Zhu applied label propagation successfully to a document classification task concerning learning the Usenet newsgroup to which a specific document belongs [238]. Pang and Lee [179] used min-cut to distinguish among objective and subjective documents. Zheng-Yu Niu et al. [78] experimented with applying label propagation to word disambiguation, using two different distance measures; they report significant improvements when replacing cosine distance with Jensen-Shannon divergence. Goldberg and Zhu [93] apply label propagation to a sentiment categorization task. Their graph construction includes connecting each unlabeled node to its k labeled neighbors and k' unlabeled neighbors. This allows control of the supervised vs. the unsupervised aspect of learning (for $k' = 0$, the algorithm becomes a supervised k-nearest neighbors algorithm). Zhou et al. [232] apply label spreading to the 20-newsgroups document classification task, with encouraging results.

2.3.6 Advantages and Disadvantages

When applicable, graph-based SSL has obvious advantages over traditional supervised approaches: the distribution $p(\mathtt{x})$ of unlabeled data provides valuable information for computing an accurate estimate of $p(\mathtt{y}|\mathtt{x})$, which translates into good predictions on the unlabeled data made with small labeled sets, and potentially better predictions on the labeled data as well when labeled data is noisy. Moreover, it turns out that many real-world situations fit the data profile required by SSL: a relatively small amount of labeled data plus a large amount of unlabeled data.

An advantage shared by most graph-based SSL algorithms (except for the simple mincut algorithm [27]) is that they treat both label inputs and label outputs as real, continuous values. This is not self-evident in all formulations of the algorithms. For example, the canonical description of the label propagation algorithm uses discretization of training labels by representing labels as Kronecker vectors $\delta_\ell(\mathtt{y}_k)$, as shown in Eq. 2.3.

Also, output is often discretized too, by means of $\arg\max$ selection. A natural generalization is to use soft labels on input (non-degenerated label probability distributions for the labeled samples) and soft labels on output (skip the $\arg\max$ step). In the case of modeling a continuous function, one label suffices; the quadratic cost function (Eq. 2.23) ensures a good-quality regression—assuming

the graph reflects similarity across samples accurately. Goldberg and Zhu [93] exploit this property to learn a continuous ranking function starting from discrete values (discrete ratings of one to four stars). They use one label with continuous values, initialized with natural numbers in $\{0, 1, 2, 3\}$ representing movie ratings. The label propagation algorithm regresses a continuous function underlying the ratings, and the final step rounds the function to return results in the same form as inputs. Agarwal [3] describes a semi-supervised method to learn a hierarchical ranking function. Such versatility of the learned function opens the door to new applications, such as n-best list rescoring in NLP applications: after labeling with soft labels, test samples can be sorted in increasing order of label value.

On the other hand, constructing the graph is an empirical process that reflects researcher's understanding of the domain. Graph construction is highly sensitive to the choice of similarity measure and its parameterization (e.g. α in Eq. 2.1 and the maximum number of connected neighbors). There is little theory helping the choice of a similarity measure, which suggests that for many feature spaces applications make suboptimal choices. Moreover, in HLT applications, many features are discrete and heterogeneous (word, part-of-speech, root, stem, various counts, presence/absence of a characteristic, etc.), and it is unclear how a smooth distance measure can be computed over such feature sets.

Also, the issue of scalability in semi-supervised learning has so far received an ad-hoc treatment. Graph construction prescribes one graph vertex for each sample, and sometimes the graph construction process creates even more vertices to model e.g. additional knowledge sources [93]. A usual method for increasing scalability is to make the matrix W sparse by imposing a k nearest neighbors or an ϵ-radius neighborhood. However, there is little systematic study of similarity measures that are at the same time scalable (such as slow-growing metrics [114]) and generally suitable for constructing good similarity graphs. Moreover, even if the number of edges per vertex is artificially limited, the sheer number of vertices could still be problematic for storing the graph in working memory.

Chapter 3

GRAPH CONSTRUCTION

As mentioned in Chapter 2, constructing an accurate similarity graph is the most important step in achieving good results with graph-based algorithms. Although many algorithms exploiting graph-based structures exist, the issue of graph construction has remained an empirical and crafty process that has forced each application to develop its own heuristic methods to overcome this difficult step. The fidelity with which the graph reflects similarities among samples influences successful application of graph-based methods much more than the particulars of the learning algorithm applied to the graph. This dependence on task-specific preprocessing discourages wide, generic use of graph-based learning. In contrast, other machine learning methods—such as neural networks, support vector machines, or Gaussian mixture models—are more amenable to direct usage with lightly-preprocessed features using standard tools. A recent survey on semi-supervised learning literature [239, § 6.1] notes: "We believe it is more important to construct a good graph than to choose among the methods. However graph construction [...] is not a well studied area." Other recent work [238, pp. 9] also mentions: "A good graph should reflect our prior knowledge about the domain. At the present time, its design is more of an art than science."

In this chapter we present novel approaches to graph construction, with a focus on choosing the similarity measure and on reducing the time complexity of the construction step. Most importantly, we propose a hybrid two-staged system using two distinct classifiers. The first classifier is trained to predict probability distributions over the label set $\{1, \ldots, \ell\}$, and the second (the graph-based learner) uses the probability distributions as its input features. We explain how this setup leads to good-quality graphs because it obviates many difficulties in choosing a similarity measure.

3.1 Similarity

The quality of a similarity graph is determined by the choice of similarity measure among samples. A good similarity measure should obviously be *indicative*, meaning that two highly similar samples are correspondingly likely to bear the same label. But a good similarity measure should also be *smooth*, i.e., similarity should vary smoothly, without discontinuities, from highly similar samples to less similar samples; in other words, similarity should convey confidence information. This is because the extent to which two samples are similar or dissimilar is very important in obtaining a rich, expressive graph that allows labeling not only by means of direct similarity, but also by propagation into neighborhoods. A non-smooth similarity measure will create a graph in which clusters have small volume and high density, whereas test points that are noisy, non-confident, or slightly off the predicted distribution would be far away from any cluster and therefore hard to classify correctly.

Let us analyze qualitatively how smoothness affects graph quality, operating on an extreme example. Consider working with a similarity measure σ_{01} that is discretized from an expert estimate

as follows:

$$\sigma_{01}(\mathtt{x}, \mathtt{x}') = \begin{cases} 0 & \text{if expert predicts } \hat{\mathtt{y}} \neq \hat{\mathtt{y}}' \\ 1 & \text{if expert predicts } \hat{\mathtt{y}} = \hat{\mathtt{y}}' \end{cases} \tag{3.1}$$

where $\hat{\mathtt{y}}, \hat{\mathtt{y}}' \in \{1, \ldots, \ell\}$ are discrete label predictions. Such a measure seems rather uninteresting to use for learning. If it is sometimes unreliable then it conveys no information about the confidence of the prediction; if it is of excellent quality then it obviates the learning process in the first place. However, averaging similarities over a large number of edges endows the discrete similarity with smoothness information at the cost of a denser graph—and consequently one that requires more computation during label propagation. Given that each edge is less informative when using a coarse smoothness, more data points and more edges are necessary for defining a good-quality graph. So ultimately a non-smooth similarity measure is still workable if there is enough data to bring smoothness information from the edge mesh. Fundamentally, more edges of unit weight approximate fewer edges with real weight. In the interest of graph construction time, however, we are interested in keeping the similarity graph sparse, which leads us to the conclusion that a smooth similarity measure is needed for fast graph construction. We will define the needed smoothness criterion for the similarity measure later in this chapter.

3.2 Distance vs. Similarity

Consider that a choice of features has been made and a similarity measure is to be defined. For certain feature sets, defining a similarity directly is a natural process with strong intuitive backing. In fact feature sets amenable to intuitive similarity definitions are easy to find in HLT. Consider, for example, using variable-length strings of tokens (such as, but not limited to, words, characters, or syllables) directly as features. The feature space is therefore the Kleene closure [147] Σ^* over some alphabet Σ. Such features are not fixed-sized vectors and are best compared directly for similarity through partial and approximate matching. The BLEU score [180] is a widely-used similarity measure built around n-gram co-occurrence. Various string kernels allowing for partial matches and gaps have recently received increasing attention. Such kernels, again, compute a similarity measure directly. Chapter 4 uses a string kernel on a Machine Translation experiment.

However, in many other cases, in HLT as in other domains, features are fixed-length vectors of real numbers (e.g. MFCC[1] vectors, frequency of occurrence, or even scores computed by a complementary system) and/or categorical tags and Boolean values (e.g. word in a vocabulary, part of speech, capitalization information). In such cases it is often useful to reframe choosing a *similarity* measure into choosing a *distance* measure. This is because vector distances are better studied and understood; vectorial feature spaces are most often characterized by distance measures, not by similarity measures. Similarities are then obtained from distances through a Gaussian kernel. Given a distance $d : \mathtt{X} \times \mathtt{X} \to \mathbb{R}_+$, a Gaussian kernel defines a family of similarity measures σ_α as follows [238]:

$$\sigma_\alpha : \mathtt{X} \times \mathtt{X} \to (0, 1] \quad \sigma_\alpha(\mathtt{x}_i, \mathtt{x}_j) = \exp\left[-\frac{d(\mathtt{x}_i, \mathtt{x}_j)^2}{\alpha^2}\right] \tag{3.2}$$

[1] MFCC stands for Mel Frequency Cepstral Coefficients, the dominant representation of speech data today.

where α is a bandwidth hyperparameter, usually optimized experimentally. (Some authors use $2\alpha^2$ instead of α^2, but that is just a convention meant to simplify certain equations.) For a given node $\mathtt{x}$, the partial application $\sigma_\alpha(\mathtt{x}, \cdot)$ defines a Gaussian radial basis function with origin in $\mathtt{x}$. Although there is no proof that σ_α is the optimal way of converting a distance to a similarity measure, strong empirical evidence shows it to be an appropriate choice. Shepard has argued [201] that similarity—at least as defined by, and as applicable to, experimental cognitive science—has an inverse exponential relationship to distance, conjecture known as the Universal Law of Generalization. This law has been confirmed in numerous cognitive experiments involving human and non-human subjects, such as confusion between linguistic phonemes [162], sizes of circles [157], spectral hues as perceived by people [75] and pigeons [98], and spatial generalization by bees [46]. In all cases experiments have confirmed a dependency of perceived similarity to distance in the form of an inverse exponential. Chater and Vitányi [43] further argued, with additional experimental evidence, that a better definition of similarity makes it proportional to the inverse exponential of the squared distance, which follows Eq. 3.2. They have argued that the same relation holds for non-Euclidean distances as well, providing further empirical evidence for using the Gaussian kernel to convert distances to similarities. (However, this does not imply that the Gaussian kernel is optimal for a graph-based algorithm, which may act very different than the human perceptual system.)

The Gaussian kernel decreases monotonically with distance. The hyperparameter α controls the bandwidth or resolution of the kernel by deciding how close two points have to be in order to be considered similar. Small values of α make the kernel highly selective, at an extreme forcing most test samples at uninformatively high distances from all other samples. Large values of α engender the opposite effect of "crowding" the space by making samples indistinguishably similar with one another. Figure 3.1 illustrates the σ_α function for various values of α.

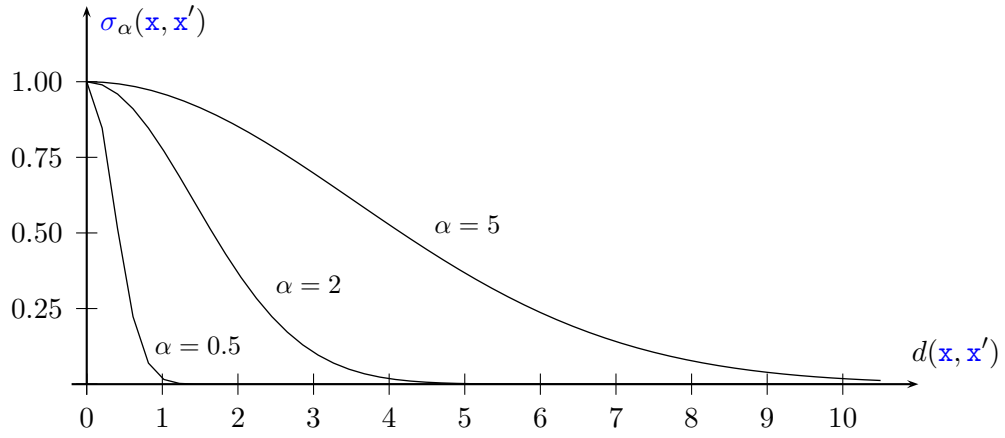


Figure 3.1: The Gaussian kernel used for converting distances to similarities. The value of the hyperparameter α controls the aperture of the similarity window.

3.2.1 Distance Measures

The σ_α function is used to initialize the adjacency matrix of the similarity graph $\mathtt{W}$ directly by assigning $\mathtt{w}_{ij} = \sigma_\alpha(\mathtt{x}_i, \mathtt{x}_j)$. What is needed then is a distance measure d that computes an estimated distance between two samples, and an appropriate choice for hyperparameter α. The distance measure does not need to be a metric; positivity and symmetry are the only *prima facie* requirements. Good accuracy for close-by samples is required for constructing a good graph, but not at far range because similarity decays exponentially with distance. As mentioned in § 3.1, continuity of d is also highly necessary for creating a good-quality graph.

In the absence of a principled method, generic distance measures for vectors are often used, although it is understood they may not be optimal.

Minkowski Distance An obvious candidate is one of the Minkowski distance measures of order p:

$$L^p(\mathbf{a}, \mathbf{b}) \triangleq \left(\sum_{i=1}^{F} \left| \mathbf{a}_{[i]} - \mathbf{b}_{[i]} \right|^p \right)^{1/p} \tag{3.3}$$

where F is the dimensionality of feature vectors, and $\mathbf{a}_{[i]}$ is the i^{th} slot of vector $\mathbf{a}$. Minkowski distances include the well-known and often-used Manhattan distance L^1 and Euclidean distance L^2. A fundamental problem with Minkowski distances in heterogeneous spaces is that the unit of measure on each dimension influences the outcome, which makes it difficult to choose a proper unit for each dimension. The quantities across dimensions may be largely different or not even comparable because they have different types (e.g. a real-valued vs. a discrete variable, or a frequency value vs. an amplitude value). The practical negative consequence is that in a heterogeneous space, one of the dimensions might easily dominate all others and essentially decide single-handedly the magnitude of the distance. Therefore, a per-dimension normalization becomes necessary:

$$L^p_\alpha(\mathbf{a}, \mathbf{b}) = \left(\sum_{i=1}^{F} \alpha_i \left| \mathbf{a}_{[i]} - \mathbf{b}_{[i]} \right|^p \right)^{1/p} \tag{3.4}$$

The α_i coefficients are often chosen such that they ensure equal spread (e.g. standard deviation or range) in each dimension. That choice, however, may still be suboptimal because some features might be more indicative than others.

Cosine Distance A simple way to avoid relative magnitude issues is to use a distance define as one minus cosine similarity, quantity often referred to as "cosine distance:"

$$d(\mathbf{a}, \mathbf{b}) = 1 - \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \cdot \|\mathbf{b}\|} = 1 - \frac{\sum_{i=1}^{F} \mathbf{a}_{[i]} \mathbf{b}_{[i]}}{\sqrt{\sum_{i=1}^{F} \mathbf{a}^2_{[i]} \sum_{i=1}^{F} \mathbf{b}^2_{[i]}}} \tag{3.5}$$

Cosine distance depends only on the cosine of the angle between the two feature vectors $\mathbf{a}$ and $\mathbf{b}$, quantity independent on the magnitude of the vectors. Ideally the distance would be equally

sensitive in all directions. Therefore, cosine distance works best in feature spaces where features are homogeneous and orthogonal [148]. For many feature sets these properties are not guaranteed, so it is likely that cosine distance is suboptimal. Still, cosine distance is often the distance measure of choice in the absence of a proper understanding of the feature space because it is computationally inexpensive and performs well on many tasks.

3.3 Data-Driven Graph Construction

To construct a quality graph, an optimal distance measure should be used. The truth of the matter is that in the intricate feature spaces met in HLT applications we do not generally have principled criteria for choosing one particular distance measure (e.g. Minkowski or Cosine distance), nor do we have a formal means to preprocess features in ways that make them provably amenable to a particular distance measure. Therefore, our decision is to *learn* a representation of the feature space that makes it easy to define an optimal distance.

We propose an empirical data-driven technique for graph construction. This approach is central to all of our applications of graph-based learning to HLT. The technique involves a two-pass system employing two classifiers. First, a supervised classifier is trained on the labeled subset to transform the initial feature space (consisting of e.g. lexical, contextual, or syntactic features) into a homogeneous and continuous representation in the form of soft label predictions. The soft label predictions are (predicted) probability distributions over labels, that is, vectors of length ℓ containing positive real numbers that sum to 1. Then, the soft label predictions are used as *feature vectors* by the graph-based semi-supervised learner in conjunction with a similarity measure specialized for probability distributions. In effect, a supervised predictor is employed as a feature transformation device by the graph-based engine.

Figure 3.2 summarizes the structure of the two-pass classifying system.

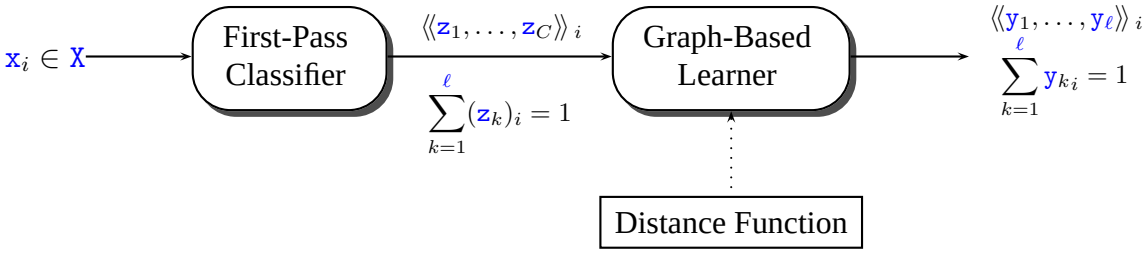


Figure 3.2: Structure of a two-pass learning system. The first-pass classifier accepts original features $\mathtt{x}_i \in \mathtt{X}$ and outputs probability distribution estimates $\mathtt{z}_i$ over labels. The graph-based learner uses these estimates as input features in conjunction with a distance function that is suitable for probability distribution spaces.

Before explaining how this choice is useful, more detail on the setup is in order. The first-pass

classifier learns a function

$$Z : \mathtt{X} \rightarrow [0,1]^{\ell} \quad Z(\mathtt{x}) = \langle\langle \mathtt{z}_1, \ldots, \mathtt{z}_{\ell} \rangle\rangle \tag{3.6}$$

$$\sum_{i=1}^{\ell} \mathtt{z}_{[i]} = 1 \tag{3.7}$$

The Z function is the usual classification function that the supervised classifier learns, such as the softmax output of a neural network, the normalized output of a Gaussian mixture model, or the outputs of a Support Vector Machine (SVM) fitted to a sigmoid function [182, 145].

The representation obtained at the output of the first-pass classifier is then used as a basis for measuring similarity among samples that determines the structure of the graph used for the second, semi-supervised learning step. This approach bears commonalities and differences with previously-proposed approaches as follows:

- Like cascading classifiers [5], the proposed data-driven learner uses two classifiers. The cascaded classifiers approach first uses a simple and comprehensive classifier. If that classifier makes a low-confidence decision, the second classifier—specialized in handling exceptions and possibly more computationally-intensive—is consulted. Unlike cascading classifiers, our proposed learner uses the two classifiers in *conjunction*, not in *disjunction*. Our system runs the second classifier using the first classifier's outputs as input, and the classification decision is always taken at the output of the second classifier. Other differences include use of a semi-supervised learner instead of two supervised learners.

- Similarly to principal component analysis (PCA) [202], the proposed approach transforms the input feature into an intermediate format. Unlike PCA which is an unsupervised method, the proposed approach uses the labels to train the feature transformation engine in a supervised manner.

- Several proposed approaches [67, 148, 207] learn a distance or similarity measure directly. Our approach is different in that it learns a feature representation that simplifies the choice of distance measure. The learner for the transformed features needs to train on the $\mathtt{t}$ input samples, whereas a supervised system that learns a distance measure must train on the considerably more numerous $\frac{\mathtt{t}(\mathtt{t}-1)}{2}$ pairs of samples. (In particular cases computational cost can be, however, reduced [148].)

The key advantage of using a first-pass classifier is that it moves the problem of defining a distance measure from a heterogeneous space to a homogeneous space of probability distributions. The next section is an overview of distance measures in that space.

3.4 Distance Measures for Probability Distributions

After the features have been transformed into probability distribution vectors, a variety of distance functions that are more or less specialized can be applied. The Gaussian kernel in Eq. 3.2 is applicable on top of any such distance.

Below we discuss a few distance measures used for probability distributions, along with a few properties of particular interest:

1. *Non-negativity:* $d(\mathbf{a}, \mathbf{b}) \geq 0 \; \forall \mathbf{a}, \mathbf{b} \in \mathcal{X}$
2. *Indiscernibility is identity:* $d(\mathbf{a}, \mathbf{b}) = 0 \Leftrightarrow \mathbf{a} = \mathbf{b} \; \forall \mathbf{a}, \mathbf{b} \in \mathcal{X}$
3. *Symmetry:* $d(\mathbf{a}, \mathbf{b}) = d(\mathbf{b}, \mathbf{a}) \; \forall \mathbf{a}, \mathbf{b} \in \mathcal{X}$
4. *Triangle inequality:* $d(\mathbf{a}, \mathbf{b}) \leq d(\mathbf{a}, \mathbf{c}) + d(\mathbf{c}, \mathbf{b}) \; \forall \mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathcal{X}$

Distance measures that satisfy all four properties are called metrics. As discussed, graph-based learning only requires non-negativity and symmetry. However, there is an additional motivation to pursue distances that satisfy all metric properties. This is because may algorithms that approximate the graph's connectivity matrix $\mathtt{W}$ with the nearest neighbors of each sample requires that the distance measure is a metric. The first three properties are naturally fulfilled by most distance measures; it is the triangle inequality that may not always be satisfied.

3.4.1 Cosine Distance

Cosine distance—which has already been mentioned above (Eq. 3.5)—should work properly on probability distributions. Assuming the first-pass learner did learn a good discrimination function, the outputs for different labels $\mathtt{y}$ and $\mathtt{y}'$ will be close to the Kronecker vectors $\delta_\ell(\mathtt{y})$ and $\delta_\ell(\mathtt{y}')$, respectively. Such vectors are orthogonal and therefore are distanced at 1 (the maximum value of cosine distance), whereas identical vectors are distanced at 0. It is worth noting that both cosine distance and cosine similarity are sometimes confusingly referred to as "cosine metric" although neither satisfies the triangle inequality. It is easy to prove that the cosine distance is the square of a metric [129] and that one minus squared cosine similarity is also the square of a distance, proposed under the suggestive name of "sine distance" [44].

3.4.2 Bhattacharyya Distance

The Bhattacharyya distance is defined as:

$$d_{BC}(\mathbf{a}, \mathbf{b}) = -\log \sum_{i=1}^{\ell} \sqrt{\mathbf{a}_{[i]} \mathbf{b}_{[i]}} \tag{3.8}$$

The Bhattacharyya distance is positive and reaches zero only for identical distributions. The quantity under $\log$ is also called the Bhattacharyya coefficient:

$$BC(\mathbf{a}, \mathbf{b}) \triangleq \sum_{i=1}^{\ell} \sqrt{\mathbf{a}_{[i]} \mathbf{b}_{[i]}} \tag{3.9}$$

$BC(\mathbf{a}, \mathbf{b})$ can be zero, which makes the Bhattacharyya distance unbounded—it diverges to infinity whenever at least one of the distributions is zero in each component. Bounding is not required

but is a very useful property of a distance measure, particularly for practical reasons (numeric stability). Bounding can be achieved by smoothing the two distributions prior to measuring distance, for example by parameterized interpolation with the uniform distribution:

$$\mathbf{s}_{\alpha[i]} = \alpha\mathbf{a}_{[i]} + (1-\alpha)\frac{1}{\ell} \tag{3.10}$$

The parameter α can be chosen on numeric grounds as:

$$\alpha = \frac{m}{\ell} \tag{3.11}$$

where m is the minimum admissible value of $BC(\mathbf{a}, \mathbf{b})$. It is easy to show that for that value of α and for $m < 1$, $BC(\mathbf{z}, \mathbf{z}') > m$.

The Bhattacharyya distance is symmetric and reflexive but does not obey the triangle inequality. The Bhattacharyya coefficient also does not obey the triangle inequality, but $\sqrt{1 - BC(\mathbf{a}, \mathbf{b})}$ does. This fact motivates the Hellinger distance, which is discussed below.

3.4.3 The Hellinger Distance

The Hellinger distance [184], sometimes called the Jeffries-Matusita distance, is defined as:

$$d_H(\mathbf{a}, \mathbf{b}) = \sqrt{\frac{1}{2}\sum_{i=1}^{\ell}\left(\sqrt{\mathbf{a}_{[i]}} - \sqrt{\mathbf{b}_{[i]}}\right)^2} \tag{3.12}$$

The Hellinger distance is positive and reaches zero only for equal distributions. Also, it is bounded to a maximum value of 1. Note that authors may use other multiplication constants in defining the Hellinger distance instead of $\frac{1}{2}$; we chose the constant that sets its range to $[0, 1]$. There is an easily verifiable relationship between the Hellinger distance and the Bhattacharyya coefficient:

$$d_H(\mathbf{a}, \mathbf{b}) = \sqrt{1 - BC(\mathbf{a}, \mathbf{b})} \tag{3.13}$$

As mentioned above, the Hellinger distance satisfies the triangle inequality. It is also reflexive and symmetric, so it defines a metric over probability distributions.

3.4.4 Kullback-Leibler Divergence (and Symmetrized Variant)

The Kullback-Leibler divergence is specific to probability distributions. In the discrete case, Kullback-Leibler divergence is defined as:

$$d_{\mathsf{KL}}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{\ell}\mathbf{a}_{[i]}\log\frac{\mathbf{a}_{[i]}}{\mathbf{b}_{[i]}} \tag{3.14}$$

In addition to being solidly motivated in information theory, the Kullback-Leibler divergence has many desirable properties. By Gibbs' inequality [108], $d_{\mathsf{KL}}(\mathbf{a}, \mathbf{b}) \geq 0$, and equality is reached if and only if the distribution are point-wise equal: $d_{\mathsf{KL}}(\mathbf{a}, \mathbf{b}) \Leftrightarrow \mathbf{a} = \mathbf{b}$. However, Kullback-Leibler

divergence is not symmetric: $d_{\mathsf{KL}}(\mathbf{a}, \mathbf{b}) \neq d_{\mathsf{KL}}(\mathbf{b}, \mathbf{a})$ and therefore difficult to use as a distance measure in graph-based learning.

Symmetry can be achieved in many ways, one of them being simply adding $d_{\mathsf{KL}}(\mathbf{a}, \mathbf{b})$ and $d_{\mathsf{KL}}(\mathbf{b}, \mathbf{a})$ [167]:

$$d_{\mathsf{SKL}}(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{\ell} \left(\mathbf{a}_{[i]} \log \frac{\mathbf{a}_{[i]}}{\mathbf{b}_{[i]}} + \mathbf{b}_{[i]} \log \frac{\mathbf{b}_{[i]}}{\mathbf{a}_{[i]}} \right) \tag{3.15}$$

$$= \sum_{i=1}^{\ell} \left[\left(\mathbf{a}_{[i]} - \mathbf{b}_{[i]} \right) \log \frac{\mathbf{a}_{[i]}}{\mathbf{b}_{[i]}} \right] \tag{3.16}$$

This is in fact how Kullback and Leibler originally defined the divergence.

Another issue with the Kullback-Leibler divergence is that it is not bounded—it diverges to infinity whenever one component in $\mathbf{b}$ distributions predicts near-zero probability and the corresponding component in $\mathbf{a}$ does not. Also, there is an obvious requirement for d_{SKL} to be well-defined $\mathbf{b}_{[i]} = 0$ wherever $\mathbf{a}_{[i]} = 0$ (i.e., the distributions must be *absolutely continuous* with respect to each other [133]). Similarly to Bhattacharyya distance, bounding can be obtained through smoothing by e.g. interpolating both distributions with the uniform distribution. The interpolation factor α (see Eq. 3.10) can be chosen as:

$$\alpha = 1 - \ell e^{-M} \tag{3.17}$$

where M is the maximum admissible value of $\log \frac{\mathbf{z}_{[i]}}{\mathbf{z'}_{[i]}}$.

A method that achieves symmetrizing and smoothing simultaneously is the Jensen-Shannon divergence, discussed next.

3.4.5 Jensen-Shannon (Symmetrized Smoothed Kullback-Leibler) Divergence

The Jensen-Shannon divergence [163, 151], introduced independently by Rao [185] and Lin [146], is defined as

$$d_{\mathsf{JS}}(\mathbf{a}, \mathbf{b}) = \frac{d_{\mathsf{KL}}(\mathbf{a}, \mathbf{m}) + d_{\mathsf{KL}}(\mathbf{b}, \mathbf{m})}{2} \tag{3.18}$$

where $\mathbf{m}$ is the equal-weight interpolation of $\mathbf{a}$ and $\mathbf{b}$:

$$\mathbf{m}_{[i]} = \frac{\mathbf{a}_{[i]} + \mathbf{b}_{[i]}}{2} \tag{3.19}$$

The Jensen-Shannon (J-S) divergence also has useful interpretations in information theory. It is symmetric, bounded to $[0, 1]$, and defined for any two distributions. Although the Jensen-Shannon divergence is not a metric, it has been shown that is the square of a metric [77].

3.5 Joint Optimization of the First- and Second-Pass Classifiers

The combination of first-pass classifier and graph-based learner is at best globally optimized with respect to the properties required of the graph. From the viewpoint of the graph-based learner,

a good feature space is filled with well-defined clusters that also have enough "fuzziness" at the borders to provide adaptation to unseen data. It is worth noting that after the feature transformation performed by the first-pass classifier, data does not reside on a manifold anymore. This is because the dimensionality of the transformed features is exactly ℓ, the same as the number of distinct labels. This representation is often much more compact and almost always more homogeneous than the original feature space.

3.5.1 Regularization of the First-Pass Classifier

Regularization of the first-pass classifier is essential in training a good combined system. This is because often an un-regularized classifier will output very sharp, confident distributions. As discussed in § 3.1, a discretized, discontinuous similarity measure (obtained by necessity from an equally discretized distance measure) is detrimental to the graph-based learner. A space with few and highly-concentrated clusters will not make it possible to predict good labels for data falling in its large interstices. That is why the indecision of the first-pass classifier is important: The less confident predictions establish fuzzy cluster borders and "fill" the feature space with informative attractors.

Regularization [168] is a common class of techniques aimed at improving generalization of classifiers. Generally, regularization introduces a term in the learner's objective function that penalizes complex learners. The actual penalty depends on the learner—e.g. number of model parameters, magnitude of parameters, or conditioning using priors.

In a lexicon learning application (§ 3.6) we use L_2 regularization during training the first-pass classifier, a neural network.

3.5.2 Adding and Mixing In Synthesized Data

An advantage conferred by operating in a transformed space is that data points can be easily synthesized. For example, the Kronecker vector $\delta_\ell(\mathtt{y})$ is the ideal, "golden" data point that predicts label $\mathtt{y}$ with maximum confidence. In contrast, a uniform vector would be a point of high indecision. Features can also be adjusted and combined: the normalized linear combination of feature vectors is also a feature vector. Generating meaningful, highly indicative feature vectors is not possible for many learning problems. Also, feature preprocessing is usually done in a problem-specific manner.

The output of the first-pass classifier can be manipulated for the purpose of e.g. adaptation or smoothing. It is easy to place Kronecker vectors that act as attractors toward the hard labels. We have implemented several such techniques in a word sense disambiguation application described in § 3.7 and in an acoustic classification application described in § 3.8.

3.6 Application: Lexicon Learning

We applied the two-pass classifier described above to a part-of-speech (POS) lexicon acquisition task, i.e. the labels to be predicted are the sets of POS tags associated with each word in a lexicon. Note that this is *not* a tagging task: we are not attempting to identify the correct POS of each word in running text. Rather, for each word in the vocabulary, we attempt to infer the set of *possible* POS tags. Our choice of this task is motivated by the goal of applying this technique to lexicon

acquisition for resource-poor languages: POS lexicons are one of the most basic language resources, which enable subsequent training of taggers, chunkers, etc.

Due to homonymy and polysemy, the same written word often corresponds to several meanings, and in particular—most importantly for this task—some of these meanings may map to different parts of speech. Examples are readily available in all human languages; for example, in English, the word "sport" may, depending on the context in which it's used, mean several *verbs* synonymous with "to frolic," "to trifle," "to mutate," or "to boast." To these we add a few *noun* senses, such as "athletic game," and also an *adjective* sense, as in "sport shoes." Distinguishing exactly which meaning was used in a particular context is a task called word sense disambiguation, which is the subject of another experiment described later in this paper. For now, we will set out to a somewhat lesser goal, that of deducing the possible parts of speech of all words in the lexicon of an initially unknown language. This step, albeit small, is crucial in developing higher-level linguistic tools, including word sense disambiguators themselves.

The setup for lexicon learning is as follows. We assume that a small set of words can be reliably annotated by human annotators. From those labeled words, we infer POS-sets for the remaining words by semi-supervised learning. For example, for the word "sport," the correct outcome of a POS learner would be:

```
sport: NOUN VERB ADJ
```

meaning that in English text, "sport" may be a noun, a verb, or an adjective. What is missing is as important as what is present—there are no other possible parts of speech for the word "sport."

Rather than choosing a genuinely resource-poor language for this task, we use the English Wall Street Journal (WSJ) corpus and artificially limit the size of the labeled set. This is because the WSJ corpus is widely obtainable and allows easy replication of our experiments. The eventual application would target a resource-poor language such as dialectal Arabic, in which case the labeled and unlabeled data might follow less favorable distributions: In the case of a dialect, the labeled subset would correspond to the standard language, and the unlabeled set would consist of the dialect-specific words.

We use sections 0–18 of the WSJ corpus. The number of unique words and thus the total number of samples is $\mathtt{t} + \mathtt{u} = 44\,492$. A given word may have between 1 and 4 POS tags, with an average of 1.1 per word. The number of POS tags is 36, and we treat every POS combination as a unique class, resulting in $\ell = 158$ distinct labels. In order to study the influence of the training set size on the semi-supervised effect, we use three different randomly selected training sets of various sizes: $\mathtt{t} = 5\,000$, $\mathtt{t} = 10\,000$, and $\mathtt{t} = 15\,000$ words, representing about 11%, 22%, and 34% of the entire data set respectively; the rest of the data was used for testing. In order to avoid experimental bias, we run all experiments on five different randomly chosen labeled subsets and report averages and standard deviations.

Due to the random sampling of the data it is possible that some labels never occur in the training set or only occur once. We train our classifiers only on those labels that occur at least twice, which results in 60–63 classes. Labels not present in the training set will therefore not be hypothesized and are guaranteed to be errors. We delete samples with unknown labels from the unlabeled set since their percentage is less than 0.5% on average. This decision is in keeping with a real-world scenario in which human annotators label a training corpus because in that case the selected training corpus would be representative of the language, not random.

Table 3.1 shows the features used to represent words for the purpose of lexicon learning. The categorical features are obtained by extracting the relevant words or word fragments from the training set, indexing them in a dictionary (one dictionary for each of features F_1 through F_6) and then using their index. A special symbol is allocated for an unseen dictionary entry. This case may be encountered rather frequently, particularly for small training set sizes. We have also experimented with shorter suffixes and also with prefixes but those features tended to degrade performance.

#	Feature	Type
F_1	The three-letter suffix of the word	Categorical
F_2	The four-letter suffix of the word	Categorical
$F_{3..6}$	The 4 most frequent words that immediately precede the word in text	Categorical $\times$ 4
F_7	Word contains capital letters	Boolean
F_8	Word consists only of capital letters	Boolean
F_9	Word contains digits	Boolean
F_{10}	Word contains one or more hyphens	Boolean
F_{11}	Word contains other special characters (e.g. "&")	Boolean

Table 3.1: Features used for lexicon learning.

Qualitatively, the choice of feature $F_{3..6}$ relies on the generally applicable supposition that a given part of speech tends to occur within similar contexts. Features F_1 and F_2 assume that words of a given POS tend to have the same suffix—a more language-dependent supposition. In any case, although it is easy to justify the choice of all features, they are not orthogonal. For example, if F_5 is true then F_4 is also true. Other features are also strongly correlated, for example F_1 and F_2.

3.6.1 The First-Pass Classifier

For the lexicon learning task, the first-pass classifier is a multi-layer perceptron (MLP) with the topology shown in Fig. 3.3. We discuss the MLP topology below in flow order: the adaptation layer A, the continuous mapping layer M, and then the layers i, h, and o.

3.6.1.1 The approximation layer A

As mentioned, at a minimum, we train the neural network with only 5000 labeled samples that were selected at random from the corpus. This is a scarce scenario especially considering that some features occur rather infrequently (for example F_8 or F_{11} in table 3.1). Severe problems caused by data scarcity arise when some of the input features of the unlabeled words have *never* been seen in the training set. For such samples the neural network reads untrained, randomly-initialized[2] weight values and consequently outputs arbitrary label predictions. It is technically easy to eliminate

[2]Neural network initial weights are customarily initialized with small random values. A neural network trained with discrete inputs may never update some weights if certain inputs are never seen.

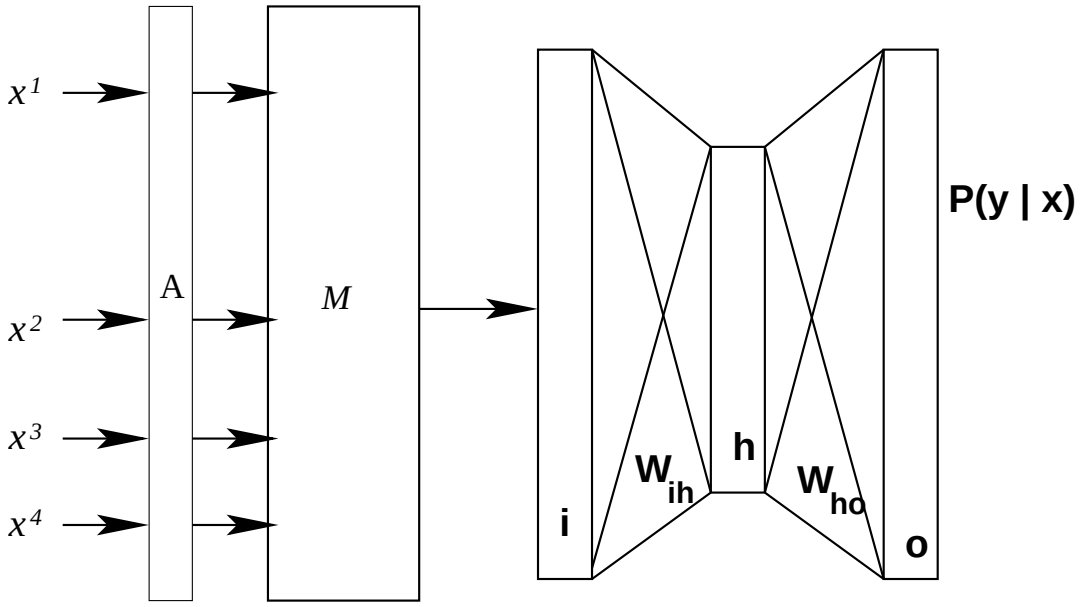


Figure 3.3: Architecture of first-pass supervised classifier (MLP) for lexicon acquisition.

randomness by overwriting untrained values to e.g. zero after training, but the fact remains that the neural network makes a meaningless prediction.

The problem of unseen patterns is of course encountered in all neural networks. What makes this case different is the presence of categorical features. A neural network working with continuous inputs can make a meaningful decision on an unseen pattern through the assumed continuity of the classification function (similar inputs produce similar outputs). In the case of categorical features, there is no continuity to be invoked, so an unseen categorical feature taps into an entirely untrained portion of the neural network.

We address this problem by creating an approximation layer A. During training, A stores all seen unique patterns in a hash table keyed by the concatenation of categorical inputs. During testing, A loads its state and watches for never-seen features. Let us assume that feature $\mathtt{x}_{[k]}$ had never been encountered during training. In that case, A finds the known input feature vector $\mathtt{x}'$ that is most similar to $\mathtt{x}$ (by measuring the Hamming distance between the vectors). Then $\mathtt{x}_{[k]}$ is replaced with $\mathtt{x}'_{[k]}$, resulting in vector $\hat{\mathtt{x}} = \langle\langle \mathtt{x}_{[1]}, \ldots, \mathtt{x}_{[k-1]}, \mathtt{x}'_{[k]}, \mathtt{x}_{[k+1]}, \ldots, \mathtt{x}_{[F]} \rangle\rangle$ that has no unseen features and is closest to the original vector.

3.6.1.2 The discrete-to-continuous mapper M

The input features are mapped to continuous values by a discrete-to-continuous mapping layer M. This layer is equivalent to a vertical concatenation of classic neural network layers operating on so-called one-hot inputs, in a setup customarily used for neural networks with categorical inputs [15, 14], which we describe below.

One-hot encoding is a simple method of adapting categorical data for use as neural network

inputs. If a categorical feature can take N distinct values, presenting an integer in $\{1, \ldots, N\}$ in lieu of a real number at the input of the MLP would be mistaken because it introduces artifact magnitude and ordering among samples. For example, the neural network "thinks" that values 1 and N are much farther apart than values $N-1$ and N and tries to learn a smooth classification function under that assumption. However, categorical values should be equally distinct (apart) from one another, and the learning process should be immune from the particular natural numbers assigned to the input categories. Therefore, categorical inputs are commonly encoded through the following mapping function:

$$H : \{1, \ldots, N\} \to \{0, 1\}^N \quad H(i) = \delta_N(i) \tag{3.20}$$

where $\delta_N(i)$ is the Kronecker vector of length N with 1 in position i and 0 elsewhere. The representation obtained this way is called the one-hot vector encoding of the categorical value. When using one-hot encoding, the Hamming distance between any two distinct inputs is the same, and therefore the result of the learning process does not depend on the particular mapping of categorical values to numbers in $\{1, \ldots, N\}$.

Let us analyze the transfer function for a neural network layer operating on a one-hot-encoded input. The transfer function of a neural network layer can be generally expressed as:

$$f : \mathbb{R}^N \to \mathbb{R}^{N'} \quad f(H) = \overrightarrow{\phi}\,(H\omega + B) \tag{3.21}$$

where N' is the number of outputs of the layer (fixed at system design time), H is the input (in our case a one-hot vector), $\omega \in \mathbb{R}^{N \times N'}$ is the weights matrix, $B \in \mathbb{R}^{N'}$ is a bias vector (both ω and B are learned model parameters), $\phi : \mathbb{R} \to \mathbb{R}$ is the activation function (chosen during design), and $\overrightarrow{\phi}\,(A)$ applies function ϕ to each element of vector A. It would appear that using one-hot encoding is memory- and computationally-wasteful because it makes a single input occupy an N-dimensional vector. For large values of N, the multiplication would be computationally intensive when implemented directly. However, we can use the information that H is a one-hot vector in rewriting the layer's transfer function (after eliminating all zero terms) as follows:

$$f : \{1, \ldots, N\} \to \mathbb{R}^{N'} \quad f(i) = \overrightarrow{\phi}\,(\omega_{\text{row } i} + B) \tag{3.22}$$

So a simple method of obtaining the output of a one-hot vector coupled to a neural network layer is to simply add the i^{th} row of matrix ω to the biases vector and then apply ϕ to each element of the result. There is no more intermediate one-hot vector to use and no more expensive matrix-vector multiplication.

There are two further simplifications we make to the transfer function. Given that we already have biases and a nonlinear activation function in the downstream hidden layer, in this layer we choose ϕ to be the identity function and we do not use a biases vector, so the transfer function for the continuous mapper simplifies down to:

$$f : \{1, \ldots, N\} \to \mathbb{R}^{N'} \quad f(i) = \omega_{\text{row } i} \tag{3.23}$$

This way the continuous mapping layer becomes computationally negligible both during use and during training, as training only affects the responsible row and not the entire matrix. This last property in fact may leave weights of ω entirely untrained for unseen categorical features. The A layer situated before M prevents that situation from occurring.

3.6.1.3 The nonlinear hidden layer and the output layer

The continuous mapping layer M cascades into a classic neural network with input layer $\mathbf{i}$, hidden layer $\mathbf{h}$, and output layer $\mathbf{o}$. To avoid potential confusion, we only count the number of hidden layers. As such, the neural network in Figure 3.3 has a total of two hidden layers, one more than a standard topology.

The activation function of the second hidden layer is based on the hyperbolic tangent function [134]:

$$\phi_h(x_i) = 1.7159 \tanh\left(\frac{2}{3}x_i\right) \tag{3.24}$$

Finally, the activation function of the last layer is the softmax function that is also nonlinear and in addition ensures a normalized output:

$$\phi_o(x_i) = \frac{\mathrm{e}^{x_{(i)}}}{\sum_{j=1}^{\ell} \mathrm{e}^{x_{(j)}}} \tag{3.25}$$

3.6.1.4 MLP training

The entire network, up to and including the M layer, is trained via backpropagation [190, Ch. 7]. The approximation layer A is not trained as its transfer function is predefined. The training criterion minimizes the regularized mean squared error on the training data:

$$L = \frac{1}{n}\sum_{t=1}^{n} \left(P(y|x,\theta) - \delta_\ell(y)\right)^2 + R(\theta) \tag{3.26}$$

where θ stands in for all parameters of the neural network (the values of all weight matrices), and R is a regularization term. We used an L_2 regularizer [169] that penalizes large values of the weight matrices. The regularizer is implemented by reducing each weight change by a factor proportional to the magnitude of the weight itself.

3.6.2 Graph-Based Learner Setup

We use a dense graph approach in conjunction with the iterative approach to label propagation. Convergence is stopped when the maximum relative difference between the values computed in two consecutive steps is less than 1%.

For data size reasons, we apply label propagation in chunks. While the training set stays permanently in memory, the test data is loaded in fixed-size chunks, labeled, and discarded. This approach has yielded similar results for various chunk sizes, suggesting that chunking is a good approximation of whole-set label propagation. In fact, experiments have shown that performance tends to degrade for larger chunk sizes, suggesting that whole-set LP might be affected by “artifact” clusters that are not related to the labels. LP in chunks is also amenable to parallelization: Our system labels different chunks in parallel.

We trained the α hyperparameter by three-fold cross-validation on the training data, using a geometric progression with limits 0.1 and 10 and ratio 2. We set fixed upper limits of edges between an unlabeled node and its labeled neighbors to 15, and between an unlabeled node and its unlabeled neighbors to 5. The approach of setting different limits among different kinds of nodes is also used in related work [93].

For graph construction we tested: (a) the original discrete input representation with cosine distance; (b) the classifier output features (probability distributions) with the Jensen-Shannon distance. These combinations were determined to be the best in several initial experiments.

3.6.3 Combination optimization

The static parameters of the MLP (learning rate, regularization rate, and number of hidden units) were optimized for the LP step by 5-fold cross-validation on the training data. This process is important because overspecialization is detrimental to the combined system: an overspecialized first-pass classifier may output very confident but wrong predictions for unseen patterns, thus placing such samples at large distances from all correctly labeled samples.

Regularization during backpropagation is crucial for achieving good smoothness of the combined system. Trained without regularization, neural networks tend to produce low-entropy, highly confident classifications. As discussed in § 3.1, such an output is detrimental for the label propagation stage. Therefore we use a strong regularization coefficient to curb the tendency of the MLP to issue low-entropy outputs. A strongly regularized neural network, by contrast, will output smoother probability distributions for unseen patterns. Such outputs also result in a smoother graph, which in turn helps the LP process. Thus, we found that a network with only 12 hidden units and relatively high $R(\theta)$ in Eq. 3.26 (10% of the weight value) performed best in combination with LP (at an insignificant cost in accuracy when used as an isolated classifier).

3.6.4 Results

Table 3.2 summarizes the experimental results obtained. We first conducted an experiment to measure the smoothness of the underlying graph, $\mathcal{S}(G)$, in the two LP experiments according to the following formula:

$$\mathcal{S}(G) = \sum_{\mathtt{y}_i \neq \mathtt{y}_j, (i>n \vee j>n)} w_{ij} \tag{3.27}$$

where $\mathtt{y}_i$ is the label of sample i. (Lower values are better as they reflect less affinity between nodes of different labels.) The value of $\mathcal{S}(G)$ was in all cases significantly better on graphs constructed with our proposed technique than on graphs constructed in the standard way (see Table 3.2). The same table also shows the performance comparison between LP over the discrete representation and cosine distance ("LP"), the neural network itself ("NN"), and LP over the continuous representation ("NN+LP"), on all different subsets and for different training sizes. For scarce labeled data (5000 samples), the neural network—which uses a strictly supervised training procedure—is at a clear disadvantage. However, for a larger training set the neural network is able to perform more accurately than the LP learner that uses the discrete features directly. The third, combined technique outperforms the first two significantly. Significance was tested using a difference of proportions

significance test; the significance level is 0.01 or smaller in all cases. The differences are more pronounced for smaller training set sizes. Interestingly, the LP is able to extract information from largely erroneous (noisy) distributions learned by the neural network.

Initial labels	Model	$\mathcal{S}(G)$ avg.	Accuracy (%)					
			Set 1	Set 2	Set 3	Set 4	Set 5	Average
5000	NN	–	50.70	59.22	63.77	60.09	54.58	57.67 ± 4.55
	LP	451.54	58.37	59.91	60.88	62.01	59.47	60.13 ± 1.24
	NN+LP	409.79	58.03	63.91	66.62	65.93	57.76	$\mathbf{62.45 \pm 3.83}$
10000	NN	–	65.86	60.19	67.52	65.68	65.64	64.98 ± 2.49
	LP	381.16	58.27	60.04	60.85	61.99	62.06	60.64 ± 1.40
	NN+LP	315.53	69.36	64.73	69.50	70.26	67.71	$\mathbf{68.31 \pm 1.97}$
15000	NN	–	69.85	66.42	70.88	70.71	72.18	70.01 ± 1.94
	LP	299.10	58.51	61.00	60.94	63.53	60.98	60.99 ± 1.59
	NN+LP	235.83	70.59	69.45	69.99	71.20	73.45	$\mathbf{70.94 \pm 1.39}$

Table 3.2: Accuracy results of neural classification (NN), LP with discrete features (LP), and combined (NN+LP), over 5 random samplings of 5000, 10000, and 15000 labeled words in the WSJ lexicon acquisition task. $\mathcal{S}(G)$ is the smoothness of the graph (smaller is better).

3.7 Application: Word Sense Disambiguation

The second task is word sense disambiguation using the SENSEVAL-3 corpus [161], which enables a comparison of our method with previously published results. The goal is to disambiguate the different senses of each of 57 words given the sentences within which they occur. There are $\mathtt{t} = 7860$ samples for training and $\mathtt{u} = 3944$ samples for testing.

In line with existing work [135, 78], we use the features described in Table 3.3. However, syntactic features, which have been used in some previous studies on this dataset [164], were not included.

We used the MXPOST tagger [186] for POS annotation. The local collocations $C_{i,j}$ are concatenated words from the context of the word to disambiguate. The limits i and j are the boundaries of the collocation window relative to the focal word (which is at index zero). The focal word itself is eliminated. For example, for the sentence "Please check this out" and the focal word "check," collocation $C_{-1,2}$ is *please_this_out* and collocation $C_{-2,1}$ is ϵ*_please_this*, where ϵ is a special symbol standing in for the void context.

Related work on the same task [135] uses collocations $C_{-1,-1}$, $C_{1,1}$, $C_{-2,-2}$, $C_{2,2}$, $C_{-2,-1}$, $C_{-1,1}$, $C_{1,2}$, $C_{-3,-1}$, $C_{-2,1}$, $C_{-1,2}$, and $C_{1,3}$ as features. In addition to those, we also used $C_{-3,1}$, $C_{-3,2}$, $C_{-2,3}$, $C_{-1,3}$, for a total of 15 distinct collocations. The extra features were selected systematically by applying a simple feature selection method: a feature x is selected if the conditional entropy $H(y|x)$ is above a fixed threshold (1 bit) in the training set, and if x also occurs in the test set (note that no label information from the test data is used for this purpose).

#	Feature	Type
$F_{1..3}$	POSs of the previous 3 words	Categorical $\times$ 3
$F_{4..6}$	POSs of the next 3 words	Categorical $\times$ 3
F_7	POS of the focal word itself	Categorical
$F_{8..22}$	Local collocations $C_{-1,-1}$, $C_{1,1}$, $C_{-2,-2}$, $C_{2,2}$, $C_{-2,-1}$, $C_{-1,1}$, $C_{1,2}$, $C_{-3,-1}$, $C_{-2,1}$, $C_{-1,2}$, $C_{1,3}$, $C_{-3,1}$, $C_{-3,2}$, $C_{-2,3}$, and $C_{-1,3}$ (see text for details)	Categorical $\times$ 15
$F_{8..}$	A bag of all words in the surrounding context	Categorical $\times$ v

Table 3.3: Features used in the word sense disambiguation task.

We compare the performance of an SVM classifier, an LP learner using the same input features as the SVM, and an LP learner using the SVM outputs as input features. To analyze the influence of training set size on accuracy, we randomly sample subsets of the training data (25%, 50%, and 75%) and use the remaining training data plus the test data as unlabeled data, similarly to the procedure followed in related work [78]. The results are averaged over five different random samplings. The samplings were chosen such that there was at least one sample for each label in the training set. SENSEVAL-3 sports multi-labeled samples and samples with the "unknown" label. We eliminate all samples labeled as unknown and retain only the first label for the multi-labeled instances.

3.7.1 SVM First-Pass Classifier Setup

The use of SVM vs. MLP in this case was justified by the very small training data set. An MLP has many parameters and needs a considerable amount of data for effective training, so for this task with only on the order of 10^2 training samples per classifier, initial testing deemed an SVM more appropriate. We use the SVM$^{\text{light}}$ package [112] to build a set of binary classifiers in a one-versus-all formulation of the multi-class classification problem. The features input to each SVM consist of the discrete features described in Table 3.3 after feature selection.

We defined one SVM per target label and we trained it to discriminate that label against the union of all others, setup that is commonly used and known as one-versus-all training [69]. We evaluate the SVM approach against the test set by using the winner-takes-all strategy: the predicted label corresponds to the SVM that outputs the largest value.

3.7.2 Label Propagation Setup

Again we set up two LP systems. One uses the original feature space (after feature selection, which benefited all of the tested systems). The other uses the SVM outputs as its sole input. Both use a cosine distance measure. Note that this experiment is to some extent an exception from the others in that it does not use probability distribution as its input. Instead, it simply uses the uncalibrated outputs of the SVM, which are theoretically unbounded and practically lie around the range $[-1, 1]$. For that reason, the distance measures discussed for probability distributions are not applicable, so we applied cosine distance. A possible alternative is to fit the SVM outputs to a Gaussian and then

normalize the results [182, 145]. We chose to use the SVM outputs directly in order to explore applicability of LP on first-pass classifiers with non-probabilistic outputs.

The α hyperparameter (Eq. 3.2) is optimized through 3-fold cross-validation on the training set.

3.7.3 Combination Optimization

Unlike MLPs, SVMs do not compute a smooth output distribution. Instead, they are trained for targets -1 for one label and 1 for the other label, and base the classification decision on the sign of the output values. In order to smooth output values with a view towards graph construction we applied the following techniques:

1. *Combining SVM predictions and perfect feature vectors:* After training, the SVM actually outputs wrong label predictions for a small number ($\approx 5\%$) of training samples. These outputs could simply be replaced with the perfect SVM predictions (1 for the true class, -1 elsewhere) since the labels are known. However, the second-pass learner might actually benefit from the information contained in the mis-classifications. We therefore linearly combine the SVM predictions with the "perfect" feature vectors $\mathbf{v}$ that contain 1 at the correct label position and -1 elsewhere:

$$s'_i = \gamma s_i + (1 - \gamma)\mathbf{v}_i \tag{3.28}$$

   where s_i, s'_i are the i^{th} input and output feature vectors and γ a parameter fixed at 0.5.

2. *Biasing uninformative distributions:* For some training samples, although the predicted class label was correct, the outputs of the SVM were relatively close to one another, i.e. the decision was borderline. We decided to bias these SVM outputs in the right direction by using the same formula as in Eq. 3.28.

3. *Weighting by class priors:* For each training sample, a corresponding sample with the perfect output features was added, thus doubling the total number of labeled nodes in the graph. These synthesized nodes are akin to "dongle" nodes as used by Zhu and Goldberg [238, 93]. The role of the artificial nodes is to serve as authorities during the LP process and to emphasize class priors.

3.7.4 Results

As before, we measured the smoothness of the graphs in the two label propagation setups and found that in all cases the smoothness of the graph produced with our method was better when compared to the graphs produced using the standard approach, as shown in Table 3.5, which also shows accuracy results for the SVM ("SVM" label), LP over the standard graph ("LP"), and label propagation over SVM outputs ("SVM+LP"). The latter system consistently performs best in all cases, although the most marked gains occur in the upper range of labeled samples percentage. The gain of the best data-driven LP over the knowledge-based LP is significant in the 100% and 75% cases.

#	System	Acc. (%)
1	htsa3 [96]	72.9
2	IRST-kernels [211]	72.6
3	nusels [136]	72.4
4	SENSEVAL-3 contest baseline	55.2
5	Niu et al. [78] LP/Jensen-Shannon	70.3
6	Niu et al. LP/cosine distance	68.4
7	Niu et al. SVM	69.7

Table 3.4: Accuracy results of other published systems on SENSEVAL-3. Systems 1, 2, and 3 use syntactic features; 5, 6, and 7 are directly comparably to our system.

Initial labels	Model	$\mathcal{S}(G)$ avg.	Accuracy (%)					
			Set 1	Set 2	Set 3	Set 4	Set 5	Average
25%	SVM	–	62.94	62.53	62.69	63.52	62.99	62.93 ± 0.34
	LP	44.71	63.27	61.84	63.26	62.96	63.30	62.93 ± 0.56
	SVM+LP	39.67	63.39	63.20	63.95	63.68	63.91	$\mathbf{63.63 \pm 0.29}$
50%	SVM	–	67.90	66.75	67.57	67.44	66.79	67.29 ± 0.45
	LP	33.17	67.84	66.57	67.35	66.52	66.35	66.93 ± 0.57
	SVM+LP	24.19	67.95	67.54	67.93	68.21	68.11	$\mathbf{67.95 \pm 0.23}$
75%	SVM	–	69.54	70.19	68.75	69.80	68.73	69.40 ± 0.58
	LP	29.93	68.87	68.65	68.58	68.42	67.19	68.34 ± 0.59
	SVM+LP	16.19	69.98	70.05	69.69	70.38	68.94	$\mathbf{69.81 \pm 0.49}$
100%	SVM	–						70.74
	LP	21.72						69.69
	SVM+LP	13.17						**71.72**

Table 3.5: Accuracy results of support vector machine (SVM), label propagation over discrete features (LP), and label propagation over SVM outputs (SVM+LP), for the word sense disambiguation task. Each learner was trained with 25%, 50%, 75% (5 random samplings each), and 100% of the training set. The improvements of SVM+LP are significant over LP in the 75% and 100% cases. $\mathcal{S}(G)$ is the graph smoothness.

For comparison purposes, Table 3.4 shows results of other published systems against the SENSEVAL-3 corpus. The "htsa3", "IRST-kernels", and "nusels" systems were the winners of the SENSEVAL-3 contest and used extra input features (syntactic relations). The Niu et al. work [78] is the most comparable to ours. We attribute the slightly higher performance of our SVM due to our feature selection process. The LP/cosine system is a system similar to our LP system using the discrete features, and the LP/Jensen-Shannon system is also similar but uses a distance measure derived from Jensen-Shannon divergence.

3.8 Application: Acoustic Classification

Perhaps the most complex systems used in HLT today are dedicated to automatic speech processing. Here we will focus on acoustic modeling, one relatively well-delimited and specialized aspect of speech processing.

From a modeling standpoint, speech recognition can be described by the equation:

$$\hat{W} = \arg\max_{W} P(W|X) \tag{3.29}$$

where $X = x_1 x_2 \ldots$ is the acoustic observation sequence, and $\hat{W} = w_1 w_2 \ldots$ is the corresponding estimated word sequence. The large task of estimating W from X can be simplified with the help of phone recognition, where the sequence of words W is replaced with a sequence of phones out of a possible phone vocabulary. The task could be simplified further by removing temporal information. In that case, a sequence of phonetic observations predicts a single phone, task known as *phone classification*. We will focus our next experiment on a phone classification task.

One important challenge for phone classification and speech recognition in general is finding a good representation of the speech signal X, specifically, extracting indicative features from the audio signal.

Today, frequency domain representations are the dominant approach to feature extraction for speech. A widely used feature representation is known as the Mel-Frequency Cepstrum Coefficients (MFCC) [63]. Bogert et al. introduced the notion of cepstrum (an anagram of "spectrum") in 1963 [31]. The cepstrum of a signal is the Fourier transform of the power spectrum of the signal. The signal is applied the Fourier transform once, then the power is obtained by squaring the transform, then the logarithm is applied to express power in decibels (dB), and finally the cepstrum is obtained by applying the Fourier transform again to the power in dB. The double application of the Fourier transform reflects the cepstrum's ability to capture relatively slow variations in the frequency spectrum of the input signal. The double transform can be analyzed like a regular signal, and notions such as quefrency and liftering have been defined by furthering the anagram metaphor [113]. It has been shown experimentally [63] that such slow variations of the power spectrum are indicative features of the speech signal.

The MFCC method is specialized for speech by being perceptually-motivated. The human ear has a specific and nonlinear frequency response, and the humans' excellent capability of understanding speech motivates imitation of at least the early stages of the hearing system, which are easily measured and relatively well understood. MFCC therefore approximates the human ear's frequency response by warping the power-frequency spectrum obtained after applying the Fourier transform into a different spectrum by using an empirical function known as the Mel frequency. Furthermore, the warped power spectrum is filtered through a series of band-pass filters, each having a triangular-shaped response [105, Ch. 6]. The purpose of the filtering is to allow for down-sampling of the signal without aberrations caused by . A notable difference from the classic cepstral transform is that the second transform applied is Discrete Cosine instead of Fourier. It has been experimentally showed that the Discrete Cosine instead of Fourier yields better speech features than the Fourier transform [36]. The importance of the function parameters decreases with their order. Application commonly use the first 13 coefficients (the continuous components at each quefrency), to which three more coefficient sets may be added, each containing 13 coefficients: (a) the 1–2 Hz modula-

tion energy; (b) the 3–15 Hz modulation energy; and (c) the 20-43 Hz modulation energy, for a total of 52 possible coefficients. Our experiments use the first 26 coefficients.

Contemporary acoustic modeling approaches are typically using a tried-and-true technique: after sound acquisition and extraction of MFCC features, a hidden Markov model (HMM) with Gaussian mixture (GM) probabilistic models is being trained. Today's state-of-the-art systems further improve accuracy and robustness by using discriminative training and adaptation to test data using techniques such as MLLR [86] or MAP [89].

Several alternative or complementary approaches have been explored in the past, including different ways of modeling output distributions, such as Support Vector Machines (SVMs) [87] and neural networks [33], as well as novel training techniques, such as large-margin training [198]. However, adoption of new methods by the mainstream ASR community has been slow—with some exceptions [233, 210]—mainly because the standard methodology is well-tested, efficient, and easy to use, and also because new models or learning procedures often do not scale well to large datasets. Exploration is difficult mainly because of the data sizes involved: training even a highly optimized speech recognition system takes hours or days. On such large data sets, sophisticated machine learning methods are hardly applicable, even if they are theoretically superior and achieve good results on artificial or small tasks. A field researcher or developer would be therefore inclined towards spending time on incremental improvements on the existing techniques instead of trying radically new approaches that are liable to have an extremely long experimental cycle. It could be argued that due to sheer data size, the ASR community is forced to improve on relatively well-understood local optima instead of exploring in search of qualitatively better approaches.

Continued progress in ASR, however, does require exploring novel approaches, including new machine learning techniques, as well as adapting these to large data sets and the computational constraints that present-day ASR systems are subject to. In the following we investigate graph-based learning as a way to improve over standard acoustic models.

Applying graph-based learning to speech is a potentially advantageous endeavor. As discussed, graph-based classification enforces global consistency across training and test samples, so it is inherently adaptive. In contrast, related traditional systems (such as nearest-neighbor) only rely on similarity between the test and training samples. Graph-based learning is typically used in a semi-supervised, transductive setting where a relatively small amount of labeled data is used in conjunction with a large amount of unlabeled data. However, as we will show below, it can also be used as a post-processing step applied to a standard supervised classifier trained on a large amount of labeled data and tested on a small amount of unseen data, which is the typical scenario in speech processing. In this case, graph-based learning provides a form of adaptation to the test data by constraining the decisions made by the first-pass classifier to accommodate the underlying structure of the test data.

On the other hand, applying graph-based learning to acoustic classification raises unique challenges:

- *Similarity measure:* As discussed in Chapter 2, choosing an appropriate similarity measure is key to graph-based learning. It is unclear what similarity measure would be optimal in acoustic feature spaces.

- *Adaptation to Sample Size Discrepancy:* Originally, graph-based learning was formulated for semi-supervised scenarios, where a large amount of unlabeled but a small amount of labeled

data are present. In many speech processing applications, we find the opposite situation. In these cases, graph-based learning can still be of benefit due to the global consistency assumption it enforces, thus effectively implementing adaptation in a different sense than commonly used. However, this requires changes to the basic algorithm.

- *Scalability:* Acoustic data is typically available in large quantities. Constructing a full similarity graph would be feasible only for very small speech corpora. We will discuss our approach to scalability of phone classification in Chapter 5.

We describe in the following subsections how our system addresses these challenges. The setup consists of the two-pass system described in § 3.3 in conjunction with Jensen-Shannon divergence (§ 2.3) as a distance measure. Results on an 8-class vowel classifier are presented with the goal of demonstrating the effect on speaker adaptation. Our approach improves significantly over state-of-the-art adaptation algorithms.

3.8.1 Adaptation to Sample Size Discrepancy

Adaptation is an important challenge in speaker-independent ASR systems. Label propagation is inherently adaptive because it uses the self-similarity of the test data in addition to the similarity of the test data with the training data. To properly exploit the adaptive nature of label propagation, we operate a simple but essential change to the matrix $\mathtt{W}$.

First, let us consider the situation $\mathtt{t} \gg \mathtt{u}$. This is the case when a speech classification or recognition system is trained against many hours of data and then presented a brief utterance, such as a phrase or sentence. The samples of the test utterance will bear similarity edges with the training samples and also similarity edges with other test samples. Given that there are much more many labeled samples than unlabeled ones, and also that similarity is additive (per Theorem 5.4.2), it follows that the accumulated similarity with labeled data will be much stronger than the similarity with unlabeled data, even when similarity with each individual training sample is much smaller than similarity with other test samples.

A graph-based learner in which the edge weights linking unlabeled to labeled samples are much stronger than edges linking unlabeled samples with one another will degenerate into an unsophisticated nearest-neighbor classifier: random walks will be always or almost always absorbed directly by labeled vertices, therefore test samples will be labeled in proportion to the accumulated connection strength for each label.

If only the k nearest neighbors are used in building the graph, the effect is less pronounced but still present. Due to the large quantity of training data, the likelihood of finding similar training data is higher, so the k top slots may be saturated with similar entries, which lead to strong weights after summation. In contrast, even though one or a few unlabeled neighbors may be very similar, the dearth of unlabeled samples means that unlabeled-unlabeled connections are still at a large disadvantage. Even a relatively low threshold such as $k = 10$ means a handicap of up to one order of magnitude for the unlabeled-unlabeled connections.

To benefit of adaptation, we want to manipulate the density of the graph in the region of the test utterance. We achieve this by adjusting $\mathtt{w}_{ij}$ linking unlabeled samples with one another by:

$$\mathtt{w}_{ij} \leftarrow \frac{\mathtt{t}}{\mathtt{u}} \mathtt{w}_{ij} \quad \forall\, i > \mathtt{t}, j > \mathtt{t} \tag{3.30}$$

This artificially simulates that there are as many test as train samples, greatly enhancing the adaptive properties of the algorithm. Although simple, this adjustment is extremely effective; without it, the classification degenerates in nearest-neighbor (i.e., the label propagation algorithm converges in exactly one step). We confirmed experimentally that the unadjusted graph never improves upon the first-pass classifier.

3.8.2 Interpolation with Prior Distributions

For the training set we have access to the true labels and consequently to the sample prior probability distributions:

$$P_p^{\mathtt{y}_i} = \langle\!\langle 0_1, \ldots, 0_{\mathtt{y}_i-1}, 1_{\mathtt{y}_i}, 0_{\mathtt{y}_i+1}, \ldots, 0_\ell \rangle\!\rangle = \delta_{\mathtt{t}}(\mathtt{y}_i) \tag{3.31}$$

($\delta_\ell(n)$ denotes a Kronecker vector of length ℓ with 1 in the n^{th} position and 0 elsewhere.) These prior distributions represent the ground truth, so they are highly informative for classification. Using them exclusively, however, would lose smoothness information, so they should best be used in interpolation with the soft predictions resulting from the first-pass classifier running against its own training data. We chose an equal-weight interpolation $\frac{P + P_p}{2}$ throughout our experiments.

Interpolation with priors is interesting from two perspectives. First, interpolation achieves a similar effect to Zhu's dongle vertices [238, § 4.6]. Zhu suggested and successfully used additional labeled vertices (that he called dongle vertices) that encode additional knowledge about data, such as the predictions of an external classifier. For example, each unlabeled sample may be linked to a dongle node that bears a label (soft or hard) as predicted for that sample by another classifier. The strength of the connection is commensurate to the desired influence of that additional classifier over the label propagation process. On the manifold approximated by the graph, the presence of dongle vertices creates additional labeled "holes" that attract random walks originating in unlabeled vertices and as such bias the labeling process. In effect, dongle vertices change labeled point density on the manifold in the vicinity of unlabeled points. Interpolation achieves a similar effect by only changing feature vectors and consequently connection weights, without adding any new vertices. In fact, after graph reduction (§ 5.4) is taken into account, the manifold reveals itself as a space with exactly ℓ labeled attractors, one for each label. Interpolation of weights with the Kronecker vectors is equivalent to adding dongle vertices for the corresponding corners of the space that encode maximally confident label decisions. This has the effect of compensating the systematic errors and the noise sensitivity of the first-pass classifier. An important aspect is that interpolation does not impact graph size and scalability; in contrast, adding dongle vertices increase the number of vertices and may increase the size of the associated matrices $\mathtt{P}_{\mathtt{UU}}$ and/or $\mathtt{P}_{\mathtt{UL}}$.

The second interesting aspect of interpolation is that it directly uses the *features-labels* duality, a property of the two-pass classifier. In the graph-based system, features and labels have the same semantics, whereas in a traditional classifier, features and labels belong to distinct spaces. The duality allows us to mix them by injecting the label-derived Kronecker vectors into the features of the labeled samples by simply averaging the two.

3.8.3 Data

We performed experiments on an 8-vowel classification task collected for the Vocal Joystick (VJ) project [118], whose goal is to develop voice-controlled assistive devices for individuals with motor

impairments. In the typical setup, a VJ user can exercise analog, continuous control over mouse cursor movements by using vowel quality, pitch, or loudness. One of the components of the VJ system is a speaker-independent vowel classifier whose output is used to control, for example, the direction in which a mouse cursor moves. In this and similar scenarios, phonetic classification that is robust against speaker variation is of utmost importance in order to avoid rejection of the system by the user due to inaccurate recognition of control commands.

For training this classifier, a corpus was collected consisting of 11 hours of recorded data of which we selected a subset. The sizes of the train, development, and test data are shown in Table 3.6.

Set	Speakers	Samples	Non-silent audio
Training	21	$420 \cdot 10^3$	1.16h
Development	4	$200 \cdot 10^3$	0.56h
Test	10	$80 \cdot 10^3$	0.22h

Table 3.6: Training, development, and testing data used in the Vocal Joystick experiments.

This scenario is a good test bed for our proposed approach since an already tuned, high-performing baseline system with standard adaptation methods exists for this data set. In addition, the focus on phonetic classification allows us to focus on the acoustic models while ignoring e.g. language model and search effects that would characterize large-vocabulary systems. At the same time, this corpus is vastly more realistic than the toy tasks used in machine learning since it contains hundreds of thousands of samples.

3.8.4 Experiments and Results

We tested our phone classification system by directly using the outputs of the best classifier on the VJ corpus to date, created by Li [144]. Li's classifier is a multi-layer perceptron (MLP) enhanced with a regularized adaptation algorithm. The adaptation algorithm uses a regularizer that prevents the regularized model diverging too much from the unadapted system, thus avoiding overtraining on adaptation data. We used the same MLP (50 hidden units and a window size of 7 samples) and the same adaptation algorithm as Li.

We apply our system to both the non-adapted MLP outputs and the adapted outputs. In each case, a graph (of reduced size using the result of Proposition 1) was built for each test utterance, after which iterative label propagation was applied to the graph. As an additional baseline we use GMMs (a) without adaptation and (b) with MLLR adaptation. The adaptation experiments used 5-fold cross-validation, each time using a held-out part of the test data for computing adaptation parameters. The results are shown in Table 3.7. Boldface numbers are significantly better than the comparable baselines.

The similarity of choice was Jensen-Shannon divergence; to confirm that it is a good-quality distance, we compared it with development set performance for two commonly-used distance measures: Cosine distance and Euclidean distance. They both engendered higher error rates ($22.62 \pm 11.23\%$ for Cosine and $22.48 \pm 11.00\%$ for Euclidean).

Model	Error Rate (%)	
	Dev	Test
GMM, no adaptation	n/a	39.62
MLP, no adaptation	24.81±10.69	31.91±9.39
MLP+GBL, no adaptation	**21.91±10.52**	**28.75±12.31**
GMM+adaptation	n/a	20.05±3.76
MLP+adaptation	n/a	12.18±3.51
MLP+adaptation+GBL	n/a	**8.32±3.21**

Table 3.7: Error rates (means and standard deviations over all speakers) using a Gaussian Mixture Model (GMM), multi-layer perceptron (MLP), and MLP followed by a graph-based learner (GBL), with and without adaptation. The highlighted entries represent the best error rate by a significant margin ($p < 0.001$).

3.9 Discussion of the Two-Pass Classifier Approach

In this chapter we investigated a two-step procedure for graph construction that uses a supervised classifier in conjunction with a graph-based learner. The advantages of the two-pass classifier system are:

- *Uniform range and type of features*: The output from a first-pass classifier can produce well-defined features, in the form of posterior probability distributions. This eliminates the problem of input features having different ranges and types (e.g. binary vs. multivalued, continuous vs. categorical attributes) which are often used in combination.

- *Feature postprocessing*: The transformation of features into a different space also opens up possibilities for postprocessing (e.g. probability distribution warping) depending on the requirements of the second-pass learner. In addition, specialized distance measures defined on probability spaces (§ 3.4) can be used, which avoids violating assumptions made by metrics such as Euclidean and cosine distance.

- *Optimizing class separation:* The learned representation of labeled training samples might reveal better clusters in the data than the original representation: a discriminatively-trained first pass classifier will attempt to maximize the separation of samples belonging to different classes. Moreover, the first-pass classifier may learn a feature transformation that suppresses noise in the original input space.

Difficulties with the proposed approach might arise when the first-pass classifier yields confident but wrong predictions, especially for outlier samples in the original space. For this reason, the first-pass classifier and the graph-based learner should not simply be concatenated without modification, but the first classifier should be optimized with respect to the requirements of the second.

Experiments suggest that the resulting system combines the strengths of both classifiers. The first-pass classifier offers the graph-based learner a uniform and low-dimensional feature set to work with. That feature format is better suited for an optimally-functioning distance measure. Measurements put the proposed two-pass approach to classification in contrast with a more traditional approach of using stock distance measures on top of the raw features. Results show that the approach using the outputs of the first-pass classifier as features for the graph-based classifier is superior to the conventional approach.

Next chapter will mark a departure from the experimental setup discussed above. Instead of using fixed-length real-valued vectors as features and discrete label values, we will focus on defining a theoretical and practical framework for applying graph-based learning to *structured* inputs and outputs.

Chapter 4

GRAPH-BASED LEARNING FOR STRUCTURED INPUTS AND OUTPUTS

The theoretical study and practical applications introduced in Chapter 3 used a Gaussian similarity kernel to compute a similarity graph. The similarity kernel worked on top of a distance measure, which in turn was defined over fixed-length vectors containing either problem-specific features or probability distributions obtained from a first-pass classifier. The inputs to the overall learning system were always unstructured—fixed-length feature vectors containing real numbers. Certain features had Boolean or categorical values, in which case we took special measures to transform them into real-numbered values, such as the one-hot approach (§ 3.6.1.2). The predicted labels were categorical as well (e.g., POS tag or word sense).

It is worth noting that in the applications presented above, the components of the input vector did sometimes exhibit interdependence, which confers structure to the feature space $\mathcal{X}$. For example, the lexicon learning experiment (§ 3.6) uses features (refer to Table 3.3) that are obeying certain constraints, the most obvious being that feature F_8 (Boolean feature that is true if the word consists only of capital letters) logically implies F_7 (Boolean feature that is true if the word contains capital letters). There are, without a doubt, more subtle interdependencies and intrinsic structure in the features in Table 3.3, for example there is a strong correlation between F_2 and F_1, the latter being a suffix of the former. Part of the value of the two-pass classifier discussed was that it could learn a similarity measure and ultimately a classification function without requiring heavy feature selection or preprocessing. Ultimately, however, the learner was *unstructured* because it ignored structural information of the input or output space. Exploiting such information could be advantageous because structural constraints reduce the size of the search space, allowing a faster and more focused learning. Also, many learning problems do not even fit the classic mold of finding a function that maps real-valued vectors to categorical labels. The field of learning with structured inputs and outputs has received increasing attention in recent years, and is the subject of this chapter within the context of graph-based semi-supervised learning. Our contribution in this chapter is to extend graph-based learning to learning tasks with structured inputs and outputs, and to apply the resulting theoretical framework to a machine translation task.

4.1 Structured Inputs and Outputs

Traditionally, the input set $\mathcal{X}$ of a learning problem is modeled as a vector space of real-valued or categorical features, and the output set $\mathcal{Y}$ is modeled as a discrete, finite set of categorical labels. However, in many problems either or both of $\mathcal{X}$ and $\mathcal{Y}$ may be structured spaces that may or may not be finite. The structure could concern not only $\mathcal{X}$ and $\mathcal{Y}$, but also a relationship between them. Examples include:

- *Spatial structure:* In many image processing applications—such as image segmentation—the input is an entire image in raster format, and the "labels" are sets of regions of pixels denoting,

for example, objects of interest within the image.

- *Sequential/temporal structure:* In a natural language tagging application, the input consists of a sequence of words and the output is a sequence of tags, one for each word. The ordering of elements in both input and output is important. Defining sequencing on input vs. output may be very different, as in e.g. an optical character recognition application.

- *Hierarchical structure:* Natural language parsers produce syntactic trees as their output.

- *Combinatorial structure:* Machine translation applications often use *alignments*—bipartite graphs that show the correspondence of each word or phrase in the source language to a word or phrase in the target language.

The classification above is not exhaustive because arbitrary kinds of structural constraints may be added to inputs, outputs, or their combination.

The machine learning approaches that we have discussed until now build an estimate of the conditional probability $p(\mathtt{y}|\mathtt{x})$, usually in form of a probability distribution over the discrete labels $\{1, \ldots, \ell\}$. A natural extension of this approach to structured data is to analytically define $p(\mathtt{y}|\mathtt{x})$ as a parameterized function that obeys by definition the structural constraints of $\mathcal{X}$ and $\mathcal{Y}$. Then, parameter estimation by using e.g. gradient-based or maximum-margin techniques accomplishes the learning task. This approach has been successfully used in maximum-margin Markov models [215], kernel conditional random fields [131], hidden Markov support vector machines [6], and support vector machines for structured output spaces [218].

Another possibility is to forego analytic definition for $p(\mathtt{y}|\mathtt{x})$ and instead focus on regressing a real-valued *scoring* function s. Such a scoring function accepts a pair of input and output data and computes a real-valued score:

$$s : \mathcal{X} \times \mathcal{Y} \to \mathbb{R} \cup \{-\infty\} \tag{4.1}$$

The scoring function encapsulates all structural constraints and yields larger numbers for better matched pairs of inputs and outputs; the nature of the scoring function is that it always fulfills whatever structural constraints must be satisfied by $\mathtt{x}$ and $\mathtt{y}$. Training data pairs are considered highly feasible so they are assigned high values of s. Conversely, infeasible, unlikely, or unwanted pairs are assigned low values of s. For completeness, if a pair $\langle\!\langle \mathtt{x}, \mathtt{y} \rangle\!\rangle$ does not satisfy the structural constraints, $s(\mathtt{x}, \mathtt{y}) \triangleq -\infty$. Given this setup, estimated structured labels are obtained by solving:

$$\hat{\mathtt{y}} = \arg\max_{\mathtt{y} \in \mathcal{Y}} s(\mathtt{x}, \mathtt{y}) \tag{4.2}$$

Often it is possible that not all pairs in $\mathcal{X} \times \mathcal{Y}$ are feasible. Some applications denote

$$\mathcal{Y}(\mathtt{x}) \triangleq \{\mathtt{y} \in \mathcal{Y} \mid s(\mathtt{x}, \mathtt{y}) \neq -\infty\} \tag{4.3}$$

which eliminates a priori unfeasible combinations from the search space, in which case the learning problem can be reformulated as

$$\hat{\mathtt{y}} = \arg\max_{\mathtt{y} \in \mathcal{Y}(\mathtt{x})} s(\mathtt{x}, \mathtt{y}) \tag{4.4}$$

The definition of s and the method of estimating the $\arg\max$ function are application-specific. The function s is unable to emit estimated labels $\hat{\mathtt{y}}$ directly; instead, it learns an estimate of how good a given feature/label pair is. Therefore, using a scoring method for structured learning requires the existence of a *hypothesis generator function* χ.

$$\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y}) \tag{4.5}$$

where $\mathcal{F}(\mathcal{Y})$ is the finite power set of the (potentially infinite) set $\mathcal{Y}$:

$$\mathcal{F}(\mathcal{Y}) = \{A \in \mathcal{P}(\mathcal{Y}) \mid \operatorname{card}(A) < \infty\} \tag{4.6}$$

The disadvantage of a score-based formulation of structured learning is that the method is not complete in that it must work in tandem with a hypothesis generator, which poses its own learning problems. The advantage of the approach is that it allows using unstructured real-valued function regression algorithms with structured data. Such a learning problem is often simple and may scale well to large problems. In contrast, an approach that maps $\mathcal{X}$ to $\mathcal{Y}$ directly is often complex and difficult to scale.

That the codomain of χ consists of finite sets is an important detail from both a theoretical and a practical perspective. Theoretically, a finite codomain of χ makes it possible to define finite similarity graphs and therefore apply graph-based learning. Practically, reducing the search space for $\mathtt{y}$ increases the speed of search considerably regardless of the method used. The hypothesis generator χ is usually a generative learning system that is fast and has good recall, but lacks in precision (has false positives).

4.2 Graph-Based Semi-Supervised Formulation

As we have shown in Chapter 2, label propagation is capable of learning the harmonic function over a graph starting from a few vertices where the value of the function is constrained (the training, labeled vertices). Until now the learned function modeled probability values exclusively. Assembling several probability values in normalized vectors modeled probability distributions over sets of mutually exclusive labels. The scoring-based approach to structured learning provides an opportunity to apply graph-based methods to structured learning problems by regressing the scoring function s directly instead of computing probabilities. To build a graph, we need to define a similarity function between input-output pairs:

$$\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to \mathbb{R}_+ \tag{4.7}$$

Alternatively, we could define a distance function with the same domain and codomain, and then apply the Gaussian kernel to it for obtaining similarities, as we did in Chapter 3. Choosing between similarity and distance depends on the nature of $\mathcal{X}$ and $\mathcal{Y}$; for the applications we discuss below, the most natural approach is to define a similarity directly.

Given σ, a similarity graph containing the training data and the test hypotheses for a given sample can be constructed. Each vertex represents either a pair of input and output values $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$ obtained from the training set, or a test hypothesis $\langle\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\rangle$. Instead of the continuous probability distributions associated with labels, this time there will be only one continuous real-valued

"label" associated with each vertex, the scoring function s. The scores will be learned by the application of a graph-based semi-supervised learning method such as label propagation.

To understand how label propagation works for regressing a function, consider again the cost function, a.k.a. smoothness (Eq. 2.23), that the label propagation algorithm minimizes:

$$\mathcal{S} = \sum_{\substack{i,j\in\{1,\ldots,\mathtt{t}+\mathtt{u}\} \\ i>\mathtt{t} \vee j>\mathtt{t} \\ k\in\{1,\ldots,\ell\}}} \mathtt{w}_{ij} \left(\mathtt{f}_{ik} - \mathtt{f}_{jk}\right)^2 \tag{4.8}$$

under the constraint (recall that $\delta_N(n)$ is a Kronecker delta vector of length N valued at 1 in position n and 0 elsewhere):

$$\mathtt{f}_{\text{row } i} = \delta_\ell(\mathtt{y}_i) \quad \forall i \in \{1,\ldots,\mathtt{t}\} \tag{4.9}$$

In our case there is only one label to compute (the score itself) so $\ell = 1$, the weights $\mathtt{w}_{ij}$ are values of the similarity function $\sigma(\langle\langle \mathtt{x}_i, \mathtt{y}_i\rangle\rangle, \langle\langle \mathtt{x}_j, \mathtt{y}_j\rangle\rangle)$, the $\delta_\ell(\mathtt{y}_i)$ vectors become the training scores $s(\mathtt{x}_i, \mathtt{y}_i)\ \forall i \in \{1,\ldots,\mathtt{t}\}$, and the $\mathtt{f}$ matrix (in our case degenerating to a column vector) contains values of the s function, resulting after substitution in:

$$\mathcal{S} = \sum_{\substack{i,j\in\{1,\ldots,\mathtt{t}+\mathtt{u}\} \\ i>\mathtt{t} \vee j>\mathtt{t}}} \sigma(\langle\langle \mathtt{x}_i, \mathtt{y}_i\rangle\rangle, \langle\langle \mathtt{x}_j, \mathtt{y}_j\rangle\rangle) \left(s(\mathtt{x}_i, \mathtt{y}_i) - s(\mathtt{x}_j, \mathtt{y}_j)\right)^2 \tag{4.10}$$

The constraint is now implicit in the immutability of train scores; the constrained intermediate matrix $\mathtt{f}$ has disappeared entirely.

Similar to the probability case, $\mathcal{S}$ is a proper loss to minimize because it penalizes inconsistent score assignments—those that score highly similar regions with abruptly-varying score values. Score values diffuse from labeled vertices and follow the high- and low-density regions on the manifold built by σ. We can now formalize structured graph-based learning as follows.

Definition 4.2.1 (Graph-Based Formulation of Structured Learning for Regression)**.** Consider a structured learning problem defined by features $\mathtt{X} = \langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}}\rangle\rangle \subset \mathcal{X}^{\mathtt{t}+\mathtt{u}}$, training labels $\mathtt{Y} = \langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{t}}\rangle\rangle \subset \mathcal{Y}^{\mathtt{t}}$, corresponding training scores $\langle\langle s_1, \ldots, s_{\mathtt{t}}\rangle\rangle \in \mathbb{R}^{\mathtt{t}}$, similarity function $\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [0, 1]$, and hypothesis generator function $\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y})$. We define the similarity graph for the structured learning problem as an undirected weighted graph with real-valued vertex labels, constructed as follows:

- add one vertex v_i for each training pair sample $\langle\langle \mathtt{x}_i, \mathtt{y}_i\rangle\rangle\ \forall i \in \{1,\ldots,\mathtt{t}\}$, labeled with the score s_i (training pair samples have predefined scores);

- add one vertex v_{ij} (with unknown score, initially set to 0) for each pair consisting of a test sample $\mathtt{x}_i$ and a hypothesis $(\chi(\mathtt{x}_i))_j$, where $i \in \{\mathtt{t}+1,\ldots,\mathtt{t}+\mathtt{u}\}$ and $j \in \{1,\ldots,\text{card}(\chi(\mathtt{x}_i))\}$;

- for each test vertex v_{ij} and each training vertex v_k, define one edge with the weight

$$\mathtt{w}_{ijk} = \sigma(\langle\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j\rangle\rangle, \langle\langle \mathtt{x}_k, \mathtt{y}_k\rangle\rangle) \tag{4.11}$$

- for each pair of test vertices v_{ij} and v_{kl}, define an edge linking them with weight

$$\mathtt{w}_{ijkl} = \sigma(\, \langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle \,,\, \langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle \,) \tag{4.12}$$

The structural constraints of $\mathcal{X} \times \mathcal{Y}$ have not disappeared—they are now folded into the definition of σ, which bridges the structure of the input space with the unstructured regression framework. Devising good definitions of σ is the concern of the following sections.

4.2.1 Learning With Only Positive Examples

The similarity graph for structured learning as per Definition 4.2.1 needs the training scores $\langle\!\langle s_1, \ldots, s_{\mathtt{t}} \rangle\!\rangle \in \mathbb{R}^{\mathtt{t}}$. Certain problems naturally present the learning system with such scores. For example, in a sentiment categorization application [178] such as a movie review system, training data may consist of a set of sentences accompanied by an integer-valued rating from 0 (very unfavorable) to 3 (very favorable). Test data consists of texts without an explicit rating. Such a setup allows using graph-based learning to regress a real-valued scoring function that is a continuous extension of the integral training scores. After regression, test scores can be kept as such or discretized, by rounding, back to the same integer values as in training. This application has been demonstrated by Goldberg and Zhu [93].

In other learning problems, the train set contains examples and counter-examples, i.e. "good" training pairs $\langle\!\langle \mathtt{x}, \mathtt{y} \rangle\!\rangle_+$ and "bad" training pairs $\langle\!\langle \mathtt{x}, \mathtt{y} \rangle\!\rangle_-$. In such situations, a common approach is to assign each positive training sample a constant high score s_+, and each negative training sample a constant low score s_-. Then regression learns a real-valued function with range $[s_-, s_+]$. A given test sample will be "pulled" towards the positive or negative vertices as dictated by the graph structure. The actual constants s_- and s_+ dictate the highest and lowest score received by any test sample—in label propagation, all learned scores will fall in between these limits by the maximum principle of harmonic functions [1]. Aside from the obvious requirement $s_- < s_+$, there are no other restrictions with regard to choosing these values; we are only interested in their ordering. Some applications define limits such as -1 and 1 or 0 and 1. In keeping with our previous application when the computed scores had probability semantics, we choose $s_- = 0$ and $s_+ = 1$ throughout this chapter.

Many structured learning problems, however, only define a training set containing only positive examples, that is, correct pairs $\langle\!\langle \mathtt{x}_i, \mathtt{y}_i \rangle\!\rangle \ \forall i \in \{1, \ldots, \mathtt{t}\}$. Moreover, all training pairs are *equally* realizable, desirable, or "good" (there is no confidence information associated with the training data). It would appear that only a little change in setup is needed: assign the high score ($s_+ = 1$) to all training samples and leave no sample with score $s_- = 0$. This naïve setup is, however, ill-advised: In the absence of negative samples with low scores, label propagation will promptly learn the scoring function that minimizes $\mathcal{S}$ down to zero—the constant function valued at 1 at all points. The traditional setup of label propagation that we described in § 2.3.2 did not have this problem because the system predicted probability distributions over multiple and mutually exclusive labels; a training sample carrying one label was automatically a negative sample for all other labels.

Automatic generation of negative samples is an option for certain problems, but one that should be approached carefully because not all negative samples are useful, particularly in high-dimensional spaces. Consider a structured problem where $\mathcal{X} \times \mathcal{Y}$ is such a large space. Then, by necessity, the actual train data and test hypotheses points will only fill a small portion of that space.

(If the learning problem is formulated properly for graph-based learning, the training data and the correct hypotheses will form a lower-dimensional manifold in that space.) Generating random hypotheses would simply place random points in that sparsely populated space, a strategy that falls prey to the curse of dimensionality: those random points will be equally far from any correct hypotheses and incorrect ones, and as such will be uninformative. A "good negative" example must be dissimilar with all positive training pairs (which is easy to accomplish) but also *similar* with the incorrect or inferior pairs predicted by the hypothesis generator χ. Such a generator would need to follow the characteristic of the hypothesis generator and its proneness to making systematic errors, a requirement that is difficult to fulfill.

We will use a different approach that avoids the necessity of generating negative samples. The idea is to *infer* negative samples by exploiting information provided by the similarity function σ. For each training sample $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$, $i \in \{1, \ldots, \mathtt{t}\}$, we construct not only one vertex v_{i+} as prescribed by standard graph construction ("the positive vertex"), but also one extra "negative" vertex v_{i-}. The score assigned to v_{i+} is always $s_+ = 1$, whereas the score assigned to v_{i-} is always $s_- = 0$. "Positive" and "negative" for vertices refers to them representing positive (realizable) vs. negative (unrealizable) training samples, not a mathematical sign. In fact, given our choice of scores $s_+ = 1$ and $s_- = 0$, a more evocative nomenclature would be "positive" and "ground," justified by the electric circuit analogy [68] that we discuss further in § 4.4.5. Having constructed the extra training vertices v_{i-}, we must connect them to the rest of the graph. To do so, we compute edge weights from the edge weights linking each sample to the v_{i-} vertices. First we require that the similarity function σ is bounded to the finite closed range $[s_-, s_+]$:

$$\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [s_-, s_+] \tag{4.13}$$

We assume that whenever σ evaluates to s_- that means the involved samples are entirely dissimilar, and whenever σ evaluates to s_+ that means the samples are entirely similar (or better put, equivalent for the purposes of comparing for similarity). Then we rely on the simple observation that, under these assumptions, a test pair $\langle\langle \mathtt{x}_j, \mathtt{y}_j \rangle\rangle$, $j \in \{\mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u}\}$, that is similar to a training pair $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$, $i \in \{1, \ldots, \mathtt{t}\}$, with similarity value s_{ij}, can also be considered dissimilar to the same training pair to the extent $s'_{ij} \triangleq 1 - s_{ij}$. Put another way, the test pair $\langle\langle \mathtt{x}_j, \mathtt{y}_j \rangle\rangle$ can be considered *similar* to the extent $1 - s_{ij}$ with an imaginary negative sample that complements the training point $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$. So the positive samples plus the bounded similarity value provide enough information for graph-based learning if we add one synthetic negative training sample for each positive training sample and amend the similarity function appropriately.

We will formalize these considerations in the definition below.

Definition 4.2.2 (Graph-Based Formulation of Structured Learning with Only Positive Training Samples)**.** Consider a structured learning problem defined by features $\mathtt{X} = \langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\rangle \subset \mathcal{X}^{\mathtt{t}+\mathtt{u}}$, training labels $\mathtt{Y} = \langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{t}} \rangle\rangle \subset \mathcal{Y}^{\mathtt{t}}$, similarity function $\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [0, 1]$, and hypothesis generator function $\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y})$. A similarity graph for the structured learning problem is an undirected weighted graph with real-valued vertex labels, constructed as follows:

- add one labeled vertex v_{i+} for each training pair sample $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$ $\forall i \in \{1, \ldots, \mathtt{t}\}$, with the label equal to 1;

- add one labeled vertex v_{i-} for each training pair sample $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$ $\forall i \in \{1, \ldots, \mathtt{t}\}$, with the label equal to 0;

- add one test vertex v_{ij} for each test sample consisting of a point $\mathtt{x}_i$ and a hypothesis $(\chi(\mathtt{x}_i))_j$, where $i \in \{\mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u}\}$ and $j \in \{1, \ldots, \operatorname{card}(\chi(\mathtt{x}_i))\}$;

- for each test vertex v_{ij} and each training vertices v_{k+} and v_{k-}, if $\sigma(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle, \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle) > 0$, define one edge linking v_{ij} to v_{k+} and one linking v_{ij} to v_{k-}, with the respective weights

$$\mathtt{w}_{ijk+} = \sigma(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle, \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle) \tag{4.14}$$

$$\mathtt{w}_{ijk-} = 1 - \mathtt{w}_{ijk+} \tag{4.15}$$

- for each pair of test vertices v_{ij} and v_{kl}, if $v_{ij} \neq v_{kl}$, define an edge linking them with weight

$$\mathtt{w}_{ijkl} = \sigma(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle, \langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle) \tag{4.16}$$

The resulting graph has paths passing from the training source vertices to their corresponding sink vertices through test vertices. The semi-supervised effect is brought about by the additional connections between test vertices. In practice, the graph (which might be very dense) may be approximated by only keeping its strongest edges.

One decision that needs close scrutiny is the choice of a linear function for the weight assignments $\mathtt{w}_{ijk-} = 1 - \mathtt{w}_{ijk+}$. The basic requirement is just a monotonically decreasing function defined on $[0, 1]$ and with a range in $[0, 1]$. Many monotonically decreasing functions could be chosen to map the range $[0, 1]$ onto itself, and the choice of a linear function must be justified appropriately. We show below that the choice is well grounded because, save for the semi-supervised effect, it computes s assignments consistent with the overall similarity with the training set, as proved in the theorem below. The theorem ignores for now any semi-supervised effect (induced by edges linking different hypotheses) and applies to the supervised subproblem. In that case we show that choosing the linear function in Eq. 4.15 leads to a sensible result: the score assignment for a given sample is, in fact, the averaged similarity between that sample and the training samples with which it bears similarity.

Theorem 4.2.3. *Consider a similarity graph for structured learning defined for features* $\mathtt{X} = \langle\!\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\!\rangle$, *positive training labels* $\mathtt{Y} = \langle\!\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\!\rangle$, *similarity function* $\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [0, 1]$, *and hypothesis generator function* $\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y})$. *Then, if all unlabeled-unlabeled edges are zero, label propagation will yield as solution the scores:*

$$s(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle) = \frac{1}{C_{ij}} \sum_{k=1}^{\mathtt{t}} \sigma(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle, \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle) \tag{4.17}$$

where C_{ij} *is the count of labeled vertices* k *for which* $\sigma(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle, \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle) > 0$.

Proof. We have shown that the harmonic function over the graph is unique and we also know that label propagation computes the harmonic function, so all we need to show is that the value in the hypothesis satisfies the harmonic property. A given vertex v_{ij} has C_{ij} edges to source vertices v_{ijk+}

$\forall k \in \{1, \ldots, C_{ij}\}$, and another C_{ij} edges to sink vertices $v_{ijk'+}$ $\forall k' \in \{1, \ldots, C_{ij}\}$. The weighted average of these two connections is

$$a_{ij} \triangleq \frac{\sum_{k=1}^{C_{ij}} \mathtt{w}_{ijk+} s(v_{k+}) + \sum_{k=1}^{C_{ij}} \mathtt{w}_{ijk-} s(v_{k-})}{\sum_{k=1}^{C_{ij}} \mathtt{w}_{ijk+} + \sum_{k=1}^{C_{ij}} \mathtt{w}_{ijk-}} \tag{4.18}$$

where $\mathtt{w}_{ijk+}$ and $\mathtt{w}_{ijk-}$ are the weights of the edges linking vertex v_{ij} to vertices v_{ijk+} and v_{ijk-}, respectively. In our case the scores $s(v_{k+})$ are all zero so they nullify the second sum in the denominator. Also, from the definition, $\mathtt{w}_{ijk+} + \mathtt{w}_{ijk-} = 1$ so the denominator sums up to C_{ij}, so we obtain

$$a_{ij} = \frac{1}{C_{ij}} \sum_{k=1}^{\mathtt{t}} \mathtt{w}_{ij+} s(v_{k+}) \tag{4.19}$$

which is exactly the harmonic condition. So s satisfies the harmonic property and, being unique, is the function computed by label propagation. □

This result shows that choosing the linear relation $\mathtt{w}_{ijk-} = 1 - \mathtt{w}_{ijk+}$ in Definition 4.2.2 leads to score assignments that (ignoring semi-supervised effect induced by unlabeled-unlabeled connections) are equal to the average similarity of each that hypothesis with the training samples it is similar to. Convergence to the trivial solution ($s = 1$ for all samples) has been avoided, and the scoring obtained is consistent with our notion of similarity: hypotheses that are more similar to some samples in the training set will receive higher scores.

The presence of unlabeled-to-unlabeled connections may improve score quality under the manifold assumptions discussed in Chapter 2: graph-based learning enforces not only consistency of the score across training and test data, but also across the test samples.

The resulting graph has a large number of vertices, two for each training sample and one for each hypothesis. Even if nearest-neighbor techniques are used for limiting the connectivity, scalability might become an issue. We introduce in § 5.5 a means to reduce the number of vertices by orders of magnitude without affecting the result of the learning process.

4.3 Similarity Functions for Structured Inputs and Outputs

In the formulation given above, the performance of the approach hinges on defining a good similarity function σ. A good similarity function should properly handle the structured nature of inputs and outputs, ultimately making for an expressive, smooth similarity.

Defining similarity across structured spaces is a recurring problem that has many applications beyond graph-based learning. This section provides a brief overview of such similarity functions. Many of the recently-studied similarity functions are *kernel functions*—functions that can be expressed as a dot product between two vectors associated with the inputs through a mapping functions. Section 4.3.1.3 introduces formal definitions for kernel functions. For graph-based learning

it is not necessary that the similarity function is a kernel function (the only requirements are positive and symmetric), but kernel functions have many desirable properties, notably an expressive representation and efficient evaluation.

Sequence Kernels Sequence kernels are a direct generalization of the traditional fixed-length feature vectors. Instead of defining one sample as one vector $\mathtt{x} \in \mathbb{R}^n$, sequence kernels define a sample as a variable-length catenation of such vectors, i.e.

$$\mathtt{x} = \langle\langle \mathtt{x}^{(1)}, \mathtt{x}^{(2)}, ..., \mathtt{x}^{(k)} \rangle\rangle \tag{4.20}$$

$$\mathtt{x}^{(i)} \in \mathbb{R}^n \quad \forall i \in \{1, \ldots, k\} \tag{4.21}$$

where $n \in \mathbb{N}^*$ is a constant but $k \in \mathbb{N}$ is a sample-dependent variable.

Sequence kernels have found natural applicability to systems using speech as inputs, for speech is a variable-length signal consisting of real-valued vectors (e.g. the cepstral coefficients [66, Ch. 6]). Campbell et al. [39] defined a sequence kernel suitable for training a Support Vector Machine along with an efficient mean-squared error training criterion method. They applied the sequence kernel to speaker recognition and language recognition tasks. Solomonoff et al. [205] used a similar setup to prevent loss of performance of a speaker recognition system in the presence of variations of handset and channel characteristics.

String Kernels String kernels are similarity functions defined over variable-length catenations of symbols extracted from a finite alphabet. There is some amount of confusion in literature about *string* kernels vs. *sequence* kernels, terms that are often used interchangeably. We consistently refer to sequence kernels as kernels over variable-length catenations of vectors of real numbers, and to string kernels as kernels over variable-length catenations of symbols extracted from a *finite* alphabet.

A simple example of a string kernel would be a 0/1 similarity that compares for lexicographic equality, but such a function would be too non-smooth to have any interesting properties. Edit distance gives a better notion of similarity than the 0/1 similarity, as do many other non-exact match measures. Naturally, string kernels are of particular interest to Human Language Technology applications because strings model human language text directly. Section § 4.4.5 discusses string kernels in detail, as they will be used in our application of Graph-Based Learning to Statistical Machine Translation.

Convolution Kernels Haussler [102] established a formalism for defining kernels over structured data having countable sets as support, including strings, trees, and graphs. His work was predated by research on string kernels, which he generalized into a framework also applicable to trees and graphs. A convolution kernel defines a kernel over a structure in terms of kernel evaluations on parts of that structure.

Tree Kernels Tree kernels are similarity measures between trees and are also of interest to Natural Language Processing because of their applicability to syntax trees. Collins and Duffy [51] describe tree kernels with NLP applications under the framework of convolution kernels and show applications to parse trees. Culotta and Sorensen [60] applied tree kernels to a relation extraction task. Vishwanathan and Smola [223] take the route of transforming the trees into strings by using a non-ambiguous mapping, followed by use of regular string kernels for tree comparison.

Graph Kernels Given that strings are restricted trees and trees are restricted graphs, a natural further generalization of structured kernels is defining similarities over graphs. (Graph kernels are not directly related to Graph-Based Learning.) Due to the formidable expressive power of graphs, graph kernels are the most difficult to define. In 2003, Gartner et al. [88] have shown that any similarity function on graphs that can fully recognize graph structure is NP-hard and also have shown that approximate matches are computable in polynomial time. One similarity criterion is based on the lengths of all walks between two vertices, and the other is based on the number of occurrences of given label sequences in labeled graphs (the more label sequences two labeled graphs have in common, the more similar they are deemed). Kashima and Inokuchi [115] define an approximate kernel by means of random walks of finite lengths and subsequently apply it to classification of chemical compounds [116]. Cortes et al. [53] introduced rational kernels, which operate efficiently on weighted transducers. Graph kernels are of interest to a host of Human Language Technology applications because graphs occur naturally in many input and output representations: many NLP tools produce syntactic and semantic information (such as named entities, dependency structures, anaphora, discourse relations) that can be most gainfully exploited in a graph framework [212]; finite state transducers are used in several HLT areas [165]; and the word or phrase alignments used by today's SMT systems form a bipartite graph.

Choosing a Kernel for Statistical Machine Translation We propose below an application of structured Graph-Based Learning to Statistical Machine Translation. For this kind of application, the inputs and outputs are sentences, which are highly structured entities, so either string, tree, or graph kernels could be investigated as candidates for σ. Of these, we chose string kernels because they are the simplest to operate with, most scalable, and most directly related to the most common automatic evaluation criterion for Statistical Machine Translation (SMT) (as discussed in 4.4.7). Before describing our proposed application of graph-based learning to SMT, we will discuss string kernels (our similarity measure of choice) in detail.

4.3.1 Kernel Methods

String kernels are an instance of kernel functions, an important concept of modern machine learning. This section introduces the appropriate background. Kernel methods [103] form a category of machine learning methods that has received increasing attention in the past years. This section reviews the main ideas behind kernel methods and builds background necessary to introduce string kernels, which in turn are the basis of our similarity measure.

Positive definite kernels are motivated by the need to apply linear methods to machine learning problems that can be best tackled by nonlinear systems. The kernel-based method essentially consists of mapping the input space to a different space, called the mapped space or the *feature space*. Afterwards, the linear machine learning method is used in that space. Although the learned function (e.g. a classification boundary) is linear, the transform is nonlinear so the relationship between the learned function and input space is nonlinear as well, leading, for example, to a curved decision boundary obtained through a linear classification algorithm. The mapping function must have properties that make it a good choice for the input space and is a good place to introduce problem-specific knowledge into the learning process. Also, efficient learning must be possible, which restricts both the mapping and the learning method in ways we will describe below. We first

define a kernel function formally.

Definition 4.3.1. Consider a function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. If there exists a Hilbert space $\mathcal{H}$ and a function $\Phi : \mathcal{X} \to \mathcal{H}$ such that

$$\kappa(\mathbf{x}, \mathbf{x}') = \left\langle \Phi(\mathbf{x}), \Phi(\mathbf{x}') \right\rangle \ \forall \mathbf{x}, \mathbf{x}' \in \mathcal{X} \tag{4.22}$$

then we call κ a *kernel function*, Φ a *feature map* of κ, and $\mathcal{H}$ the *feature space* associated with κ and Φ.

Choosing the right kernel for a given problem is an active area of research. Using a kernel function κ becomes advantageous under the following circumstances:

- The original linear machine learning method defines a dot product $\left\langle x, x' \right\rangle = \sum_{i=1}^{d} x_{[i]} x'_{[i]}$ on $\mathbb{R}^d$ and uses the dot product exclusively in calculations;
- The mapped space $\mathcal{H}$ is arguably a feature representation that is better amenable to linear methods (e.g., hyperplane separation) than the original space $\mathcal{X}$.

For example, the mapping $\Phi : \mathbb{R}^2 \to \mathbb{R}^3$, $\Phi(\langle\langle x_1, x_2 \rangle\rangle) = \langle\langle x_1^2, \sqrt{2} x_1 x_2, x_2^2 \rangle\rangle$ allows a plane in three dimensions to separate points that would be separated by an ellipsis in the original bi-dimensional space. Furthermore, the dot product in the mapped space is simply the square of the dot product in the initial space, as can be readily shown through simple algebraic manipulation. This means that a method that learns a separating hyperplane (e.g. a Support Vector Machine [32]) can be used for planar radial separation at virtually no added computational cost, in spite of it working in a higher-dimensional "intermediate" space.

To show that a given function κ is a kernel, it is necessary to define $\mathcal{H}$ and Φ analytically and show that the fundamental relationship in Eq. 4.22 holds. That might sometimes be difficult, so the question arose of finding out whether a function κ is a kernel by verifying properties of κ directly. The functions that are equivalent to dot products in mapped spaces are called *positive definite functions*, which we define below. The proof of equivalence between kernel functions and positive definite functions can be found in literature [103, 159, 59].

Definition 4.3.2. A real symmetric matrix $M \in \mathbb{R}^{n \times n}$ satisfying $\sum_{i,j} c_i c_j M_{ij} \geq 0 \ \forall k \in \mathbb{N}^*, \langle\langle c_1, \ldots, c_k \rangle\rangle \in \mathbb{R}^k$ is called *positive definite*. A function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ for which the matrix $K_{ij} \triangleq \kappa(\mathbf{x}_i, \mathbf{x}_j)$ (called the Gram matrix) is positive definite $\forall n \in \mathbb{N}^*, \langle\langle \mathbf{x}_1, \ldots, \mathbf{x}_n \rangle\rangle \in \mathcal{X}^n$ is called a *positive definite* function.

Substituting positive definite functions for dot products in machine learning algorithms automatically introduces mapped spaces and uses them in learning. Also, as kernels usually come in simple analytic form, they provide much faster evaluation than actually computing inner products in the mapped space directly. In fact the mapped space might be infinite-dimensional. Due to its remarkable effect of mapping the original features into a much more expressive feature space at a low cost, said substitution is called "the kernel trick" in the machine learning community.

The connection of kernel methods with graph-based learning becomes apparent if we observe that the usual similarity function defining the graph in Eq. 2.1

$$\mathtt{w}_{ij} = \exp\left[-\frac{d(\mathtt{x}_i, \mathtt{x}_j)^2}{\alpha^2}\right] \tag{4.23}$$

is, in fact, a kernel function [103].

Of particular interest to kernel methods are the *reproducing kernel Hilbert spaces* [62, 72], for which the mapped space has the form $\mathcal{H} = \mathcal{X} \to \mathbb{R}$ and is associated with a continuous kernel κ. The corresponding inner product is:

$$\left\langle \Phi(\mathtt{x}), \Phi(\mathtt{x}') \right\rangle = \int_{\mathcal{X}} \Phi_y(\mathtt{x}) \Phi_y(\mathtt{x}') dy \tag{4.24}$$

if $\mathcal{X}$ is continuous, and

$$\left\langle \Phi(\mathtt{x}), \Phi(\mathtt{x}') \right\rangle = \sum_{y \in \mathcal{X}} \Phi_y(\mathtt{x}) \Phi_y(\mathtt{x}') \tag{4.25}$$

if $\mathcal{X}$ is discrete (whether finite or not). Our next sections are concerned with defining reproducing kernel Hilbert spaces over discrete spaces only with the kernel value defined as per Eq. 4.25.

4.3.1.1 Normalized Kernels

For any kernel function, the inner product space defined by the mapping Φ is complete under the following norm definition [171, 187]:

$$\|\Phi(\mathtt{x})\| = \sqrt{\langle \Phi(\mathtt{x}), \Phi(\mathtt{x}) \rangle} = \sqrt{\kappa(\mathtt{x}, \mathtt{x})} \tag{4.26}$$

It is easy to verify that the properties of the norm are satisfied following the definition of a positive definite function, so any Hilbert space is also a Banach space with norm $\|\Phi\|$. (However not all Banach spaces can define a corresponding dot product.)

By the Cauchy-Schwartz inequality [171], we have:

$$\left|\left\langle \Phi(\mathtt{x}), \Phi(\mathtt{x}') \right\rangle\right| \leq \|\Phi(\mathtt{x})\| \cdot \|\Phi(\mathtt{x}')\| \tag{4.27}$$

$$\left|\kappa(\mathtt{x}, \mathtt{x}')\right| \leq \sqrt{\kappa(\mathtt{x}, \mathtt{x}) \kappa(\mathtt{x}', \mathtt{x}')} \tag{4.28}$$

which implies that the normalized kernel is bound within $[-1, 1]$ (or $[0, 1]$ if all kernel values are nonnegative, as is often the case). To introduce normalization, we need to define a distinguished subset of $\mathcal{X}$ as follows.

Definition 4.3.3. Given a kernel function κ defined on set $\mathcal{X}$, we define the *nonsingular kernel subset*, denoted $\mathcal{X}^{\kappa *}$, as:

$$\mathcal{X}^{\kappa *} \triangleq \mathcal{X} \setminus \{\mathtt{x} \in \mathcal{X} \mid \kappa(\mathtt{x}, \mathtt{x}) = 0\} \tag{4.29}$$

For any kernel, $\kappa(\mathtt{x}, \mathtt{x}) = 0 \Rightarrow \mathtt{x} = 0$, but we defined $\mathcal{X}^{\kappa *}$ as depending on $\kappa(\mathtt{x}, \mathtt{x}) = 0$ as opposed to $\mathtt{x} = 0$ because there are kernels that do not evaluate to 0 even in the origin (for which consequently $\mathcal{X} = \mathcal{X}^{\kappa *}$).

Definition 4.3.4. We define the *normalized feature map* $\hat{\Phi}$:

$$\hat{\Phi}(\mathtt{x}) : \mathcal{X}^{\kappa *} \to \mathbb{R}^{\mathcal{X}} \quad \hat{\Phi}(\mathtt{x}) = \frac{\Phi(\mathtt{x})}{\|\Phi(\mathtt{x})\|} \tag{4.30}$$

The function is well defined because the denominator is never 0; in fact it is easy to verify that

$$\|\hat{\Phi}(\mathtt{x})\| = 1 \quad \forall \mathtt{x} \in \mathcal{X}^{\kappa *} \tag{4.31}$$

The functional $\hat{\Phi}$ is a feature map of the *normalized kernel* $\hat{\kappa} : \mathcal{X}^{\kappa *} \times \mathcal{X}^{\kappa *} \to \mathbb{R}$.

$$\hat{\kappa}(\mathtt{x}, \mathtt{x}') = \left\langle \hat{\Phi}(\mathtt{x}), \hat{\Phi}(\mathtt{x}') \right\rangle = \left\langle \frac{\Phi(\mathtt{x})}{\|\Phi(\mathtt{x})\|}, \frac{\Phi(\mathtt{x}')}{\|\Phi(\mathtt{x}')\|} \right\rangle = \frac{\langle \Phi(\mathtt{x}), \Phi(\mathtt{x}') \rangle}{\|\Phi(\mathtt{x})\| \cdot \|\Phi(\mathtt{x}')\|} \tag{4.32}$$

$$= \frac{\kappa(\mathtt{x}, \mathtt{x}')}{\sqrt{\kappa(\mathtt{x}, \mathtt{x}) \cdot \kappa(\mathtt{x}', \mathtt{x}')}} \tag{4.33}$$

Normalized kernels are important to the study of kernel methods because they often eliminate the dependency of kernel's value on inconsequential characteristics of the inputs, such as size or sparseness, and also allow for easier combination with other similarity measures. Without normalization, kernel evaluation would yield values that only give a relative notion of similarity. Also, a normalized kernel engenders a distance with metric properties over $\mathcal{X}^{\kappa *}$, as we will show in the next section.

4.3.1.2 Relationship with Distance

Given a kernel function κ defined on a discrete set $\mathcal{X}$ with the mapping Φ with values in a reproducing kernel Hilbert space, consider computing the Euclidean distance between two points in mapped space:

$$d_\kappa\left(\mathtt{x}, \mathtt{x}'\right) = \sqrt{\sum_{w \in \mathcal{X}} \left(\Phi_w(\mathtt{x}) - \Phi_w(\mathtt{x}')\right)^2} \tag{4.34}$$

$$= \sqrt{\sum_{w \in \mathcal{X}} \Phi_w(\mathtt{x})^2 - 2 \sum_{w \in \mathcal{X}} \Phi_w(\mathtt{x}) \Phi_w(\mathtt{x}') + \sum_{w \in \mathcal{X}} \Phi_w(\mathtt{x}')^2} \tag{4.35}$$

$$= \sqrt{\kappa(\mathtt{x}, \mathtt{x}) - 2\kappa(\mathtt{x}, \mathtt{x}') + \kappa(\mathtt{x}', \mathtt{x}')} \tag{4.36}$$

Following a similar expansion, the normalized distance computes as follows:

$$\hat{d}_\kappa\left(\mathtt{x}, \mathtt{x}'\right) = \sqrt{\hat{\kappa}(\mathtt{x}, \mathtt{x}) - 2\hat{\kappa}(\mathtt{x}, \mathtt{x}') + \hat{\kappa}(\mathtt{x}', \mathtt{x}')} \tag{4.37}$$

By the definition of $\hat{\kappa}$, $\hat{\kappa}(\mathtt{x}, \mathtt{x}) = \hat{\kappa}(\mathtt{x}', \mathtt{x}') = 1$, so

$$\hat{d}_\kappa\left(\mathtt{x}, \mathtt{x}'\right) = \sqrt{2 - 2\hat{\kappa}(\mathtt{x}, \mathtt{x}')} \tag{4.38}$$

As Euclidean distances, d_κ and $\hat{d}_\kappa$ readily satisfy the metric properties (§ 3.4) directly from their definition. This is of high practical interest because many algorithms for fast nearest neighbors searching require at least a subset of the metric properties. For example, the kd-tree data structure

that we use in Chapter 5 for accelerating nearest neighbor computation, requires properties that Euclidean distance fulfills.

However, finding the closest neighbors according to d_κ does not necessarily find the most similar items: $d_\kappa(\mathtt{x}, \mathtt{x}') < d_\kappa(\mathtt{x}, \mathtt{x}'')$ that does not necessarily imply $\kappa(\mathtt{x}, \mathtt{x}') > \kappa(\mathtt{x}, \mathtt{x}'')$, i.e., kernel values are not in a monotonic relationship with the corresponding distances. Here is where normalization comes into play powerfully. We prove a simple theorem below that is of importance to practical approaches.

Theorem 4.3.5. *Consider a kernel* (κ, Φ) *and points* $\mathtt{x}, \mathtt{x}', \mathtt{x}'' \in \mathcal{X}^{\kappa *}$. *If* $\hat{d}_\kappa(\mathtt{x}, \mathtt{x}') < \hat{d}_\kappa(\mathtt{x}, \mathtt{x}'')$, *then* $\hat{\kappa}(\mathtt{x}, \mathtt{x}') > \hat{\kappa}(\mathtt{x}, \mathtt{x}'')$.

Proof. We take the difference of squares $\hat{d}_\kappa(\mathtt{x}, \mathtt{x}'')^2 - \hat{d}_\kappa(\mathtt{x}, \mathtt{x}')^2$ applying their simplified form in Eq. 4.38:

$$\hat{d}_\kappa(\mathtt{x}, \mathtt{x}'')^2 - \hat{d}_\kappa(\mathtt{x}, \mathtt{x}')^2 = 2(\hat{\kappa}(\mathtt{x}, \mathtt{x}') - \hat{\kappa}(\mathtt{x}, \mathtt{x}'')) \tag{4.39}$$

$\hat{d}_\kappa$ is a metric so $\hat{d}_\kappa(\mathtt{x}) \geq 0 \; \forall \mathtt{x} \in \mathcal{X}^{\kappa *}$. Consequently, $\hat{d}_\kappa(\mathtt{x}, \mathtt{x}'') > \hat{d}_\kappa(\mathtt{x}, \mathtt{x}') \Leftrightarrow \hat{d}_\kappa(\mathtt{x}, \mathtt{x}'')^2 > \hat{d}_\kappa(\mathtt{x}, \mathtt{x}')^2$, which immediately leads to the conclusion $\hat{\kappa}(\mathtt{x}, \mathtt{x}') - \hat{\kappa}(\mathtt{x}, \mathtt{x}'') > 0$. □

Practically, Theorem 4.3.5 shows that using a nearest-neighbors according to the normalized distance $\hat{d}_\kappa$ will find the most similar samples in a data set if it takes the precaution of eliminating points for which $\kappa(\mathtt{x}, \mathtt{x}) = 0$ (if any) from the potential candidates in the search. This is achieved easily in practice with negligible computational cost.

4.3.1.3 String Kernels

At the highest level, string kernels are simply kernel functions defined on $\mathcal{X} = \Sigma^*$ for some discrete vocabulary Σ. Use of mappings and similarity measures defined on Σ^* and akin to kernels was already widespread with strings prior to the introduction of kernel methods. For example, consider the following map:

$$\Phi : \Sigma^* \to \mathbb{R}^\Sigma \quad \Phi_w(s) = \operatorname{card}\{v, v' \in \Sigma^* \mid s = vwv'\} \tag{4.40}$$

We use the notation $\mathbb{R}^\Sigma$ as a shortcut for $(\Sigma \to \mathbb{R})$, i.e., the set of functions defined on Σ with values in $\mathbb{R}$. Also, we use the notation $\Phi_w(s)$ as a shortcut for the longer, more explicit notation $[\Phi(s)](w)$, which reveals that Φ is a functional applied to s yielding a function that in turn is applied to w. The shortcut notations are a generalization of the usual notations in multidimensional Euclidean spaces ($\mathbb{R}^n$ and x_k, respectively).

Eq. 4.40 defines a mapping that describes strings solely through the words they contain, without regard to their order, modeling technique known as the "bag-of-words" model (or, depending on the vocabulary used, "bag-of-characters"). If s is large (e.g., an entire document), word frequency has been shown to be a good predictor for the topic covered if the words are stemmed and if stop words (e.g., "and", "not") are eliminated [142, 109]. Taking the inner product in the corresponding

normalized mapped space yields:

$$\left\langle \hat{\Phi}(s), \hat{\Phi}(t) \right\rangle = \frac{\langle \Phi(s), \Phi(t) \rangle}{\|\Phi(s)\| \cdot \|\Phi(t)\|} = \frac{\sum_{w \in \Sigma} \Phi_w(s) \Phi_w(t)}{\sqrt{\left(\sum_{w \in \Sigma} \Phi_w(s)^2 \right) \left(\sum_{w \in \Sigma} \Phi_w(t)^2 \right)}} \tag{4.41}$$

which is nothing but *cosine similarity*, the popular similarity measure used in Information Retrieval [155] and NLP [200]. Joachims has first used the kernel properties of cosine similarity in a document categorization task using a Support Vector Machine as a classifier [109]. It is worth noting that in this case the kernel trick is not of use because the feature space is explicit and the kernel is computed directly as an inner product in feature space. The inner product can be completed in $\mathcal{O}(|\Sigma|)$ time if preprocessing extracts sorted feature vectors, computable in turn in $\mathcal{O}(|s| \cdot \log |s|)$ time for each input string s.

The p-Spectrum Kernel (n-gram kernel) Leslie et al. [139] introduced, in the context of a protein classification task, a direct generalization of the bag-of-words kernel that maps a string into the space of all possible strings of length exactly p. The kernel is also called the n-gram kernel in the NLP literature [92], for obvious reasons. The feature of a string s at coordinate $u \in \Sigma^p$ is the number of occurrences of u in s.

$$\Phi : \Sigma^* \to \mathbb{R}^{\Sigma^p} \quad \Phi_w(s) = \operatorname{card}\{v, v' \in \Sigma^* \mid s = vwv'\} \tag{4.42}$$

This definition is very similar to the bag-of-words kernel in Eq. 4.40, with the essential difference that in this case w is a fixed-length string of length p, whereas in Eq. 4.40 it is a single element of Σ. The inner product is computed in the expected manner:

$$\kappa(s,t) = \sum_{w \in \Sigma^p} \Phi_w(s) \Phi_w(t) \tag{4.43}$$

In this case the kernel trick is highly useful for computing the inner product, as enumerating all p-length substrings and matching them would take time exponential in p. Better approaches have been proposed that rely on preprocessing the strings into informative structures in linear time. Leslie et al. [139] built a trie data structure [26] out of one of the strings and achieved an overall time complexity of $\mathcal{O}(p(|s| + |t|))$. Using a generalized suffix tree [189, 223] reduces complexity to $\mathcal{O}(|s| + |t|)$.

The Mismatch Kernel Leslie et al. also introduced the mismatch kernel [140], which is similar to the p-spectrum kernel but makes the similarity measure smoother by allowing up to a constant $m < p$ mismatches in the substrings of length p. The mapping function is:

$$\Phi_w(s) = \operatorname{card}\left\{v, v' \in \Sigma^* \mid s = vuv' \wedge |u| = p \wedge m \geq \operatorname{card}\{i \in \{1, \ldots, p\} \mid w_i \neq u_i\}\right\} \tag{4.44}$$

The inner card function counts the mismatches between two strings of length p. The outer card therefore counts all possible substrings $u \in \Sigma^p$ of s that are within m mismatches from coordinate string w.

The implementation defines and uses a mismatch tree which is akin to a suffix tree [97]. The overall complexity obtained was $\mathcal{O}\left(\left(|s|+|t|\right)k^m|\Sigma|^m\right)$. Leslie and Kuang [138] subsequently introduced two more variations on the mismatch kernel: the substitution kernel, which models the probability of replacing one symbol with another, and wildcard kernel, which allows up to m instances of a special wildcard character in the match. All of these models have trie-based implementations and similar complexity profiles.

The n-Length Gap-Weighted String Kernel The mismatch kernel suggests further generalization to kernels that support arbitrary insertions of extra symbols in either of the strings, which opens the door to *gapped string kernels*, a family of string kernels of particular interest to NLP at large and to Statistical Machine Translation in particular. In order to describe gapped string kernels, we will first give a few additional definitions.

Definition 4.3.6. The string $s[i:j]$, $1 \leq i \leq j \leq |s|$ is the substring $s_i...s_j$ of s. We say that t is a *gapped subsequence* of s with indices vector $\mathbf{i}$ (denoted as $t = s[\mathbf{i}]$) if $\exists \mathbf{i} = \langle\langle i_1, \ldots, i_{|u|} \rangle\rangle \in \mathbb{N}^{|u|}$ with $1 \leq i_1 < \ldots < i_{|t|} \leq |s|$ such that $t_j = s_{i_j}\ \forall j \in \{1, \ldots, |t|\}$. The length of the gapped subsequence t with indices vector $\mathbf{i}$ is $i_{|t|} - i_1 + 1$. Σ^n is the set of all sequences of length n.

(We observe the convention prevalent in recent literature to consistently imply contiguity when discussing "substrings" and non-contiguity when discussing "subsequences.") We now define a mapping function that maps a string in Σ^* onto the space of all of its gapped subsequences of length n. The more "spread" a subsequence is (i.e., with more numerous and/or wider gaps), the less representative it is of the string as a whole. This intuition is formalized by using a penalty factor $\lambda \in (0, 1]$ that makes matches with longer gaps exponentially less important. The base of the exponentiation is λ and the exponent used is the total length of the gapped subsequence.

For the finite set Σ, $n \in \mathbb{N}^*$, and $\lambda \in (0, 1]$, we define the *n-length gap-weighted feature mapping with penalty λ* as the following functional:

$$\Phi : \Sigma^* \rightarrow \mathbb{R}^{\Sigma^n} \tag{4.45}$$

$$\Phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{i_{|u|}-i_1+1} \tag{4.46}$$

It is implied that $\Phi_u(s) = 0$ if there is no vector $\mathbf{i}$ such that $u = s[\mathbf{i}]$, i.e., strings s and u have no element in common. The corresponding kernel is defined as a regular dot product in mapped space:

$$\kappa(s,t) \triangleq \sum_{u \in \Sigma^n} \Phi_u(s)\Phi_u(t) \tag{4.47}$$

It would appear that the rich, informative similarity notion given by gapped matches is also its Achilles' heel. Computing all terms directly and then summing them is impractical, as they are combinatorially numerous. However, Lodhi et al. [149] defined a gapped string similarity kernel that requires $\mathcal{O}(n \cdot |s| \cdot |t|)$ time and space to compute 1-gapped, 2-gapped,..., n-gapped similarities between two strings s and t by using dynamic programming techniques. The cost is also incremental—each $\mathcal{O}(|s| \cdot |t|)$ iteration computes similarity for length m and saves state for computing similarity for length $m+1$. This is helpful because m-gapped similarity is a decreasing function

of m. An implementation might decide to stop computation early if the similarity has fallen below a threshold.

Finally, Rousu and Shawe-Taylor proposed a sparse dynamic programming approach that reduces complexity to $\mathcal{O}(n \cdot \mathrm{card}(M) \cdot \log|t|)$, where $M = \{\langle\langle i, j\rangle\rangle \in \mathbb{N}^2 \mid s_i = t_j\}$ is the set of index pairs of matching string elements.

The All-Lengths Gap-Weighted String Kernel A number of alternative kernels related to the above were proposed by Yin et al. [230] along with dynamic programming algorithms. Of particular interest is the all-lengths gapped kernel with the mapping function:

$$\Phi : \Sigma^* \to \mathbb{R}^{\Sigma^*} \tag{4.48}$$

$$\Phi_u(s) = \sum_{\mathbf{i}:u=s[\mathbf{i}]} \lambda^{i_{|u|}-i_1+1} \tag{4.49}$$

Note that in this case the codomain of Φ has changed from Σ^n to Σ^*, which means that the mapped space is now the space of all strings. In spite of this space being considerably larger, the dynamic programming algorithm only needs $\mathcal{O}(|s| \cdot |t|)$ time and $\mathcal{O}(\min(|s|, |t|))$ space. This, together with having less parameters, makes the all-lengths kernel potentially more attractive than the n-length kernel.

4.4 Structured Graph-Based Semi-Supervised Learning for Machine Translation

A field that has recently benefitted from steady progress in machine learning and equally steady growth of corpora sizes is Statistical Machine Translation (SMT). Although contemporary SMT systems have not achieved human-level translation capabilities on general text, they have made important inroads into tackling this difficult problem.

In the following we describe a practical application of our formulation of graph-based learning with structured inputs and outputs: an algorithm to improve consistency in phrase-based SMT. As we have discussed theoretically, we define a joint similarity graph over training and test data and use an iterative label propagation procedure to regress a real-valued scoring function over the graph. The resulting scores for unlabeled samples (translation hypotheses) are then combined with standard model scores in a log-linear translation model for the purpose of reranking. We evaluate our approach on two machine translation tasks and demonstrate absolute improvements of 2.6 BLEU points and 2.8% PER (without adaptation), and 1.3 BLEU points and 1.2% PER (with in-domain adaptation data) over state-of-the-art baselines on evaluation data.

Machine translation is a hard problem with highly structured inputs, outputs, and relationships between the two. Today's SMT systems are complex and comprise many subsystems that use various learning strategies and fulfill certain specialized roles. Applying a new learning technique to an SMT task is usually—and most effectively—carried by integrating the new learning technique within a multi-module SMT system and measuring the overall impact of that technique. To understand the motivation behind applying structured graph-based learning to SMT, a description of the standard architecture of a state-of-the-art SMT system is in order.

4.4.1 Architecture of Contemporary Phrase-Based SMT Systems

Contemporary SMT systems follow a fairly standard architecture. The essential flow consists of preprocessing, system training, decoding, and postprocessing. In the training stage, the model is trained by using parallel sentences in the source and target languages. The system-wide model consists of various models (such as a language model and a translation model), which all feed an overall log-linear probability model. Once the model is trained, a process called *decoding* is used to obtain estimated translation for test sentences. The decoder is a search engine (usually having no trained parameters) that searches for the translation that maximizes the probability of the translation given the test input. Decoding may entail the generation and rescoring of n-best lists, which is the framework we will focus on.

The test phase usually operates at sentence level: one input sentence is read, processed, translated, and "forgotten" as the next input sentence is read. This is the usual setup (although certain departures do exist [219]).

In the following we briefly describe the main activities performed by a phrase-based SMT system.

Preprocessing This is the activity performing all processing necessary for adapting raw text inputs to tokenized data (words). Subsequent stages operate at the token level. A system as simple as a vocabulary-driven indexer that identifies words separated by whitespace is a rough archetype of this stage. However, preprocessors may become much more involved depending on the input languages and on the task at hand. Scripts that have no explicit word separation (such as Chinese) require a learning machine procedure on its own for word segmentation [206, 231, 82]. Also, highly-inflected languages (such as German, Arabic, or Greek) benefit from a morphology-informed preprocessor lest the vocabulary size increases and relationships between various inflections of the same word are lost. Preprocessing also usually detects simple symbolic categories such as numbers and dates [193]. Preprocessing is performed on both source and target sides for the training data (by different subsystems that take into account the specificities of the source and target languages, respectively) and on the source side for test data.

Training Training the decoder is done with parallel texts in the source and target language. The basic approach aims at computing parameters that maximize $p(\mathtt{y}|\mathtt{x})$, which, after applying the Bayes rule, becomes:

$$\arg\max_{\mathtt{y}} p(\mathtt{y}|\mathtt{x}) = \arg\max_{\mathtt{y}} p(\mathtt{x}|\mathtt{y})p(\mathtt{y}) \tag{4.50}$$

Of the two factors, $p(\mathtt{y})$ is computed by using a *language model* on the target language side, a problem that has been investigated extensively [50, 209, 105, 45]. The *translation model* $p(\mathtt{x}|\mathtt{y})$ is the more difficult subsystem to train; a variety of training methods are being used, such as word-alignment induced phrases [126, 175], syntactic phrases [126], and phrase-alignment [126, 153]. Additional models may be used in the rescoring process, and the weights of the log-linear model associated with them are trained on the training set (parallel sentences in the source and target languages), usually by using Minimum Error Rate training [172]. Each subsystem participating in rescoring may be trained in a different way.

Decoding The decoding engine finds hypotheses $\mathtt{y}_{ij} = (\chi(\mathtt{x}_i))_j$ (candidate sentences in the target languages) that are the best candidates for translating test sentences $\mathtt{x}_i$. We slightly depart at this point from the prevalent notations used in SMT literature for source and target sentences (s and t) in order to integrate the SMT process within the notations we have used in the general section on structured learning. In the same vein it is worth noting that the use of "hypothesis" here is consistent with our definition of the term in § 4.1.

The decoder [125] essentially (a) segments each source (test) sentence into phrases; (b) translates each source phrase into a phrase in the target language; and (c) reorders the obtained phrases to obtain a translation. Each of these steps is subject to large variations. For a given sentence, several segmentations into phrases are possible. Also, for each phrase in the source language, several translation phrases are possible. Finally, for a given target phrase set, numerous reorderings are possible. In order to reduce the number of hypotheses, several other models estimate hypothesis probabilities and prune out unlikely translations. The other models may include a language model and a distortion model that accounts for word reorderings. These models are integrated within a log-linear model (described below in § 4.4.3), which associates an overall score with each hypothesis.

The decoder could be used as is by simply taking the so-called 1-best result, i.e. the hypothesis with the largest score. A better option is to have the decoder output the N*-best list*, which is a collection of the hypotheses that have received the N largest scores. In many SMT applications N is on the order of 10^3. N-best lists have the advantage of providing a good approximation of the hypothesis set, while also keeping its size within manageable limits.

Rescoring Also known as reranking, rescoring operates on the N-best lists output by the decoder. The hypothesis space has been reduced by the decoder, so here is the point at which more computationally-intensive models can be applied. The distinction between decoding and ranking stems therefore from practical necessity: mathematically, the models used in the rescoring stage could have been applied against the larger hypothesis space searched by the decoder, but that would have made the approach computationally infeasible.

The scores computed by the models in the rescoring stage are integrated within another instance of a log-linear model (§ 4.4.3). The scores computed by the decoder's various models are usually integrated within this last log-linear model. The model is in principle the same as the one used in the decoder, but is trained separately and possibly implemented following different engineering tradeoffs (as it operates on a smaller input space but a larger number of models).

The rescoring stage is where sophisticated models may be inserted in the overall system in a scalable manner, and is the point at which we insert our graph-based engine. Integration is facile because we defined structured learning to fit perfectly within a rescoring framework. The formalism for a hypothesis defined in § 4.1 corresponds to the notion of "hypothesis" as an element of the N-best list.

4.4.2 Phrase-Based Translation

The phrase-based approach to translation (as opposed to word-based) is the most important recent development in SMT, and is ubiquitous in today's systems. Phrase-based translation systems operate on phrases as the unit of translation. The translator divides source language text into phrases, translates each phrase into a target language phrase, and then possibly reorders the output phrases to

obtain the translated sentence. Phrases vary in length and a few given adjacent words may or may not fall within the same phrase(s). The context horizon for phrase segmentation is the sentence or the chunk (a large constituent of a sentence). Phrase lengths may differ across source and target, and include lengths 0 and 1.

There are several ways of dividing sentences in phrases and pairing phrases in the source and target languages. Och et al. [176] learn phrase alignments from a parallel corpus that has already been word-aligned. The popular Giza++ Machine Translation system [174] generates word alignments that can be subsequently used for learning phrases. Koehn et al. [126] add a number of heuristics to the process. Yamada and Knight [228] and Imamura [106] proposed choosing linguistically-motivated phrases, i.e. the system should only consider phrases that are constituents. Such a restriction has low coverage and eliminates many useful phrases, so it obtains inferior results when compared to statistical-based phrase learners. However, using syntactically-motivated phrases in conjunction with statistically acquired phrases has good performance and also reduces decoding time [100]. Finally, Marcu and Wong [153] proposed a model that learns phrases jointly, direct from an (unaligned) parallel corpus.

The system we use in our experiments is the University of Washington Machine Translation System [120], which uses Och's algorithm [176] for learning phrases from a word-aligned corpus. The word alignments are obtained with Giza++.

4.4.3 Log-Linear Models

During decoding and rescoring, the prevalent means of aggregating several models into one meta-model is log-linear modeling [20, 173]. Log-linear models (a.k.a. exponential models) are based on a powerful intuitive justification and an equally powerful mathematical justification. Intuitively, a good model that needs to respect certain constraints (usually presented in the form of experimental evidence) must not "overcommit," i.e. it should assume no other constraints except those presented; aside from respecting the given constraints, it should assume that the distribution of all data is as uniform as possible. This translates directly to intently choosing the model of *maximum entropy* from the universe of all constraint-abiding models. Mathematically, maximizing the conditional log-likelihood of the training data is equivalent to (i.e. is the convex dual of) minimizing the entropy subject to the given constraints [20].

A log-linear model receives as input several *feature functions* f_i:

$$f_i : \mathcal{X} \times \mathcal{Y} \to \mathbb{R} \quad \forall i \in \{1, \ldots, |f|\} \tag{4.51}$$

that map possible input/output pairings to real numbers (or categorical outputs). Some features may be defined only on $\mathcal{X}$ or on $\mathcal{Y}$. A notable category of feature functions are binary features, modeled as numbers in $\{0, 1\}$ (which the paper introducing log-linear models [20] has used exclusively). Although features f_i and the scoring function s have similar definitions, there is one notable difference: whereas the value of the score $s(\mathtt{x}, \mathtt{y})$ must increase with the feasibility/desirability of the pair $\langle\langle \mathtt{x}, \mathtt{y} \rangle\rangle$, there is no such requirement for a feature function f_i. The only requirement is that $f_i(\mathtt{x}, \mathtt{y})$ correlates, or inversely correlates, with the feasibility of $\langle\langle \mathtt{x}, \mathtt{y} \rangle\rangle$. Using these features,

the log-linear model computes the likelihood of a given pair as:

$$p_\lambda(\mathtt{y}|\mathtt{x}) = \frac{\exp\left(\sum_{i=1}^{|f|}\lambda_i f_i(\mathtt{x},\mathtt{y})\right)}{\sum_{\mathtt{y}'\in\mathcal{Y}}\exp\left(\sum_{i=1}^{|f|}\lambda_i f_i(\mathtt{x},\mathtt{y}')\right)} \tag{4.52}$$

where $\lambda \in \mathbb{R}^{|f|}$ is the vector containing the log-linear model's parameters, called feature weights or model scaling factors. The denominator ensures normalization for a properly-defined probability and need not be computed if only the $\arg\max p_\lambda(\mathtt{x},\mathtt{y})$ is of interest.

4.4.3.1 Training Log-Linear Models for SMT

For Statistical Machine Translation, the state-of-the-art method is Minimum Error Rate Training (MERT) proposed by Och in 2003 [172] and subsequently improved by Och and others [71, 150, 41]. MERT is a rather general training method that trains the parameters of the log-linear model to minimize a smoothed error count. The method is parameterizable by the training criterion; in SMT, training maximizes directly BLEU [180] or PER [173] against a development set. For example, training for maximizing BLEU solves the problem

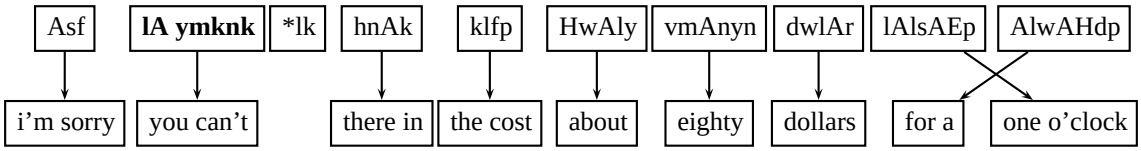
$$\lambda^* = \arg\max_\lambda \textsc{Bleu}(\mathbf{e}^*_\lambda; \text{references}) \tag{4.53}$$

where $^*_\lambda$ is the candidate translation obtained by using model parameters λ. The function is not smooth and has many local minima, which makes the $\arg\max$ search difficult. MERT selects the best candidate translation out of an n-best list (candidate translation) by using coordinate ascent; within an iteration the parameter that improves the score gets optimized while the others are fixed.

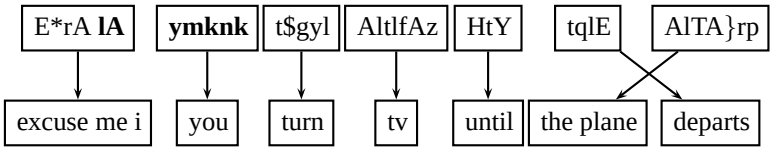
4.4.4 Constraining Translations for Consistency

The translation of a given sentence depends on the maximum global sentence score as computed by the final log-linear model in the rescoring stage. The global score may be dominated by different models at different times, and there is no inherent smoothing that fosters similar translations for similar input sentences. Therefore, it sometimes happens that similar test sentences receive rather different translations. This lack of smoothness reduces the cohesiveness of the translation and in fact may favor mistaken translations.

Consider the example in Fig. 4.1, taken from the IWSLT 2007 Arabic-to-English translation task [83]. The Arabic word "ymknk" means "you can" and "lA" negates it such that the phrase "lA ymknk" means "you may not"/"you cannot." In the first case, the Arabic sentence is segmented properly such that "lA ymknk" is put in correspondence to "you can't" which ultimately leads to an intelligible translation. However, in the second case, the segmentation choices were different as "lA" and "ymknk" were put in distinct phrases. This in turn led to a different translation for each word in the phrase and ultimately to the loss of the negation, which was semantically essential. The complex interactions between various components of the final log-linear model led to the surprising outcome of making apparently non-systematic mistakes when presented with similar inputs.



(Reference: sorry you can't there is a cost the charge is eighty dollars per hour)



(Reference: sorry you cannot turn the tv on until the plane has taken off)

Figure 4.1: Two baseline translations of Arabic sentences containing the same negation. The phrase "lA ymknk" ("you cannot"), where "lA" is the negation, is mistakenly segmented in the second example such that the negation is lost in the translated sentence.

A few possible approaches to addressing this problem are summarized below.

- One obvious solution is to improve the word alignments and phrase estimation. These in turn would reduce the number of incorrect segmentations.

- A confidence feature may be added for phrases to encourage frequent translations over less frequent ones.

- Similar input phrases might be forced to be always segmented in the same way. This approach falls prey to the well-known problem that natural language has many ambiguities that make proper segmentation possible only when context is taken into account (e.g. "You like Mary" vs. "You are like Mary").

Our proposed approach is to inject one additional feature function into the log-linear model that explicitly encourages *similar outputs for similar inputs*. We can naturally add a semi-supervised effect to this goal if we consider similar inputs not as measured between training sentences and test sentences, but also across different test sentences. Such a feature function may improve the translation quality: If the system issues good translations more often than bad ones, fostering consistency in translation would favor (by way of similarity with the majority) correct translations and would avoid incorrect ones.

The converse risk is that an overall poor translation will be hurt even more by dropping minority correct translations in favor of incorrect translations that are similar to other translations, also incorrect. To some extent we are able to control this effect by adjusting the relative weights of labeled-labeled and unlabeled-labeled connections. We assess improvements in the error rate by comparing the improved system with a baseline system using the standard method BLEU (discussed in detail in § 4.4.7). As far as the more subjective topic of fluency and coherence is concerned, we provide a few illustrative examples.

It is worth noting that obtaining similar translations for similar inputs is to some extent already ensured by the training process, but only in an implicit form. Essentially, the assumption that similar inputs lead to similar outputs is the basis of all statistical learning. However, the discrete nature of the signal and the interaction between models makes for sudden changes in input-to-output correspondence. Also, as mentioned in § 4.4.1, most of today's SMT systems "forget" a test sentence as soon as it was translated, so by design they are not conceived to enforce consistent translations across the entire test input. An explicit constraining feature that uses similarities between different test sentences could improve the self-consistency of a translation and also help with adaptation in case the domains and styles of the training and test data are slightly different.

The plan is therefore to add a new feature function to the log-linear model of a state-of-the-art phrase-based SMT. The new feature function is a regressed score as defined in § 4.1. This is possible because s readily fits the definition of a feature function (Eq. 4.51) and also because it is correlated with a suitably-defined similarity measure between sentences. That score is regressed using graph-based semi-supervised learning on a graph that uses sentence pairs (source plus target) as vertices and links them using similarity edges. Using the initial SMT system as a baseline, we evaluate the performance obtained after adding our feature function into the rescoring module. The following sections shape out the details of problem definition, choice of similarity function, data and system, experiments, and commented results.

4.4.5 Formulation of Structured Graph-Based Learning for Machine Translation

We first define a few concepts aimed at formalizing the notion of a sentence.

Definition 4.4.1. Let Σ be a finite set. A *string over alphabet* Σ is a finite catenation of elements from Σ. The concatenation of two strings s and t is denoted as st. The *length* of a string $s = s_1 s_2 \dots s_n$ is denoted as $|s| \triangleq n$ (for empty strings $|s| = 0$). The set of all strings over Σ is denoted as Σ^*.

A sentence in a language is therefore a string consisting of a concatenation of elements in the vocabulary of that language. In general, instead of words, the alphabet Σ may consist of larger units (e.g. phrases), smaller units (e.g. syllables or letters), or even derived units added through preprocessing such as word stems, roots, or other features. For now we use the word as a basic unit in Σ. For the purpose of translation, we define the source vocabulary Σ_S and a source sentence $\mathtt{x}$ as a string over Σ_S, so $\mathcal{X} = \Sigma_S^*$. In symmetry with source vocabulary and sentences we define the target vocabulary Σ_T and the set of target sentences as $\mathcal{Y} \in \Sigma_T^*$.

We construct our graph following Definition 5.5.1. Each test vertex represents a sentence *pair* (consisting of source and target strings), and edge weights represent the combined (source and target) similarity scores discussed in the next section. The hypothesis generator function χ is defined simply as the N-best list obtained from the first-pass decoding. We add a few parameterizations to the graph construction process aimed at speeding up the training process. Given a training set consisting of sentences $\mathtt{x}_1, ..., \mathtt{x}_\mathtt{t}$ that have the reference translations $\mathtt{y}_1, \dots, \mathtt{y}_\mathtt{t}$, a test set with sentences $\mathtt{x}_{\mathtt{t}+1}, ..., \mathtt{x}_{\mathtt{t}+\mathtt{u}}$, a scoring function s, and a hypothesis generator function χ, construction of the similarity graph proceeds as follows:

1. For each sentence $\mathtt{x}_i$, $i \in \{\mathtt{t}+1, \dots, \mathtt{t}+\mathtt{u}\}$ in the test inputs, compute a set Γ_{train_i} of similar training sentences and an ordered set of similar test sentences Γ_{test_i} by applying a similarity

function σ (discussed in the next section):

$$\Gamma_{\text{train}_i} = \langle\!\langle x \in \langle\!\langle \mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{t} \rangle\!\rangle \mid x \neq \mathtt{x}_i \wedge \sigma(\, \langle\!\langle x, \epsilon \rangle\!\rangle\, ,\, \langle\!\langle \mathtt{x}_i, \epsilon \rangle\!\rangle\,) \geq \theta \rangle\!\rangle \tag{4.54}$$

$$\Gamma_{\text{test}_i} = \langle\!\langle x \in \langle\!\langle \mathtt{x}_{\mathtt{t}+1}, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\!\rangle \mid x \neq \mathtt{x}_i \wedge \sigma(\, \langle\!\langle x, \epsilon \rangle\!\rangle\, ,\, \langle\!\langle \mathtt{x}_i, \epsilon \rangle\!\rangle\,) \geq \theta \rangle\!\rangle \tag{4.55}$$

where ϵ is the empty sentence. Using the empty sentence in the target position means that similarity is to be applied to source sides only. Note that we never compute similarities between two training samples because Γ_{train_i} and Γ_{test_i} are only defined for $\mathtt{t} < i < \mathtt{t} + \mathtt{u}$. Only those sentences whose similarity score exceeds some threshold θ are retained. The sentence $\mathtt{x}_i$ itself is not made part of Γ_{test_i}. Different values of θ can be used for training vs. test sentences; however, here we use the same θ for both sets.

2. For each test sentence-hypothesis pair $\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle$ that has a non-empty Γ_{train_i}, compute the similarity with each pair $\langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle$ $\forall \mathtt{x}_k \in \Gamma_{\text{train}_i}$. Similarity is defined by the similarity score

$$\mathtt{w}_{ijk} = \sigma(\, \langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle\, ,\, \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle\,) \tag{4.56}$$

If $\mathtt{w}_{ijk} > 0$, then connect the vertices $\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle$ and v_+ with an edge of weight $\mathtt{w}_{ijk}$, and connect the vertices $\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle$ and v_- with an edge of weight $1 - \mathtt{w}_{ijk}$.

3. Similarly, for each two pairs of test sentences and their hypotheses $\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle$ and $\langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle$, compute their similarity and use the similarity score as the edge weight between vertices representing $\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle$ and $\langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle$.

$$\mathtt{w}_{ijkl} = \sigma(\, \langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle\, ,\, \langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle\,) \tag{4.57}$$

Figure 4.2 shows a sample similarity graph. The setup is similar to a maximum flow problem with capacities proportional to edge weights, source v_+, and sink v_-. Indeed, after solving the maximum flow problem, the pressure at each edge is proportional to the regressed function [1, § 10.6]. Alternatively, there is an analogy with an electric circuit having v_+ connected to a 1V potential, v_- connected to the ground, and edge conductances given by their weights. In that network, the vertex potentials are equal to the regressed scoring function [1].

4.4.6 Decomposing the Similarity Function into Partial Functions

As defined in § 4.2, the similarity function σ accepts two pairs of inputs and outputs. However, the graph construction method defined in § 4.4.5 passes in certain cases the empty sentence ϵ to σ in the target sentence position. We need to define the semantics of σ appropriately such that it can handle incomplete arguments.

One simple approach is to require σ to be a mean of two partial functions, one operating on the input side and the other on the output side:

$$\sigma(\, \langle\!\langle x, y \rangle\!\rangle\, ,\, \langle\!\langle x', y' \rangle\!\rangle\,) = m(f_{\mathcal{X}}(x, x'), f_{\mathcal{Y}}(y, y')) \tag{4.58}$$

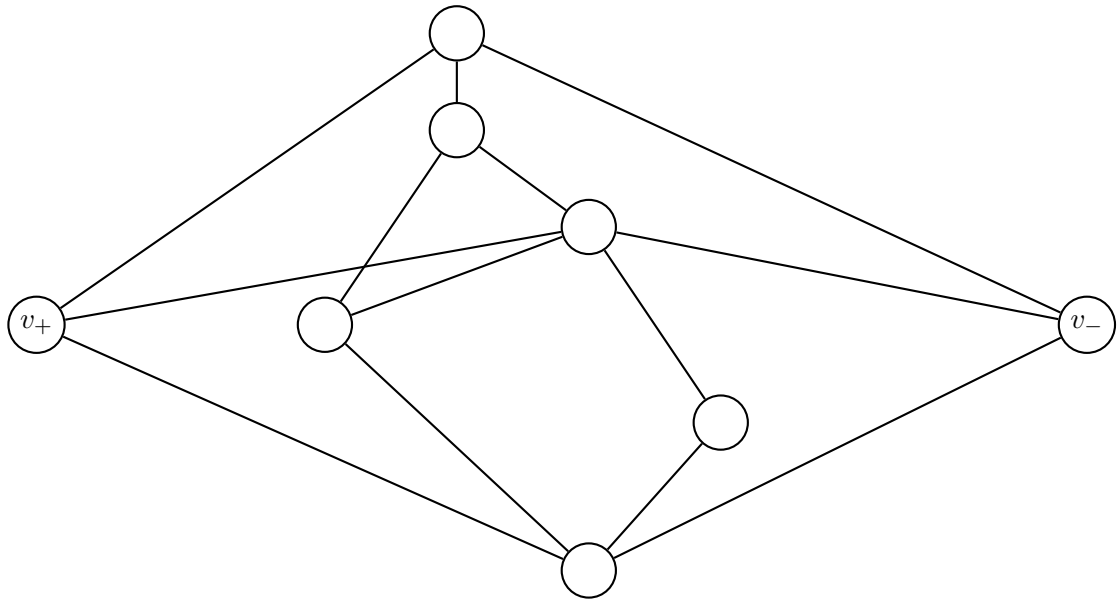


Figure 4.2: A similarity graph containing a source vertex v_+ with label 1, a sink vertex v_- with label 0, and several intermediate vertices that may or may not be connected directly to a source. Edge weights are not shown. Label propagation assigns real-valued labels at inner vertex that regress the harmonic function for the graph. A given vertex's label depends on its connections with the source and sink, and also of its connections with other vertices. In order to be meaningfully assigned a score, each test vertex must have a path (direct or indirect) to at least one of the train vertices v_+ and v_-.

where:

$$f_{\mathcal{X}} \quad : \quad \mathcal{X} \rightarrow [0,1] \tag{4.59}$$

$$f_{\mathcal{Y}} \quad : \quad \mathcal{Y} \rightarrow [0,1] \tag{4.60}$$

$$m \quad : \quad [0,1] \times [0,1] \rightarrow [0,1] \tag{4.61}$$

Functions $f_{\mathcal{X}}$ and $f_{\mathcal{Y}}$ compute partial similarities on the input and the output side respectively, and m is a mean function (e.g. arithmetic, geometric, or harmonic) that combines the two separate scores. This form is not appropriate for all structured learning problems because it is unable to capture dependencies between inputs and outputs. However, the SMT system we integrate with has several other models that are concerned with proper input-output mapping. Thus, we count on the rest of the system to capture such dependencies and we choose this measure on grounds of simplicity, noting that other similarity kernels that could capture alignment information might perform better here. This is an interesting venue for further research.

Due to the fact that the input and output domains are topologically similar (apart from being defined over different vocabularies), we choose to define σ as a mean of two identical functions

defined over strings of words from different vocabularies:

$$\sigma(\langle\langle x, y\rangle\rangle, \langle\langle x', y'\rangle\rangle) = m(\sigma_{\Sigma_S}(x, x'), \sigma_{\Sigma_T}(y, y')) \tag{4.62}$$

In this application we choose m to be the geometric mean. The geometric mean is better suited for our notion of similarity than arithmetic mean because in order for the geometric mean to be relatively large, *both* source and target side sentences must be similar. In contrast, arithmetic mean yields relatively large values for highly discrepant inputs. We confirmed that geometric mean yields better bottom-line results than arithmetic mean on our experimental test bed.

Geometric mean is still possibly suboptimal because it assigns equal importance to the source and target sides. The two sides have an inherent asymmetry because on the source side the sentence are always correct, whereas the target side comprises the test hypotheses, which are potentially incorrect, and the reference translation, which, being performed by human translators, is subject to considerable variability. The source side comparison is therefore more reliable; on the other hand, the target side comparison is also informative because it can distinguish good translations from bad ones. Additional sources of information regarding the translation (such as alignment) may be integrated in the definition of the mean function. This study does not pursue these potential directions.

The following sections focus on defining a similarity measure σ_Σ over sentences constructed over some general vocabulary Σ; it is assumed that they will be integrated into the composite similarity function σ as per Eq. 4.62. Choosing the similarity measure essentially determines the performance of the scoring function s. The similarity measure is also the means by which domain knowledge can be incorporated into the graph construction process. Similarity may be defined at the level of surface word strings, but may also include more linguistic information, such as morphological features, part-of-speech tags, or syntactic structures.

This study compares two similarity measures, the BLEU score [180] and a score based on string kernels. Whereas the former is a reasonable baseline choice because it uses the same optimization criterion for training and for evaluation, the latter yields better results in practice.

4.4.7 Using the BLEU *Score as Sentence Similarity Measure*

BLEU is one of the most popular automated methods for evaluating machine translation quality. It is based on the simple principle that the closer an candidate translation is to a correct translation, the better it is deemed to be by a human arbiter. In turn, extensive experiments have shown that good automated translations tend to share many n-grams with human-written reference translations for a range of small values of n. To compute the amount of shared n-grams for a given value of n, BLEU uses a measure called *modified precision*. The occurrences of each distinct n-gram in the candidate sentence are counted, but only up to the maximum number of occurrences of that n-gram among all reference sentences. (This prevents an artificially good precision for candidates consisting of repeated frequently-encountered n-grams.) Then the precision is computed normally by dividing the obtained count by the total number of distinct n-grams in the candidate sentence. The geometric mean of all modified precisions is computed for $n \in \{1, 2, 3, 4\}$. Finally, a brevity penalty BP multiplies the result because modified precision favors short sentences (e.g., a one-word sentence that matches one of the words in the reference sentence has modified precision equal to 1).

In detail, the BLEU score is given by the equation:

$$\text{BLEU} = \text{BP} \cdot \exp\left(\frac{1}{4}\sum_{n=1}^{4}\log p_n\right) \tag{4.63}$$

where p_n are probabilities computed for n-grams of length n as follows:[1]

$$p_n = \frac{\sum_{i=\mathtt{t}}^{\mathtt{t+u}} \sum_{\text{ngram}\in\mathtt{y}_i} \min\left(\text{Count}_{\mathtt{y}_i}(\text{ngram}), \text{MaxRefCount}_i(\text{ngram})\right)}{\sum_{i=\mathtt{t}}^{\mathtt{t+u}} \sum_{\text{ngram}\in\mathtt{y}_i} \text{Count}_{\mathtt{y}_i}(\text{ngram})} \tag{4.64}$$

The outer sum iterates over all sentences in the test set (p_n is computed globally). The inner sum iterates n-grams in one candidate translation. The quantity $\text{Count}_i(\text{ngram})$ is the count of a given n-gram within the candidate translation $\mathtt{y}_i$, and the quantity $\text{MaxRefCount}_i(\text{ngram})$ is the maximum number of occurrences of that n-gram in any of the reference translations of sentence $\mathtt{x}_i$. (In general, a translation task may avail itself of several reference translations.) Again, the $\min$ is taken to avoid repeated correct n-grams from imparting an artificially good quality to a translation.

The brevity penalty BP is computed as follows:

$$\text{BP} = \begin{cases} 1 & \text{if } c \geq r \\ \mathrm{e}^{(1-\frac{r}{c})} & \text{if } c < r \end{cases} \tag{4.65}$$

where $c \triangleq \sum_{i=\mathtt{t}}^{\mathtt{t+u}} |\mathtt{y}_i|$ is the total length of the candidate translation, and r is the length of the reference translation. When there are multiple reference translations, several variations exist as to how the reference translation length is defined. The original definition takes the reference length closest to c; the NIST definition [170] takes the shortest reference length; and other authors take the average length of all references. The data we used for experiments (§ 4.5) has only one reference available.

In using BLEU in our application, we consider each sentence one document, so the outer sum is inoperative. Also, there is only one "reference" (the other sentence in the similarity computation) so MaxRefCount is the same as Count for that sentence. So for two arbitrary sentences s and s' defined over the same vocabulary, p_n becomes:

$$p_n(s, s') = \frac{\sum_{\text{ngram}\in s} \min(\text{Count}_s(\text{ngram}), \text{Count}_{s'}(\text{ngram}))}{\sum_{\text{ngram}\in s} \text{Count}_s(\text{ngram})} \tag{4.66}$$

BLEU is not symmetric; in general $p_n(s, s') \neq p_n(s', s)$. For computing similarities between train and test translations, we use the train translation as the reference (in the s' position). For

[1] We slightly depart from the original notations [180] so as to integrate the equations within our notational system.

computing similarity between two test hypotheses, we compute BLEU in both directions and take the average.

BLEU is arguably a relevant sentence similarity measure to consider for σ at least as a baseline. This is because BLEU has been extensively used to measure the quality of translations and has been shown to correlate well with human judgment [180, 54]. In addition, the performance of the end-to-end SMT system is measured using BLEU, so it is sensible to internally use the same performance measure as the one used in evaluation.

However, BLEU has known disadvantages when applied to sentence-level similarity, as well as in general. Even as early as its initial introduction, BLEU has been meant and shown to be a good measure only at document level. The counts are accumulated over all sentences in the document and then BLEU is computed; computing BLEU for each sentence and then averaging the results would yield a different score. This introduces noise in the similarity graph and is ultimately optimizing a cost function that is not directly related to the final evaluation. Also, previous studies [38, 4, 47] have pointed out further drawbacks of BLEU. BLEU is not decomposable [47], meaning that a variation in the score for one individual sentence's translation is not always reflected into a corresponding variation of the overall translation score. Also, BLEU allows too much variation across translations of a given sentence, and a strict increase in BLEU is not always correlated with an increase in perceived quality of translation.

4.4.8 String Kernels as Sentence Similarity Measure

Guidance for a better sentence similarity measure can be found by analyzing some shortcomings of the BLEU scoring method. One issue is rigidity: n-grams fail to catch *approximate matches* that deliberately ignore extra words intercalated among the n-gram constituents. For example the phrases "lorem ipsum dolor sit amet" and "lorem dolor feugiat elit amet" have in common the subsequence "lorem dolor amet," albeit with gaps in both strings. To catch such approximate matches, we need a notion of similarity and substring matching that is more permissive than approaches based on n-grams. For example, from a BLEU scoring perspective, the phrases "red flower," "red beautiful flower," and "red pretty flower" only have two unigrams and no bigram in common. However, it is clear that the two sentences also have some longer-distance similarity because they both embed the string "red flower," albeit with a gap in the last two cases. Scoring based on n-grams fails to measure such similarity, and gapped matches are a smoother similarity measure between sentences that adapts to the variability of natural language. Granted, sometimes a missing or inserted word may dramatically alter the meaning of the sentence, but there also are combinatorially many gapped substrings in a given string, so as long as gapped string similarity is statistically "right" much more often than "wrong," it will properly tolerate and overcome the occasional noisy terms. A similarity measure based on gapped n-gram is fine-grained as it is composed of many terms. In contrast, for $n \in \{1, 2, 3, 4\}$, a sentence s may have cumulatively only up to $4 \cdot |s|$ distinct n-grams,[2] which makes each mistaken match relatively more expensive. The downside is that the high dimensionality of gapped similarity makes the function value vary wildly from very small (for most strings) to extremely large (for long strings that are almost equal). Normalization, discussed in § 4.3.1.1, is an effective measure against such a large dynamics. It is possible that more aggressive smoothing schemes could add to the effectiveness of the measure.

[2] Assuming padding with null symbols to the right.

A good starting point in defining a better similarity measure is to employ *edit distance* measures [61, 141], which allow, with penalty, sentences to match in spite of minute differences, gaps, and extraneous words. Efficient data structures and algorithms are known for computing edit distances. Edit distances have a strongly local bias, in that they require largely aligned strings and only allow for local differences between them. In contrast, translations of a given sentence may be correct in spite of considerable local differences, caused e.g. by ordering. To overcome the rigidity of the edit distance measure when used in a translation context, Watanabe and Sumita [226] introduced a modified edit distance measure that integrates the *tf-idf* criteria [195], measure that was subsequently used by Paul et al. [181] in defining a rescoring system for SMT. Other uses of edit distance as a similarity measure have been explored in the NLP literature [122, 119, 84, 183] and have involved a human translator in a semi-automated evaluation loop. For example [183], a human would edit (attempting a minimum of insertions, deletions, and replacements) an automated translation until it had the same meaning as the reference translation (but, crucially, not necessarily the same sentence structure); after that, the quality of the automated translation was assessed by using the edit distance between the automated translation and the human-modified translation.

String kernels (§ 4.3.1.3) are a general and efficiently computable similarity measure that is smoother than edit distance. To improve the match of a n-gapped string kernel with the BLEU score used for evaluation, we define a weighted kernel obtained by averaging over 4 different kernels. The BLEU score focuses not on one specific n-gram size, but instead computes a weighted average of similarities for all n-gram sizes up to a limit. The intent is to capture similarity between sentences with increased exigency. Experiments [180] have confirmed that similarity for unigrams reflects comprehensibility of the translation, whereas n-gram similarity for higher values of n reflects fluency (BLEU uses values of n up to 4).

A similarity function based on gap-weighted kernels of a fixed length n would generalize similarity as measured with BLEU by allowing gaps in the n-grams, but *only for one specific n-gram length*. In order to truly generalize BLEU scoring, we define similarity not as a gap-weighted similarity of length n, but instead as a weighted sum of gap-weighted similarities for sizes up to n. This way we finally obtain the kernel-based similarity definition, which we will use in our experiments.

Definition 4.4.2 (Kernel-Based Similarity for Machine Translation)**.** Given a finite set Σ, $n \in \mathbb{N}^*$, $\lambda \in (0, 1]$, and $W = \langle\langle w_1, \ldots, w_n \rangle\rangle \in \mathbb{R}_+^n$ with $\sum_{i=1}^{n} w_i = 1$, we define the *normalized gapped similarity of sequences over Σ up to length n with penalty λ and weights W* as:

$$\sigma_{\Sigma,n,\lambda,W} : \Sigma^* \times \Sigma^* \to [0, 1] \tag{4.67}$$

$$\sigma_{\Sigma,n,\lambda,W}(s, t) = \sum_{i=1}^{n} w_i \cdot \hat{\kappa}_{\Sigma,i,\lambda}(s, t) \tag{4.68}$$

where $\hat{\kappa}_{\Sigma,i,\lambda}$ is the normalized i-length gap-weighted string kernel over alphabet Σ with penalty λ.

In our experiments we use $n = 4$ because the evaluation method (BLEU) uses up to 4-gram similarity. The resulting similarity function is bounded by the interval $[0, 1]$, is 1 only for identical strings and 0 only for strings that do not share any word (assuming the weights vector W has no zero values). The function $\sigma_{\Sigma,n,\lambda,W}$ will be used on the source side and on the target side as partial

similarities in the overall similarity function as described by Eq. 4.62. The averaging function across the source and target side is geometric mean.

Here are a few examples showing values of $\sigma_{\Sigma,n,\lambda,W}$ for $n = 4$, $\lambda = 0.5$, and $W = \langle\langle 0.25, 0.25, 0.25, 0.25 \rangle\rangle$:
s_1 = *life is like a box of chocolate*
s_2 = *i would like a box of sweet chocolate*
s_3 = *your chocolate is in a box*
s_4 = *i have chocolate*
$\sigma_{\Sigma,n,\lambda,W}(s_1, s_2) = 0.444696$
$\sigma_{\Sigma,n,\lambda,W}(s_1, s_3) = 0.213679$
$\sigma_{\Sigma,n,\lambda,W}(s_2, s_4) = 0.134101$
$\sigma_{\Sigma,n,\lambda,W}(s_3, s_4) = 0.0833421$

As expected, similarity is strong when there are relatively many matches albeit with gaps (s_1, s_2), but is more pronounced when the order of word is different (s_1, s_3), when gaps are longer (s_2, s_4), or when strings only share few unigrams (s_3, s_4).

4.5 Experimental Setup

We evaluate our use of graph-based learning, with both BLEU and kernel similarity, against the IWSLT 2007 Italian-to-English and Arabic-to-English travel tasks [83, 197]. The Italian-to-English translation is a challenge task, where the training set consists of read sentences, but the development and test data consist of spontaneous simulated dialogs between would-be travel agents and hypothetical tourists seeking information, extracted from the SITAL corpus [40]. This is a particularly interesting task because it requires some adaptation capabilities of the model. The Arabic-to-English translation challenge, known as the "classic vintage" BTEC task consists of travel expressions similar to those found in tourist phrasebooks. For our experiments we chose the text input (correct transcription) condition only. The data set sizes are shown in Table 4.1.

Set	# sent pairs	# words	# refs
IE train	26.5K	160K	1
IE dev_1	500	4308	1
IE dev_2	496	4204	1
IE eval	724	6481	4
AE train	23K	160K	1
AE dev_4	489	5392	7
AE dev_5	500	5981	7
AE eval	489	2893	6

Table 4.1: Data set sizes and reference translations count (IE = Italian-to-English, AE = Arabic-to-English).

We divided the Italian-to-English development set into two subsets: dev_1 containing 500 sentences, and dev_2 containing 496 sentences. We use dev_1 to train the system parameters of the

baseline system and as a training set for GBL. Then dev_2 is used to tune the GBL parameters. In keeping with most of today's SMT systems, we used additional out-of-domain training corpora in the form of the Italian-English Europarl corpus [124] and 5.5M words of newswire data (LDC Arabic Newswire, Multiple-Translation Corpus and ISI automatically extracted parallel data) for the respective languages. The additional training data was used both by the baseline system and the GBL system.

The baseline system is created out of the components usually employed by the SMT research community and yields results on a par with today's state-of-the-art. Our baseline is a standard phrase-based SMT system based on a log-linear model (§ 4.4.3) with the following feature functions:

- two phrase-based translation scores;
- two lexical translation scores;
- word count and phrase count penalty;
- distortion score;
- language model score.

We use the Moses confusion network-based decoder [128] with a reordering limit of 4 for both languages. The decoder generates n-best lists of up to 2000 non-unique hypotheses per sentence in a first pass. In the second pass a trigram model based on parts of speech is used. The part of speech sequences are in turn generated by a Maxent tagger [186]. The language models are trained on the English side using SRILM [209] and modified Kneser-Ney discounting for the first-pass models, and Witten-Bell discounting for the POS models. Refer to [120] for more details about the machine translation system.

4.6 Experiments and Results

We first investigated the effect of only including edges between labeled and unlabeled samples in the graph on the Italian-to-English system. This eliminates any semi-supervised effect as similarities among test samples are not taken into consideration. The graph containing only unlabeled-to-labeled edges is equivalent to using a weighted nearest neighbor ranker that, for each hypothesis, computes average similarity with its neighborhood of labeled points, and uses the resulting average for reranking. The GBL-learned score is made part of the log-linear model, and weights are retrained for all models.

Starting with the Italian-to-English task and the BLEU-based similarity metric, we ran parameter optimization experiments that varied the similarity threshold and compared arithmetic vs. geometric mean of source and target similarity scores. Geometric mean was consistently better experimentally. As mentioned in § 4.4.6, our conjecture is that geometric mean is better suited for decomposing the similarity function because it better penalizes similarity between sentences that are highly discrepant across languages (very similar in one language and very dissimilar in the other). In this experimental stage we also chose $\theta = 0.7$.

Weighting	dev_2	eval
n/a (baseline)	22.3/53.3	29.6/45.5
(a)	23.4/51.5	30.7/44.1
(b)	23.5/51.6	30.6/44.3
(c)	23.2/51.8	30.0/44.6

Table 4.2: GBL results (%BLEU/PER) on the IE task for different weightings of labeled-labeled vs. labeled-unlabeled graph edges (BLEU-based similarity measure).

4.6.1 Experiments on Italian-to-English Translation Using BLEU *as Similarity Measure*

After the initial stage, we performed our main experiments with three different setups affecting the strength of the semi-supervised effect, as shown below.

(a) *no weighting:* similarities are kept as they are;

(b) *strongly favor supervision:* labeled-to-unlabeled edges were weighted 4 times stronger than unlabeled-unlabeled ones;

(c) *mildly favor supervision:* labeled-to-unlabeled edges were weighted 2 times stronger than unlabeled-unlabeled ones.

The weighting schemes lead to similar results. The best result obtained (b) shows a gain of 1.2 BLEU points on the development set and 1.0 BLEU points on the evaluation set, reflecting PER gains of 2% and 1.2%, respectively.

4.6.2 Experiments on Italian-to-English Translation Using the String Kernel

We next tested the string kernel based similarity measure. The parameter values were a gap penalty $\lambda = 0.5$, a maximum substring length of $k = 4$, and weights of 0.0, 0.1, 0.2, 0.7, for unigrams, bigrams, trigrams, and 4-grams respectively. These values were chosen heuristically and were not tuned extensively. Results (Table 4.3) show improvements in both development and test set. The absolute gains on the evaluation set are 2.6 BLEU points and 2.8% PER.

System	dev_2	eval
Baseline	22.3/53.3	29.6/45.5
GBL	24.3/51.0	32.2/42.7

Table 4.3: GBL results (%BLEU/PER) on the Italian-to-English IWSLT 2007 task with similarity measure based on a string kernel.

The BTEC task has test data with different characteristics than the training data, which means that an adaptive machine learning system would be at an advantage. Graph-based learning is inherently adaptive, so it is interesting to gauge to what extent adaptation contributed the better performance of the GBL system.

GBL being an inherently adaptive technique, a natural question to ask is whether the improvements brought by GBL still hold when a small amount of in-domain data is available. To effect adaptation in the baseline, we train the baseline system on the concatenation of the development and training set. This avails the phrase table of the phrases that are stylistically different from the train set and close to the test set. We first optimized the log-linear model combination weights on the entire dev_{1+2} set (the concatenation of dev_1 and dev_2 in Table 4.1) before retraining the phrase table using the combined train and dev_{1+2} data. The new baseline performance (shown in Table 4.4) is, as expected, much better than before, due to the improved training data. We then added GBL to this system by keeping the model combination weights trained for the previous system, using the N-best lists generated by the new system, and using the combined train+dev_{1+2} set as a train set for selecting similar sentences. We used the GBL parameters that yielded the best performance in the experiments described above. GBL again yields an improvement of 1.3 BLEU points and 1.2% absolute PER.

System	BLEU (%)	PER
Baseline	37.9	38.4
GBL	39.2	37.2

Table 4.4: Effect (shown on the evaluation set) of GBL on the Italian-to-English translation system trained with train+development data.

4.6.3 Experiments on Arabic-to-English Translation

For the Arabic-to-English task we used the threshold $\theta = 0.5$ and an identical setup for the rest of the system. Results using BLEU similarity are shown in Table 4.5. The best GBL system improved results on the evaluation set yields by 1.2 BLEU points, but only by 0.2% absolute in PER. Overall, results were highly sensitive to parameter settings and choice of the test set. For example, testing against dev_5, a surprisingly large improvement in of 2.7 BLEU points was obtained.

Overall, sentence similarities were observed to be lower for this task. One reason may be the already known difficulties in tokenizing Arabic text [99, 76]. The Arabic-to-English baseline system includes statistical tokenization of the source side, which is itself error-prone in that it can split the same word in different ways depending on the context. Since our similarity measure has word-level granularity, this dampens the similarity of sentences on the source side making some of them fall below the threshold. The string kernel does not yield sensible improvement over the BLEU-based similarity measure on this task. Two possible improvements would be to use sub-word granularity on the source side (which would, however, impact adversely the speed of the system), and/or use an extended string kernel that can take morphological similarity into account.

Method	dev_4	dev_5	eval
Baseline	30.2/43.5	21.9/48.4	37.8/41.8
GBL (BLEU similarity)	30.3/42.5	24.6/48.1	39.0/41.6
GBL (kernel similarity)	30.6/42.9	24.0/48.2	38.9/37.8

Table 4.5: GBL results (%BLEU/PER) on the Arabic-to-English IWSLT 2007 task with similarity measure based BLEU, $\theta = 0.5$.

4.6.4 Translation Example

Below we give an actual example of a translation improvement, showing the source sentence, the 1-best hypotheses of the baseline system and GBL system, respectively, the references, and the translations of similar sentences in the graph neighborhood of the current sentence.

Source	*Al+ mE*rp Aymknk {ltqAT Swrp lnA*
Baseline	*i'm sorry could picture for us*
GBL	*excuse me could you take a picture of the us*
References	*excuse me can you take a picture of us*
	excuse me could you take a photo of us
	pardon would you mind taking a photo of us
	pardon me could you take our picture
	pardon me would you take a picture of us
	excuse me could you take a picture of us
Similar sentences	*could you get two tickets for us*
	please take a picture for me
	could you please take a picture of us

Source	*Al+ mE*rp Ayn Tryq Al+ xrwj*
Baseline	*excuse me where the way to go out*
GBL	*excuse me where is the way to go out*
References	*excuse me where's the way out*
	pardon me how do i get out of here
	excuse me where's the exit
	pardon me where is the exit
	excuse me where's the way out
	excuse me where's the way out
Similar sentences	*where is the music hall*
	where is the household appliances department
	where is the fancy goods department
	where's the air france counter

4.7 Related Work

There are several recent approaches of structured problems with GBL. Our work is the first attempt at formalizing and applying GBL to SMT in particular.

Ueffing et al. [219] apply self-training—a different semi-supervised learning method—to SMT, with a focus on adaptation, obtaining improvements on French-to-English and Chinese-to-English translation tasks. Altun et al. [7] apply transductive graph-based regularization (a method akin to label propagation that also works on a similarity graph) to large-margin learning on structured data. The graph regularizer leads to a more expressive cost function (which e.g. is more robust in the presence of noisy training samples), but requires solving a quadratic program,with which scalability quickly becomes an issue. String kernel representations have been used in SMT in a supervised framework [213]. Finally, our approach can be compared to a probabilistic implementation of translation memories [156, 221, 132]. Translation memories are intended to help human translators by offering a database, a fuzzy query language, and an interactive console. The human translator can consult the database for translations with a source sentence (or segment) similar to the sentence (segment) to be translated. A semi-supervised aspect of translation memory systems is that the operator may also update the database with a new translation that is deemed correct. Our system not only is entirely automated, but is able to propagate similarity (akin to a fuzzy match in a translation memory) from other unknown sentences to the sentence of interest. Marcu et al. proposed a combination of a translation memory with statistical translation [152]; however, that is a combination of word-based and phrase-based translation predating the current phrase-based approach to SMT.

Chapter 5

SCALABILITY

This chapter discusses how the proposed graph-based learning approaches can be applied to large data sets, which are frequent in realistic NLP problems.

Scalability is a general term with several definitions; this chapter loosely uses the term "scalability" to refer to the ability of an algorithm to operate on large data sets (e.g., contemporary HLT corpora), as well as the ability of achieving results faster and/or operate on larger data sets when more computational resources are added. Scalability is affected by several factors, the most important being algorithmic complexity. Algorithms that require $\mathcal{O}(n^k)$ time and/or space (where n is the size of the input) have difficulty scaling up for $k > 1$. Colloquially, an algorithm is considered scalable if its time and space complexity are $\mathcal{O}(n \log n)$ or better. Algorithms (and the structure they impose over data) are the most important aspect of creating a scalable system. Also, a dimension of algorithms that has become of high importance today is parallelization. Often scalability is concerned with improving the speed or capacity of a system in proportion to the computational resources available to it. An algorithm that can be decomposed in separately-computable tasks scales better than one with a more serial data dependency pattern.

The statistical properties of data also affect scalability of a learning system. Machine learning algorithms often make fundamental assumptions about their input's properties. The extent to which these assumptions are met affects the running time of the algorithm. For example, a neural network will take a longer time to train if data is noisy and not easily separable.

Last but not least, implementation and systems-level optimization aspects are not to be ignored. Often, changing a constant factor that is irrelevant with regard to complexity influences the time behavior of the algorithm considerably. Furthermore, on contemporary systems featuring deep memory hierarchies, data set size often affects speed dramatically, sometimes leading to paradoxical effects.

With regard in particular label propagation, the essential scalability issues can be summarized as follows:

- *In-core graph size:* the matrices $\mathtt{P_{UL}}$ and $\mathtt{P_{UU}}$ grow with $\mathtt{u} \cdot \mathtt{t}$ and $\mathtt{u}^2$ respectively;

- *Graph building time:* building $\mathtt{P_{UL}}$ and $\mathtt{P_{UU}}$ entails computing the similarities $\sigma(\mathtt{x}_i, \mathtt{x}_j)$, $i \in \{1, \ldots, \mathtt{t}\}$ and $j \in \{\mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u}\}$, plus the similarities $\sigma(\mathtt{x}_i, \mathtt{x}_j)$, $i, j \in \{\mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u}\}$, which, in a direct implementation, adds to a total count of similarity evaluations $\mathtt{t} \cdot \mathtt{u} + \frac{\mathtt{u}(\mathtt{u}-1)}{2}$. Such computation becomes prohibitive even for moderately-sized data sets.

- *Hyperparameter tuning:* Optimization of ancillary hyperparameters is a machine learning problem in its own right. A poorly chosen hyperparameter can affect the algorithm adversely, whereas an extensive hyperparameter tuning process adds to the total running time of the

algorithm. Tuning is particularly important in semi-supervised learning: The dearth of labeled data typical to SSL setups also translates to low availability of cross-validation data (which is used for tuning model parameters).

In wake of the varied concerns raised by scalability, this chapter includes a mix of theoretical-algorithmic and practical-implementation considerations. Scalability being a cross-cutting issue, we believe that the best strategy is a holistic approach that systematically addresses the problem at all levels. Throughout this chapter, we will show how scalability is improved by the following tactics:

- *Improve algorithmic complexity:* For the graph building step, we exploit the structure of the input (feature) space. The exact method depends on the properties of the space, for example we use very different approaches in string space (Chapter 4) versus continuous space (Chapter 3). For the label propagation step, we define an accelerated sequential convergence algorithm and a parallel extension of it.

- *Reduce the in-core data set size:* Given a set of training and test data, we are aiming at reducing the size of the in-memory structures that support the label propagation algorithm.

- *Use simple, scalable, principled hyperparameter tuning:* Hyperparameter tuning for the Gaussian kernel used in conjunction with distance measures (§ 2.1) is a learning problem of its own, which affects the duration of graph construction. In this chapter we propose a simple and scalable tuning method inspired from maximum margin techniques.

Our approach to reducing computation and shrinking $\mathtt{P}_{\mathtt{UL}}$ and $\mathtt{P}_{\mathtt{UU}}$ in size is to take advantage of the structure of the input features to efficiently estimate the most similar items. Then, we approximate the rest to zero. The result is a graph with fewer edges—an approximation of the "real" graph, but one that is of good quality because the most important edges are kept. (In fact, in most problems, the similarity measure is only an estimate of the real similarity between samples, so eliminating low-weight edges often helps reducing noise in the graph.) Given that the approximate $\mathtt{P}_{\mathtt{UL}}$ and $\mathtt{P}_{\mathtt{UU}}$ have many slots equal to zero, we can store them as a sparse matrices [70], which solves the size scalability issue too.

We start by proving a few properties of interest of the label propagation algorithm, after which we will give an improved definition of the algorithm. The properties concern the evolution of intermediate solutions (the $\mathtt{f}_{\mathtt{U}}$ matrix) during iteration towards convergence, and will allow us to devise algorithms that converge faster. We will show that intermediate solutions grow monotonically when starting from zero, and that improving an arbitrary element of $\mathtt{f}_{\mathtt{U}}$ improves the global solution as well (individual improvements are never in competition).

5.1 Monotonicity

How do elements of $\mathtt{f}_{\mathtt{U}}$ evolve throughout Algorithm 1? Answering this question gives insight into accelerating convergence, and also gives information about numeric stability and early stopping. To this end we provide the following theorem.

Theorem 5.1.1. *When starting with $\mathtt{f}'_{\mathtt{U}} = 0$ in Algorithm 1, each element of $\mathtt{f}_{\mathtt{U}}$ increases monotonically towards convergence.*

Proof. By induction over the iteration step t.

Base: For $t = 1$, $\mathtt{f}_{\mathtt{U}}^{\text{step } 1} = \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}}$. Since $\mathtt{f}_{\mathtt{U}}^{\text{step } 0}$ started at zero, $\left(\mathtt{f}_{\mathtt{U}}^{\text{step } 1}\right)_{ij} \geq \left(\mathtt{f}_{\mathtt{U}}^{\text{step } 0}\right)_{ij}$ $\forall i \in \{1, \ldots, \mathtt{u}\}, j \in \{1, \ldots, \ell\}$.

Inductive step: At step $t + 1$

$$\mathtt{f}_{\mathtt{U}}^{\text{step } t+1} - \mathtt{f}_{\mathtt{U}}^{\text{step } t} = \mathtt{P}_{\mathtt{UU}}\mathtt{f}_{\mathtt{U}}^{\text{step } t} + \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} - \mathtt{P}_{\mathtt{UU}}\mathtt{f}_{\mathtt{U}}^{\text{step } t-1} - \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}} \tag{5.1}$$

$$= \mathtt{P}_{\mathtt{UU}}\left(\mathtt{f}_{\mathtt{U}}^{\text{step } t} - \mathtt{f}_{\mathtt{U}}^{\text{step } t-1}\right) \tag{5.2}$$

By the induction hypothesis, $\mathtt{f}_{\mathtt{U}}^{\text{step } t} - \mathtt{f}_{\mathtt{U}}^{\text{step } t-1}$ has only positive elements, so all elements in the product are also positive. □

Note that since $\mathtt{f}_{\mathtt{U}}^{\text{step } 1} = \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}}$, it is trivial to verify that

$$(\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}})_{ij} \leq \left(\mathtt{f}_{\mathtt{U}}^{\text{step } \infty}\right)_{ij} \leq 1 \quad \forall i \in \{1, \ldots, \mathtt{u}\}, j \in \{1, \ldots, \ell\} \tag{5.3}$$

where $\mathtt{f}_{\mathtt{U}}^{\text{step } \infty}$ is $\mathtt{f}_{\mathtt{U}}$ after convergence. This suggests that for faster convergence a good choice for the initial $\mathtt{f}_{\mathtt{U}}$ is in the middle of its possible range:

$$\left(\mathtt{f}_{\mathtt{U}}^{\text{step } 0}\right)_{ij} = \frac{(\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_{\mathtt{L}})_{ij} + 1}{2} \tag{5.4}$$

We will use monotonicity to a greater effect in stochastic label propagation in § 5.2. Also, monotonicity has an important consequence with regard to numeric stability. As opposed to convergence through alternating values, monotonic convergence always finishes even when computation is affected by limited precision.

5.2 Stochastic Label Propagation

The order in which graph vertices are considered, i.e., the order of rows in $\mathtt{P}$ and $\mathtt{Y}_{\mathtt{L}}$ do not matter for convergence beyond node identity because none of the previously demonstrated theorems rely on a specific order. Indeed, in the method of relaxations [68] (which is a serial uni-dimensional label propagation iteration) nodes are spanned in *some* order, not a specific order. Intuitively, the order could even be changed from one epoch to the next. In the following we prove a powerful theorem that states not only that nodes can be spanned in any order, but even in random order, without regard to possibly updating a node several times before updating all (or any) others. Of course, doing this practically would be detrimental to performance, but this theorem has important consequences with regard to unsynchronized parallel execution of label propagation, as well as accelerated serial implementations.

First, let us introduce a new algorithm for label propagation. Instead of using matrix algebra to update all elements of $\mathtt{f}_{\mathtt{U}}$ in one epoch, Algorithm 2 updates exactly one randomly-chosen element at a time.

Algorithm 2: Stochastic Label Propagation

Input: Labels $\langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_\mathtt{t} \rangle\rangle \in \{1, \ldots, \ell\}^\mathtt{t}$; similarity matrix $\mathtt{W} \in \mathbb{R}_+^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$ with $\mathtt{w}_{ij} = \mathtt{w}_{ji} \geq 0 \; \forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; tolerance $\tau > 0$.

Output: Matrix $\mathtt{f}_\mathtt{U} \in [0,1]^{\mathtt{u}\times\ell}$ containing unnormalized probability distributions over labels.

$\mathtt{w}_{ii} \leftarrow 0 \; \forall i \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$;

$\mathtt{p}_{ij} \leftarrow \dfrac{\mathtt{w}_{ij}}{\sum_{k=1}^{\mathtt{t}+\mathtt{u}} \mathtt{w}_{ik}} \; \forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; // initialize $\mathtt{P}$

$(\mathtt{Y}_\mathtt{L})_{\text{row } i} \leftarrow \delta_\ell(\mathtt{y}_i) \; \forall i \in \{1, \ldots, \mathtt{t}\}$; // initialize $\mathtt{Y}_\mathtt{L}$

$\mathtt{f}_\mathtt{U} \leftarrow 0$;

repeat

  $i \leftarrow$ random integer in $\{1, \ldots, \mathtt{u}\}$;

  $j \leftarrow$ random integer in $\{1, \ldots, \ell\}$;

  $(\mathtt{f}_\mathtt{U})_{ij} \leftarrow (\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik}(\mathtt{f}_\mathtt{U})_{kj}$;

until $\max_{\substack{i\in\{1,\ldots,\mathtt{u}\}\\ j\in\{1,\ldots,\ell\}}} \left|(\mathtt{P}_{\mathtt{UU}}\mathtt{f}_\mathtt{U} + \mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij} - (\mathtt{f}_\mathtt{U})_{ij}\right| \leq \tau$;

Algorithm 2 is of no practical use because it is very inefficient: it performs only one update and then a full test for the harmonic property for $\mathtt{f}_\mathtt{U}$ for every iteration. Moreover, one update does not guarantee progress because it is possible that the particular element chosen did already satisfy the harmonic condition $(\mathtt{f}_\mathtt{U})_{ij} = (\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik}(\mathtt{f}_\mathtt{U})_{kj}$ before the update. The purpose of Algorithm 2 is solely to demonstrate that element updates can be performed in truly any order. We now prove a lemma that is needed for the proof of convergence.

Lemma 5.2.1. *If $\mathtt{W}$ and $\mathtt{Y}$ allow a harmonic matrix $\mathtt{f}_\mathtt{U}^\infty$, then at the beginning of any iteration of Algorithm 2, $(\mathtt{f}_\mathtt{U})_{ij} \leq (\mathtt{f}_\mathtt{U}^\infty)_{ij} \; \forall i \in \{1, \ldots, \mathtt{u}\}, j \in \{1, \ldots, \ell\}$.*

Proof. By induction on iteration steps.

Base: Matrix $\mathtt{f}_\mathtt{U}$ starts at zero. The first update sets the slot to $(\mathtt{f}_\mathtt{U})_{ij}$ to $(\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij}$. Then $(\mathtt{f}_\mathtt{U})_{ij} \leq (\mathtt{f}_\mathtt{U}^\infty)_{ij}$ because $\mathtt{f}_\mathtt{U}^\infty$ is the sum of $\mathtt{P}_{\mathtt{UU}}\mathtt{f}_\mathtt{U}^\infty$ and $\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L}$, and $\mathtt{P}_{\mathtt{UU}}\mathtt{f}_\mathtt{U}^\infty$ has only nonnegative elements.

Inductive step: By the induction hypothesis, before the update, $(\mathtt{f}_\mathtt{U})_{ij} \leq (\mathtt{f}_\mathtt{U}^\infty)_{ij} \; \forall i \in \{1, \ldots, \mathtt{u}\}, j \in \{1, \ldots, \ell\}$. Therefore

$$(\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik}(\mathtt{f}_\mathtt{U})_{kj} \leq (\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik}(\mathtt{f}_\mathtt{U}^\infty)_{kj} \tag{5.5}$$

as weighted average with positive coefficients in $\mathtt{P}_{\mathtt{UU}}$. But the right-hand side term is equal to $(\mathtt{f}_\mathtt{U}^\infty)_{ij}$, so the new value is less than or equal to $(\mathtt{f}_\mathtt{U}^\infty)_{ij}$, which concludes the proof. □

Lemma 5.2.2. *At the beginning of each iteration of Algorithm 2, the following condition is satisfied* $\forall i \in \{1,\dots,\mathtt{u}\}, j \in \{1,\dots,\ell\}$*:*

$$(\mathtt{f_U})_{ij} \leq (\mathtt{P_{UL}Y_L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik}(\mathtt{f_U})_{kj} \tag{5.6}$$

Proof. By induction on step t.

Base: Before the first step, $(\mathtt{f_U})_{ij} = 0 \leq (\mathtt{P_{UL}Y_L})_{ij}$.

Inductive step: During the $(t+1)^{\text{th}}$ step, all elements are not updated (thus vacuously satisfying monotonicity), except one, call it $(\mathtt{f_U})_{ij}$. Taking the difference between the values before and after step $t+1$ yields:

$$\left(\mathtt{f_U}^{\text{step } t+1}\right)_{ij} - \left(\mathtt{f_U}^{\text{step } t}\right)_{ij} = (\mathtt{P_{UL}Y_L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik} \left(\mathtt{f_U}^{\text{step } t}\right)_{kj} - \left(\mathtt{f_U}^{\text{step } t}\right)_{ij} \tag{5.7}$$

By the induction hypothesis, $\left(\mathtt{f_U}^{\text{step } t+1}\right)_{ij} - \left(\mathtt{f_U}^{\text{step } t}\right)_{ij} \geq 0$. □

As a direct consequence of this lemma, elements of $\mathtt{f_U}$ increase monotonically throughout iterations of Algorithm 2. We have shown that $\mathtt{f_U}$ has monotonically increasing elements and has $\mathtt{f_U}^{\infty}$ as an upper bound, so by the monotone convergence theorem there exists a matrix $\mathtt{f_U}^{*} = \lim_{t\to\infty} \mathtt{f_U}$. It is necessary to prove that $\mathtt{f_U}^{*} = \mathtt{f_U}^{\infty}$, as $\mathtt{f_U}$ may stop updating, leaving Algorithm 2 iterating *ad infinitum*. Therefore we provide the following theorem. It is different from the proof of convergence of classic label propagation [237] and from Theorem 2.3.2 by randomly improving one element of the $\mathtt{f_U}$ instead of the entire $\mathtt{f_U}$.

Theorem 5.2.3. *If the random selection of* i *and* j *in Algorithm 2 reaches every element in* $\{1,\dots,u\}$ *and* $\{1,\dots,\ell\}$ *respectively with probability greater than a constant* $p > 0$*, then Algorithm 2 converges in the same conditions and to the same solution as Algorithm 1.*

Proof. Given $p > 0$, updating any given element in $\mathtt{f}$ is a binomial stochastic process that updates each element with probability approaching 1 for $t \to \infty$. By the two previous lemmas, elements in $\mathtt{f_U}$ are monotonically increasing and bounded, so there exists $\mathtt{f_U}^{*} = \lim_{t\to\infty} \mathtt{f_U}^{\text{step } t}$. That matrix $\mathtt{f_U}^{*}$ satisfies $(\mathtt{f_U}^{*})_{ij} = (\mathtt{P_{UL}Y_L})_{ij} + \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik}(\mathtt{f}^{*}_{U})_{kj}$ $\forall i \in \{1,\dots,\mathtt{u}\}, j \in \{1,\dots,\ell\}$, which is easily recognized as the element-wise form of the matrix relation $\mathtt{f_U}^{*} = \mathtt{P_{UL}Y_L} + (\mathtt{P_{UU}})_{ik}\mathtt{f_U}^{*}$. But there is only one harmonic function satisfying the relation for $\mathtt{W}$ and $\mathtt{Y}$, so $\mathtt{f}^{*} = \mathtt{f}^{\infty}$. □

5.3 Applications of Stochastic Label Propagation

Theorem 5.2.3 is very powerful because it offers an algorithm the freedom to update the elements of $\mathtt{f_U}$ under absolutely any schedule, without any ordering restriction. Moreover, and most importantly, convergence may also be faster than in the classic iterative label propagation because a new value $(\mathtt{f_U})_{ij}$ that is closer to the desired result is used immediately, as soon as it is computed, as opposed to the epoch-oriented approach in which a whole set of updates is computed on the side

in one iteration to be used in the next. (However, memory hierarchy effects might make such an implementation potentially slower if the order in which $\mathtt{f_U}$ is spannedis cache-unfriendly.) Most interestingly, Theorem 5.2.3 allows updates of $\mathtt{f_U}$ performed by concurrent processes operating on different row sections of $\mathtt{P}$. The fact that convergence is unaffected by the order of updating implies that the algorithm is tolerant to out-of-order memory updates and benign races, as long as each individual update of a floating-point number is atomic.

The following two subsections propose two applications of Theorem 5.2.3 by introducing two distinct algorithms, one serial, one parallel.

5.3.1 In-Place Label Propagation

The idea behind in-place label propagation (Algorithm 3) is to do a classic matrix multiplication (just as in the original iteration formula), but instead of computing a new matrix $\mathtt{f}'$ from $\mathtt{f}$, simply reassign each element back to $\mathtt{f}$ as soon as it is computed.

Algorithm 3: In-Place Label Propagation

Input: Labels $\mathtt{Y}$; similarity matrix $\mathtt{W} \in \mathbb{R}_+^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$ with $\mathtt{w}_{ij} = \mathtt{w}_{ji} \geq 0$ $\forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; tolerance $\tau > 0$.

Output: Matrix $\mathtt{f_U} \in [0,1]^{\mathtt{u}\times\ell}$ containing unnormalized probability distributions over labels.

1 $\mathtt{w}_{ii} \leftarrow 0 \; \forall i \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$;

$\mathtt{p}_{ij} \leftarrow \dfrac{\mathtt{w}_{ij}}{\sum_{k=1}^{\mathtt{t}+\mathtt{u}} \mathtt{w}_{ik}} \; \forall i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; // initialize $\mathtt{P}$

2

3 $(\mathtt{Y_L})_{\text{row } i} \leftarrow \delta_\ell(\mathtt{Y}_i) \; \forall i \in \{1, \ldots, l\}$;

4 $\mathtt{f_U} \leftarrow 0$;

5 **repeat**

6 $\quad \mathtt{f}'_\mathtt{U} \leftarrow \mathtt{f_U}$;

7 $\quad$ **for** $i \in \langle\langle 1, \ldots, \mathtt{u} \rangle\rangle$ **do**

8 $\quad\quad$ **for** $j \in \langle\langle 1, \ldots, \ell \rangle\rangle$ **do**

9 $\quad\quad\quad (\mathtt{f_U})_{ij} \leftarrow \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik}(\mathtt{f_U})_{kj} + (\mathtt{P_{UL}}\mathtt{Y_L})_{ij}$;

10 $\quad\quad$ **end**

11 $\quad$ **end**

12 **until** $\tau \geq \max_{j\in\{1,\ldots,\ell\}} \max_{i\in\{1,\ldots,\mathtt{u}\}} \left(\mathtt{f_U} - \mathtt{f}'_\mathtt{U}\right)_{ij}$;

Each epoch (i.e., a full pass through the outermost **repeat** loop) first stores a copy of $\mathtt{f_U}$ in $\mathtt{f}'_\mathtt{U}$ and then spans $\mathtt{f_U}$ one element at a time. Each innermost loop iteration updates one element in $\mathtt{f_U}$. Algorithm 3 is similar to Algorithm 1 (with the matrix operation made explicit element-wise), with one crucial difference. In Algorithm 1, a new estimate for $\mathtt{f_U}$ was computed *on the side* to then replace $\mathtt{f_U}$ for the next epoch. Algorithm 3, in contrast, computes new values for $\mathtt{f_U}$ are computed *in place* and available immediately for subsequent computations within the same epoch. For example,

the better estimate for $\mathtt{f}_\mathtt{U}$ values at row 1 are used to compute values at row 2. By the end of the epoch, updates in row $\mathtt{u}$ benefit of cumulative updates in all other rows. In contrast, at the end of an epoch of Algorithm 1 still are updated with values computed in the previous epoch. This makes Algorithm 3 converge faster than Algorithm 1, while still correct because of Theorem 5.2.3.

In-place updates put the termination condition under scrutiny. It would appear that computing the maximum difference between elements of $\mathtt{f}_\mathtt{U}$ and $\mathtt{f}'_\mathtt{U}$ would be an insufficient condition because it could terminate the algorithm too early due to a subtle effect. Updates committed early in one epoch are available for immediate use; therefore, later rows benefit of better approximations than earlier rows. Conversely, at the end of any epoch, it is possible that elements in the first row are at a much larger error than elements in the last row. For example, consider that the update made to some column c in the first row, $(\mathtt{f}_\mathtt{U})_{1c}$ was deemed correct. But after that, in the worst case, $(\mathtt{f}_\mathtt{U})_{2c}$, $(\mathtt{f}_\mathtt{U})_{3c}$, …, $(\mathtt{f}_\mathtt{U})_{\mathtt{u}c}$ also got updated, each within the maximum tolerance as well. Each of these updates take $(\mathtt{f}_\mathtt{U})_{1c}$ further away from meeting the harmonic condition, however the algorithm may be "fooled" into considering $(\mathtt{f}_\mathtt{U})_{1c}$ correct and terminate early with a large error at that position. The following theorem shows that with its termination condition, Algorithm 3 does compute the correct solution within tolerance τ.

Theorem 5.3.1. *Algorithm 3 terminates and computes the harmonic function $\mathtt{f}_\mathtt{U}^\infty$ within tolerance τ.*

Proof. Termination results as a consequence of Theorem 5.2.3. The updates performed by Algorithm 3 converge to the harmonic function, and after sufficiently many steps, the difference $\mathtt{f}_\mathtt{U} - \mathtt{f}'_\mathtt{U}$ drops below any constant value.

To prove correctness, we are interested in the state of $\mathtt{f}_\mathtt{U}$ after the last iteration, and particularly the way each element is influenced by elements changed after it. (If there was no influence, then $\mathtt{f}_\mathtt{U}$ would satisfy the charge.) Each element $(\mathtt{f}_\mathtt{U})_{ij}$ is affected by changes to $(\mathtt{f}_\mathtt{U})_{(i+1)j}$, $(\mathtt{f}_\mathtt{U})_{(i+2)j}$, …, $(\mathtt{f}_\mathtt{U})_{\mathtt{u}j}$. All changes are positive by the monotonicity theorem. But there is one difficulty—even if $(\mathtt{f}_\mathtt{U})_{ij} - (\mathtt{f}_\mathtt{U})'_{ij}$ is small indicating a small distance from the solution, that difference could be increased by subsequent changes to the lower rows. We need to compute the deviation from the harmonic condition. The harmonic value for $(\mathtt{f}_\mathtt{U})_{ij}$ at the end of the epoch is

$$h_{ij} = \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} (\mathtt{f}_\mathtt{U})_{kj} + (\mathtt{P}_{\mathtt{UL}} \mathtt{Y}_\mathtt{L})_{ij} \tag{5.8}$$

The actual value computed for $(\mathtt{f}_\mathtt{U})_{ij}$ during the epoch is

$$(\mathtt{f}_\mathtt{U})_{ij} = \sum_{k=1}^{i} (\mathtt{P}_{\mathtt{UU}})_{ik} (\mathtt{f}_\mathtt{U})_{kj} + \sum_{k=i+1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} (\mathtt{f}'_\mathtt{U})_{kj} + (\mathtt{P}_{\mathtt{UL}} \mathtt{Y}_\mathtt{L})_{ij} \tag{5.9}$$

form that clarifies that some updates were done with the old values copied in $\mathtt{f}'_\mathtt{U}$ and some updates were done with already-updated values. To compute how far $(\mathtt{f}_\mathtt{U})_{ij}$ is from the harmonic condition

at the end of the epoch, we take the difference $h_{ij} - (\mathtt{f}_{\mathtt{U}})_{ij}$, obtaining

$$h_{ij} - (\mathtt{f}_{\mathtt{U}})_{ij} = \sum_{k=i+1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{U}\mathtt{U}})_{ik}(\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}})_{kj} \tag{5.10}$$

$$\leq \max_{k\in\{i+1,\ldots,\mathtt{u}\}} (\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}})_{kj} \sum_{k=i+1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{U}\mathtt{U}})_{ik} \tag{5.11}$$

$$\leq \max_{k\in\{i+1,\ldots,\mathtt{u}\}} (\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}})_{kj} \tag{5.12}$$

$$\leq \max_{k\in\{1,\ldots,\mathtt{u}\}} (\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}})_{kj} \tag{5.13}$$

$$\leq \max_{j\in\{1,\ldots,\ell\}} \max_{k\in\{1,\ldots,\mathtt{u}\}} (\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}})_{kj} \tag{5.14}$$

The last form is exactly the termination condition, so when the algorithm stops, all elements are within τ of the harmonic condition. □

The fact that only the maximum of $\mathtt{f}_{\mathtt{U}} - \mathtt{f}'_{\mathtt{U}}$ is needed allows us to compute the solution without storing $\mathtt{f}'_{\mathtt{U}}$ at all, only the running maximum. This leads us to Algorithm 4, which does not require $\mathtt{f}'_{\mathtt{U}}$ anymore. (Also $\mathtt{P}$ can be easily computed as an in-place replacement over $\mathtt{W}$, detail we left out of Algorithm 4.) Such an implementation is important in environments where extra memory allocation is either not desirable or not possible. Also, on many contemporary architectures, a smaller working set often translates in faster speeds for comparable computational load. Plus, when we scale to multiple processors in the next section, it will be a notable advantage that each processor does not need extra private memory.

Our practical experiments use Algorithm 4 as the basis for implementation.

5.3.2 Multicore Label Propagation

The most interesting practical consequence of Theorem 5.2.3 is that label propagation can be parallelized easily and with low overhead on today's processing architectures.

Classic label propagation can be easily parallelized to run on one processor for each of the ℓ labels. This is because computations of different columns in $\mathtt{f}_{\mathtt{U}}$ are independent from one another (the optional row-normalization must only be done at the end of convergence). This, however, is not true scalability; most application have a small fixed ℓ and a large variable $\mathtt{u}$, so scaling up should be performed by finding a way to divide work across $\mathtt{u}$.

Theorem 5.2.3 and Algorithm 4 do allow scalability over $\mathtt{u}$. If a system has k processors, each processor computes in-place $\frac{\mathtt{u}}{k}$ rows of a shared matrix $\mathtt{f}_{\mathtt{U}}$. The crucial aspect that pertains to scalability is that writes to elements of $\mathtt{f}_{\mathtt{U}}$, although they do engender race conditions (because a value written by one process is read by all others), do not need to be synchronized at all per Theorem 5.2.3 if we assume that each write is atomic. Also, each processor reads data written by others and writes data never written by others, so there are no write-after-write conflicts that could cause wasted computation. But an issue of wasted computation still exists. Although newly computed values are never lost (there is guaranteed overall progress), computation power is spent on recomputing the same value, assuming there are no updates to a particular column.

Algorithm 4: Memory-Economic In-Place Label Propagation

Input: Labels $\mathtt{Y}$; similarity matrix $\mathtt{W} \in \mathbb{R}_+^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$ with $\mathtt{w}_{ij} = \mathtt{w}_{ji} \geq 0$ $\forall\, i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; tolerance $\tau > 0$.

Output: Matrix $\mathtt{f}_\mathtt{U} \in [0,1]^{\mathtt{u}\times\ell}$ containing unnormalized probability distributions over labels.

$\mathtt{w}_{ii} \leftarrow 0\ \forall\, i \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$;

$\mathtt{p}_{ij} \leftarrow \dfrac{\mathtt{w}_{ij}}{\sum_{k=1}^{\mathtt{t}+\mathtt{u}} \mathtt{w}_{ik}}\ \forall\, i, j \in \{1, \ldots, \mathtt{t}+\mathtt{u}\}$; // initialize $\mathtt{P}$

$(\mathtt{Y}_\mathtt{L})_{\text{row } i} \leftarrow \delta_c(\mathtt{y}_i)\ \forall\, i \in \{1, \ldots, \mathtt{t}\}$;

$\mathtt{f}_\mathtt{U} \leftarrow 0$;

repeat

  $\tau_m = 0$;

  for $i \in \langle\!\langle 1, \ldots, \mathtt{u} \rangle\!\rangle$ **do**

    for $j \in \langle\!\langle 1, \ldots, \ell \rangle\!\rangle$ **do**

      $a \leftarrow (\mathtt{f}_\mathtt{U})_{ij}$;

      $(\mathtt{f}_\mathtt{U})_{ij} \leftarrow \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik}(\mathtt{f}_\mathtt{U})_{kj} + (\mathtt{P}_{\mathtt{UL}}\mathtt{Y}_\mathtt{L})_{ij}$;

      $\tau_m \leftarrow \max(\tau_m, (\mathtt{f}_\mathtt{U})_{ij} - a)$;

    end

  end

until $\tau \geq \tau_m$;

With regard to atomicity of writes, on a 32-bit system, a parallel implementation necessitates that each floating-point value is written and read atomically. A single-precision IEEE 754 value is written atomically, whereas on a 64-bit system, a double-precision IEEE 754 is written atomically. (Most 32-bit systems allow atomic 64-bit writes through special processor instructions.) A 32-bit system that needs to perform 64-bit computations can combine Algorithm 4 with Algorithm 1 to perform computations on the side (in private memory). A rendez-vous mechanism at the end of each epoch synchronizes over all processors and commits batched changes in bursts, thus factoring interlocking costs over many writes. A different approach to 64-bit computation on 32-bit machines is to first run a parallel algorithm on 32-bit floating point numbers. The result of this algorithm can be converted to 64-bit numbers and used as the initial values for the serial version of the algorithm. Since the 32-bit result is a close approximation of the harmonic function, the serial part of the algorithm will converge rapidly.

Ironically, although unsynchronized writes are beneficial, they also raise the problem of termination detection: since all processes are independent, there is no coordination and therefore no chance to tell the processors when to stop. Therefore, a minimum amount of coordination must be added to stop when the harmonic function has been computed within a given tolerance. Each epoch, every process must check on a shared "continue" Boolean variable that informs the process whether it should continue or terminate. A separate process runs independently of all others, just checking whether $\mathtt{f}_\mathtt{U}$ satisfies the harmonic property within tolerance τ. Once that happens, the separate

process sets the "continue" shared variable to false, and all threads terminate. The monotonicity theorem ensures that any extra work done after $\mathtt{f}_\mathtt{U}$ has passed the harmonic test with tolerance τ will only improve the solution.

An important advantage of a parallel application of Algorithm 3 manifests itself on *relaxed memory models*,[1] which, at the time of this writing, dominate the multiprocessor desktop computing market. In a relaxed memory model, updates to shared memory performed by one processor may not be seen by other processors in the same order as they are written. An important consequence of Theorem 5.2.3 is that out-of-order reads and writes do not affect convergence. Also, on certain multiprocessor machines, special instructions must be issued at least once an epoch to make sure data is (a) committed to shared memory, and (b) re-loaded from shared memory. Otherwise, updates or some of the updates may only be written to and/or read from local, processor-private cache memory. If such instructions are only executed once per epoch, the overhead incurred by synchronization is negligible. Although this chapter does not aim at devising machine-specific algorithms, we do want to convey that Algorithm 3 is directly convertible into scalable parallel implementations of label propagation on a variety of processor architectures.

5.4 Reducing the Number of Labeled Nodes in the Graph

Existing work [65] reduces the number of nodes in the graph by using a subset selection method, at the expense of precision. The algorithm proposed below reduces the number of labeled nodes from $\mathtt{t}$ to ℓ without impacting in any way the precision of the classification. Concrete applications usually have much fewer distinct labels (e.g., a handful up to a few thousand) than labeled samples (thousands to billions), so the reduction—often on the scale of many orders of magnitude—is highly beneficial. It is always safe to assume that $\mathtt{t} \geq \ell$; if that is not the case, then there exist out-of-sample labels. Given that they are never hypothesized, the out-of-sample labels can be simply eliminated during a preprocessing step.

The intuition behind the reduction process is that labeled nodes having the same label can be "collapsed" together because their identity does not matter.

Lemma 5.4.1. *Consider the matrices* $\mathtt{P} \in [0,1]^{(\mathtt{t}+\mathtt{u})\times(\mathtt{t}+\mathtt{u})}$ *and* $\mathtt{f} \in [0,1]^{(\mathtt{t}+\mathtt{u})\times\ell}$ *initialized for the label propagation algorithm. Define the matrix* $R(a,b) \in [0,1]^{(\mathtt{t}+\mathtt{u}-1)\times(\mathtt{t}+\mathtt{u}-1)}$*, where* $1 \leq a < b \leq \mathtt{t}$*, obtained from* $\mathtt{P}$ *by adding the* a^{th} *column to the* b^{th} *column, followed by the elimination of the* a^{th} *row and* a^{th} *column:*

$$R(a,b) = \begin{bmatrix} \mathtt{p}_{1,1} & \cdots & \mathtt{p}_{1,a-1} & \mathtt{p}_{1,a+1} & \cdots & \mathtt{p}_{1,a}+\mathtt{p}_{1,b} & \cdots & \mathtt{p}_{1,n} \\ \cdots & & & & & & & \\ \mathtt{p}_{a-1,1} & \cdots & \mathtt{p}_{a-1,a-1} & \mathtt{p}_{a-1,a+1} & \cdots & \mathtt{p}_{a-1,a}+\mathtt{p}_{a-1,b} & \cdots & \mathtt{p}_{a-1,n} \\ \mathtt{p}_{a+1,1} & \cdots & \mathtt{p}_{a+1,a-1} & \mathtt{p}_{a+1,a+1} & \cdots & \mathtt{p}_{a+1,a}+\mathtt{p}_{a+1,b} & \cdots & \mathtt{p}_{a+1,n} \\ \cdots & & & & & & & \\ \mathtt{p}_{n,1} & \cdots & \mathtt{p}_{n,a-1} & \mathtt{p}_{n,a+1} & \cdots & \mathtt{p}_{n,a}+\mathtt{p}_{n,b} & \cdots & \mathtt{p}_{n,n} \end{bmatrix} \tag{5.15}$$

[1] Not to be confused with the method of relaxations; the two terms are unrelated.

Also define the matrix $g(a)$ obtained by eliminating the a^{th} row of $\mathtt{f}$:

$$g(a) = \begin{bmatrix} \mathtt{f}_{11} & \mathtt{f}_{12} & \dots & \mathtt{f}_{1\ell} \\ \dots & & & \\ \mathtt{f}_{(a-1)1} & \mathtt{f}_{(a-1)2} & \dots & \mathtt{f}_{(a-1)\ell} \\ \mathtt{f}_{(a+1)1} & \mathtt{f}_{(a+1)2} & \dots & \mathtt{f}_{(a+1)\ell} \\ \dots & & & \\ \mathtt{f}_{\mathtt{t}+\mathtt{u}1} & \dots & & \mathtt{f}_{\mathtt{t}+\mathtt{u}\ell} \end{bmatrix} \tag{5.16}$$

If the rows a and b of $\mathtt{f}$ are identical, then using $R(a,b)$ and $g(a)$ for the label propagation algorithm yields the same label predictions for the unlabeled data (the bottom $\mathtt{u}$ rows of $g(a)$, which we denote as $g(a)_{\mathtt{U}}$) as the predictions $\mathtt{f}_{\mathtt{U}}$ obtained by using $\mathtt{P}$ and $\mathtt{f}$.

Proof. Consider the iterative step $\mathtt{f}' \leftarrow \mathtt{P}\mathtt{f}$. The element $\mathtt{f}'_{kj}$ is:

$$\mathtt{f}'_{kj} = \sum_{i=1}^{\mathtt{t}+\mathtt{u}} \mathtt{p}_{ki}\mathtt{f}_{ij} = \mathtt{p}_{ka}\mathtt{f}_{aj} + \mathtt{p}_{kb}\mathtt{f}_{bj} + \sum_{\substack{i=1 \\ i \notin \{a,b\}}}^{\mathtt{t}+\mathtt{u}} \mathtt{p}_{ki}\mathtt{f}_{ij} \tag{5.17}$$

But $\mathtt{f}_{aj} = \mathtt{f}_{bj}$ by the hypothesis, therefore:

$$\mathtt{f}'_{kj} = (\mathtt{p}_{ka} + \mathtt{p}_{kb})\mathtt{f}_{bj} + \sum_{\substack{i=1 \\ i \notin \{a,b\}}}^{\mathtt{t}+\mathtt{u}} \mathtt{p}_{ki}\mathtt{f}_{ij} \tag{5.18}$$

It can be easily verified by inspection that:

$$\mathtt{f}'_{kj} = \begin{cases} g'(a)_{kj} & \text{if } a \in \{1, \dots, k\} \\ g'(a)_{(k-1)j} & \text{if } a \in \{k+1, \dots, \mathtt{t}+\mathtt{u}\} \end{cases} \forall\, j \in \{1, \dots, \ell\} \tag{5.19}$$

where $g'(a) = R(a,b) \cdot g(a)$. Given that $a < \mathtt{t}$ (by the hypothesis), it follows that $\mathtt{f}'$ and $g'(a)$ contain the same values in their bottom $\mathtt{u}+1$ rows. (The top rows are clamped and do not participate in the result.) So one step preserves the intermediate result.

By induction over the steps of the iteration, it follows that both iterations converge and after convergence, $R(a,b)$ and $g(a)$ will yield identical label predictions as $\mathtt{P}$ and $\mathtt{f}$. □

This means that the graph for Zhu's label propagation can be reduced by one labeled sample whenever there are two labeled samples having he same label. Applying the reduction process iteratively, we obtain the following theorem.

Theorem 5.4.2 (Graph Reduction)**.** *Consider a graph with $\mathtt{t}$ labeled points (accounting for ℓ labels) and $\mathtt{u}$ unlabeled points, as constructed for the label propagation algorithm. If all labeled nodes for each given label are collapsed together and the resulting parallel edges are linearly superposed (reduced to one edge by summing their weights), then the resulting graph with ℓ labeled nodes yields the same label predictions for the unlabeled data as the original graph.*

Proof. The preconditions of Lemma 5.4.1 are respected as long as there are at least two labeled nodes with the same label. This means we can apply one reduction step until only ℓ labeled node remain. During the reduction, by Lemma 5.4.1, the label propagation results are preserved. □

After reduction has been effected, a reduced matrix $R_{\min} \in [0,1]^{(\ell+\mathtt{u})\times(\ell+\mathtt{u})}$ and a reduced label matrix $g_{\min} \in [0,1]^{(\ell+\mathtt{u})\times\ell}$, an important reduction in size. The process of reduction requires only $\mathcal{O}\left(\mathtt{t}+\mathtt{u}(\mathtt{t}+\mathtt{u}-\ell)\right)$ additions and does not require additional memory. The required memory for an exact implementation is reduced from $\mathcal{O}\left((\mathtt{u}+\mathtt{t})^2\right)$ to $\mathcal{O}\left((\mathtt{u}+\ell)^2\right)$.

5.5 Graph Reduction for Structured Inputs and Outputs

We have shown in Theorem 5.4.2 that all training vertices carrying the same label can be collapsed into one if the resulting parallel edges are summed. We apply that result to the graph built for learning with structured inputs and outputs introduced in Chapter 4, Definition 4.2.2. In that graph, all source vertices can be collapsed into one source vertex, and all sink vertices can be similarly collapsed into one sink vertex. The resulting graph has only one source and one sink, as per the definition below, which is a refinement of Definition 4.2.2 where graph reduction has been implicitly carried.

Definition 5.5.1 (Graph-Based Formulation of Structured Learning with Only Positive Training Samples with Graph Reduction)**.** Consider a structured learning problem defined by features $\mathtt{X} = \langle\!\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\!\rangle \subseteq \mathcal{X}$, training labels $\mathtt{Y} = \langle\!\langle \mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{t}} \rangle\!\rangle \subseteq \mathcal{Y}$, similarity function $\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [0,1]$, and hypothesis generator function $\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y})$. A similarity graph for the structured learning problem is an undirected weighted graph with real-valued vertex labels, constructed as follows:

- add one distinguished vertex v_+ with label 1;
- add one distinguished vertex v_- with label 0;
- add one vertex v_{ij} (with initial label 0) for each hypothesis $(\chi(\mathtt{x}_i))_j$, $j \in \langle\!\langle 1, \ldots, \mathrm{card}(\chi(\mathtt{x}_i)) \rangle\!\rangle$, of each test sample $i \in \langle\!\langle \mathtt{t}+1, \ldots, \mathtt{t}+\mathtt{u} \rangle\!\rangle$;
- for each vertex v_{ij}, define one edge linking it to v_+ and one linking it to v_-, with the respective weights

$$\mathtt{w}_{ij+} = \sum_{k=1}^{\mathtt{t}} \sigma\left(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle , \langle\!\langle \mathtt{x}_k, \mathtt{y}_k \rangle\!\rangle \right) \tag{5.20}$$

$$\mathtt{w}_{ij-} = C_{ij} - \mathtt{w}_{ij+} \tag{5.21}$$

- for each pair of vertices v_{ij} and v_{kl}, define an edge linking them with weight

$$\mathtt{w}_{ijkl} = \sigma\left(\langle\!\langle \mathtt{x}_i, (\chi(\mathtt{x}_i))_j \rangle\!\rangle , \langle\!\langle \mathtt{x}_k, (\chi(\mathtt{x}_k))_l \rangle\!\rangle \right) \tag{5.22}$$

By Theorem 5.4.2, this graph computes the same scores as the much larger graph in Definition 4.2.2, a result that may seem counterintuitive. The ability to collapse together train sentences stems from all train sentences having the same score, therefore their identity does not matter: the identity of similar training sentences is not relevant; what matters for the assignment of scores to the test hypotheses is their global similarity with the training set, or, more precisely, their average similarity with their entire neighborhood of labeled points considered as a whole. A graph could be set up such that individual training sentences, or categories thereof, are meaningful to the approach (e.g. when confidence information is associated with each training sample).

5.6 Fast Graph Construction in Jensen-Shannon Space

We now turn our attention to computing distances in the real-valued multidimensional spaces discussed in Chapter 3. Recall that in many HLT applications, the inputs are a mix of categorical, Boolean, and real-valued features. The data-driven construction process with two passes presented in § 3.3 uses a first-pass classifier to convert the often heterogeneous input features to probability distributions. The main merit of this setup is that it provides the graph-based algorithm with features that are amenable to good similarity definitions. Operating directly on the original heterogeneous features using a generic distance measure such as Euclidean (Eq. 3.3) or Cosine (Eq. 3.5) is arguably suboptimal. Our experiments in Chapter 3 have shown that, indeed, using such distances with label propagation yields inferior results in terms of accuracy when compared to the two-pass system.

Let us recap how similarities are computed in our two-pass system. An often-used approach that we also adopted is to define similarity as a Gaussian kernel over a distance measure $d : \mathtt{X} \times \mathtt{X} \rightarrow \mathbb{R}_+$ (recall § 3.2 and Eq. 3.2):

$$\sigma_\alpha : \mathtt{X} \times \mathtt{X} \rightarrow (0, 1] \quad \sigma_\alpha(\mathtt{x}_i, \mathtt{x}_j) = \exp\left[-\frac{d(\mathtt{x}_i, \mathtt{x}_j)^2}{\alpha^2}\right] \tag{5.23}$$

Computationally, this reformulates computing similarities into computing distances plus a constant per-pair amount of work. Also, the equation reveals that the similarity measure is more fine-grained for points close in space than for widely separated points; the function e^{-x^2} is rapidly decreasing, for example $\sigma_\alpha(\mathtt{x}_i, \mathtt{x}_j) \approx 10^{-7}$ for $d(\mathtt{x}_i, \mathtt{x}_j) = 4\alpha$.

Characterizing distances between probability distributions and the topologies that distances induce over distributions is a topic that has recently received increasing attention from not only machine learning researchers, but also statisticians. Usually distances between distributions are derived from probability *divergences*. To clarify a terminological detail that may sometimes cause confusion due to the different uses in literature:

- A *divergence* is a relation (usually defined over probability distributions) that indicates to what extent one sample "diverges" from another. Often, one of the samples is considered the reference. As such, divergence relations are not necessarily symmetric (commutative).

- A *metric* is a relation with the classic metric properties, i.e. (a) it is positive, (b) it is zero only for identical arguments, (c) it is symmetric, and (d) it satisfies the triangle inequality. Refer to § 3.4 on page 22 for formal definitions of these properties.

- A *distance* is a relation that measures the dissimilarity of elements in a set. It must (only) be positive and symmetric. Literature often uses the terms "distance" and "metric" interchangeably because distances of choice are often actually metrics, but since recently non-metric distances have received increasing attention [2], this work carefully distinguishes between the two.

As described in detail in § 3.4, there exist several divergence measures defined over probability distributions, of which Jensen-Shannon divergence (which is a symmetrized and bounded Kullback-Leibler divergence) was the one that was most successful in our experiments (§ 3.6, § 3.7) and will be the main focus of our examples. We repeat here the definitions of Kullback-Leibler divergence (Eq. 3.14) and Jensen-Shannon divergence (Eq. 3.18) for convenience:

$$d_{\mathsf{KL}}(\mathbf{z}, \mathbf{z}') = \sum_{i=1}^{\ell} \mathbf{z}_{[i]} \log \frac{\mathbf{z}_{[i]}}{\mathbf{z}'_{[i]}} \tag{5.24}$$

$$d_{\mathsf{JS}}(\mathbf{z}, \mathbf{z}') = \frac{1}{2}\left[d_{\mathsf{KL}}\left(\mathbf{z}, \frac{\mathbf{z}_{[i]} + \mathbf{z}'_{[i]}}{2}\right) + d_{\mathsf{KL}}\left(\mathbf{z}', \frac{\mathbf{z}_{[i]} + \mathbf{z}'_{[i]}}{2}\right)\right] \tag{5.25}$$

Jensen-Shannon divergence has been used in a variety of statistical analysis and machine learning tasks, such as testing the goodness-of-fit of point estimations [158], the analysis of DNA sequences [22, 191, 23], and image edge detection [94].

We will loosely refer to the space formed by probability distributions using Jensen-Shannon divergence for measuring distances as "Jensen-Shannon space." We are looking at finding the most similar items according to the similarity in Eq. 5.23 that operates on top of the Jensen-Shannon divergence d_{JS}. Given that σ_α is monotonically decreasing, finding the most similar samples is the same as finding the ones at the smallest d_{JS} from one another.

5.6.1 Nearest Neighbor Searching

A brute-force approach to creating the unlabeled-to-unlabeled edges (matrix $\mathtt{P_{UU}}$ introduced in Chapter 2) would entail computing the similarities over the cross-product $\langle\langle \mathtt{x_{t+1}}, \ldots, \mathtt{x_{t+u}} \rangle\rangle \times \langle\langle \mathtt{x_{t+1}}, \ldots, \mathtt{x_{t+u}} \rangle\rangle$, resulting in $\frac{\mathtt{u}(\mathtt{u}-1)}{2}$ evaluations of the similarity function. In addition, creating $\mathtt{P_{UL}}$ entails $\mathtt{u} \cdot \mathtt{t}$ similarity evaluations. For large values of $\mathtt{u}$ (and/or $\mathtt{t}$), exhaustive computation of all similarities is infeasible. Therefore, it is common to only include the k edges with the largest similarity values for each node in the graph, and to use fast methods for finding the k nearest neighbors. Edges with low weights can be ignored because they encode low-probability paths in the random walk. This is an instance of the nearest-neighbors problem.

Searching for the nearest neighbors has many applications in a variety of problems, such as information retrieval [194], storing and querying media databases [203, 79], data mining [21], and of course machine learning at large [101, 104].

For dimensionality $d \leq 2$ the problem of scalable nearest neighbors has been solved: There are known methods that complete a query in $\mathcal{O}(d \log n)$ with preprocessing taking $\mathcal{O}(d \cdot n)$ space and $\mathcal{O}(d \cdot n \log n)$ time. In one-dimensional spaces the approach is the well-known binary search on a sorted array, or constructing and using a search tree. In two-dimensional spaces the optimal algorithms are using Voronoi diagrams [10].

Voronoi introduced the eponymous diagram [10] in 1907–1908. A Voronoi diagram is a decomposition of a metric space M containing points of a set S in convex disjoint cells. Each point in $s \in S$ is associated with exactly one cell containing all points in M closer to s than to any other point in S. Consequently, cell boundaries (situated on perpendicular bisector hyperplanes) are at equal distance from two or more points in S, and the disjoint union of all cells cover the entire space M. Once a Voronoi diagram is built, finding the nearest neighbor of a given point is a matter of finding the cell to which the point belongs. Voronoi diagrams have been researched mostly in two [34, 227] or three [80] dimensions; in higher-dimensional spaces $d > 3$, storage requirements $\Theta(n^{\frac{d}{2}})$ make the approach impractical.

In 1967 Cover and Hart formally defined nearest-neighbor decision rule for classification [55]. This insight, combined with an increased interest in the theory and practice of machine learning, has prompted further research in the area, particularly in spaces with large number of dimensions.

For higher-dimensional spaces ($d > 2$) there is no known solution that is generally satisfactory. Kleinberg initiated the idea of providing theoretical bounds by putting restrictions on the distance measure [121]. Karger and Ruhl defined the all-important expansion rate of a sample set [114].

Approaches to nearest neighbor algorithms fall in several categories, including locality-sensitive hashing [91], walk-based techniques (such as the approximating eliminating search algorithm a.k.a. AESA [222, 160], Orchard's algorithm [177], Shapiro's algorithm [199]), and a large number of tree-based techniques. The latter algorithms organize data in a tree structure and search using a technique derived from the branch-and-bound general strategy. The goal is to organize the space such that large portions of it do not need to be searched. The most popular of these are kd-trees [19, 17, 18], metric trees [49], and Cover Trees [24]. These tree structures require $\mathcal{O}(n)$ storage space and practically require in-core presence of the entire training data set (for building the node-based trees). The build and query complexity of trees increase rapidly with c e.g. $\mathcal{O}(c^6 \log n)$ and $\mathcal{O}(c^{12} \log n)$ for cover trees.

Recent empirical comparisons against data obeying a variety of distributions suggest that these techniques generally yield little or negative improvement over kd-trees in build and query time [117]. We therefore chose kd-trees as our nearest-neighbors method of choice (implemented as an optimized library in the D programming language), but we emphasize that any nearest neighbor technique could be chosen for Jensen-Shannon space. In particular, given that Jensen-Shannon divergence is the square of a metric (the transmission metric [77, 37]), searching techniques that make use of the triangle inequality (e.g. AESA, metric trees or cover trees) can be used as long as $\sqrt{d_{\mathsf{JS}}}$ is used for searching instead of d_{JS}. (The end result is not affected because the square root function is monotonically increasing.) Our focus on kd-trees is motivated not only by their good empirical performance and lasting success, but also by their direct applicability to Jensen-Shannon spaces, as discussed below.

5.6.2 Using kd-trees in Jensen-Shannon Space

*K-D*imensional Trees (kd-trees) have been proposed by Bentley [19] and were first analyzed theoretically by Friedman, Bentley, and Finkel [85, 18, 16]. In spite of their age, kd-trees are a widely used and investigated data structure for performing fast nearest-neighbor searches. We will first describe kd-trees as originally proposed and as usually introduced in the literature [166], after which we will follow with considerations specific to using kd-trees with the Jensen-Shannon divergence.

A kd-tree built over a space embedded in $\mathbb{R}^K$ is a binary tree that stores, at each node ν, a finite collection of points $Z_\nu = \langle\langle \mathbf{z}_{\nu 1}, \ldots, \mathbf{z}_{\nu|Z_\nu|} \rangle\rangle \in \mathbb{R}^{K \times |Z_\nu|}$. Inner nodes also store a *cutting dimension* as a number $d_\nu \in \{1, \ldots, K\}$, a *cutting value* $c_\nu \in \mathbb{R}$, and left and right children nodes which we denote as $left(\nu)$ and $right(\nu)$. (Slight variations in the exact information stored are possible, as long as the fundamental information can be accessed efficiently.) There are two invariants governing a kd-tree:

1. If ν is the root node, it covers the entire point set:

$$Z_\nu = Z \tag{5.26}$$

2. If ν is a non-leaf node:

$$Z_{left(\nu)} \cup Z_{right(\nu)} = Z_\nu \tag{5.27}$$
$$Z_{left(\nu)} \cap Z_{right(\nu)} = \emptyset \tag{5.28}$$
$$\mathbf{z}_{[d_\nu]} \leq c_\nu \quad \forall\, \mathbf{z} \in Z_{left(\nu)} \tag{5.29}$$
$$\mathbf{z}_{[d_\nu]} \geq c_\nu \quad \forall\, \mathbf{z} \in Z_{right(\nu)} \tag{5.30}$$

Note how samples with $\mathbf{z}_{[d_\nu]} = c_\nu$ may fall in either the left or the right subtree. This simplifies certain tree building algorithms and their associated data structures, as discussed in the next section. Also, this ambiguity predicts that kd-trees have problems organizing highly clustered point sets: if many points have the same coordinate values, kd-tree structuring is unable to add information helping the search. (Restricting Eq. 5.29 or Eq. 5.30 to use strict inequality would not improve on this issue for reasons that will be clarified in § 5.6.2.2.)

Discrimination by comparison of dimension d_ν against value c_ν effectively introduces a cutting hyperplane orthogonal to the d_ν^{th} Cartesian axis at point distance c_ν from the origin. Points are placed in the left or right sub-tree depending on the side of the hyperplane they are on. (Points situated on the hyperplane may be placed in either subtree, but never both.) If we imagine the complete set Z as bounded by the smallest hyperrectangle that includes all of Z's points, a kd-tree organizes that hyperrectangle into smaller disjoint hyperrectangles.

Save for observing the invariants in Eq. 5.26–5.30, kd-tree building algorithms have discretion regarding the strategy of choosing the cutting dimension d_ν and the cutting value c_ν. We will discuss some popular tree building strategies below.

5.6.2.1 Building kd-trees

Implicit kd-trees The simplest building strategy is:

- Choose d_ν in a round robin fashion going down the tree: use $d_\nu = 1 + depth(\nu) \mod K$ for each node, i.e. the root splits at dimension 1, the root's children split at dimension 2, and so forth, resetting the counter whenever the depth reaches a multiple of K.

- Choose c_ν to be the median of the projections of Z_ν on dimension d_ν. For example, if $d_\nu = 5$, c_ν is the median of values $(\mathbf{z}_{\nu 1})_{[5]}, \ldots, (\mathbf{z}_{\nu|Z_\nu|})_{[5]}$.

Choosing the median as pivot always leads to a balanced tree of size $\mathcal{O}(|Z|)$ and depth $\mathcal{O}(\log |Z|)$ because each node has a roughly equal number of left and right children. However, balanced trees do not guarantee fast searching because, unlike in binary trees organizing one-dimensional number sets, the tree does not guarantee that search can always proceed in only one branch.

One advantage of the simplest strategy is that it can organize an array of points in-place, without requiring any additional storage. Such kd-trees in which the tree structure is implied by the navigation algorithm are called *implicit kd-trees* and are mainly used in three-dimensional modeling and virtual reality applications [224, 73]. The implicit structure is that for any given array, the root node covers the entire array and the left and right children cover each one half of the array. Applying this rule recursively through the trivial array of size 1 induces the implicit kd-tree. Algorithm 5 organizes an array into an implicit kd-tree.

Algorithm 5: IMPLICITKDTREE: Organizes an array into an implicit kd-tree.

Input: Array $Z = \langle\langle \mathbf{z}_1, \ldots, \mathbf{z}_{|Z|} \rangle\rangle \in \mathbb{R}^{K \times |Z|}$; splitting dimension $d \in \{1, \ldots, K\}$ (initial value arbitrary, e.g. 1).

Output: Z is organized as an implicit kd-tree.

```
1 if |Z| > 1 then
2     s ← ⌊|Z| / 2⌋;
3     partition(Z, s, d);
4     d ← 1 + d mod K;
5     IMPLICITKDTREE(Z[1..s], d);
6     IMPLICITKDTREE(Z[s + 1..|Z|], d);
7 end
```

The algorithm avails itself of the procedure $partition$ which is an array partitioning algorithm, for example the classic "Median of Medians" algorithm by Blum et al. [30] which runs in expected $\mathcal{O}(|Z|)$ time. This bound leads to a total $\mathcal{O}(|Z| \cdot \log |Z|)$ expected run time for IMPLICITKDTREE (taking into account that the recursion depth is always $\mathcal{O}(\log |Z|)$. The partitioning criterion is ordering comparison of projections on dimension d.

Implicit kd-trees are attractive in organizing large data sets with features homogeneously spread across all dimensions. If heavy clustering across specific dimensions occurs, implicit kd-trees are not very helpful because they partition data in a manner that does not take the data characteristics into account. Partial parallelization of the construction process is possible because after partitioning the two sub-arrays are entirely isolated from each other so there is no sharing contention between them.

Splitting Across the Largest-Spread Dimension An improvement to the strategy used by implicit kd-trees is to not choose the cutting dimension in a round-robin fashion, but instead use the dimension with the largest spread. This rule was proposed with the original kd-tree definition. A small amount of additional storage is needed in the form of an implicit tree (i.e., array) of splitting dimensions. The complexity of the building process remains the same. This rule still builds

a balanced tree, but the hyperrectangles that divide the space may be arbitrarily elongated. Such elongated shapes are adverse to the searching process because in the worst case two points may be deemed close (by virtue of being in the same box), yet may be arbitrarily far from each other by being situated at opposite ends of an elongated box.

Splitting Across the Midpoint A technique that always constructs hyperrectangles with small aspect ratios is to split across the midpoint of the longest side. This approach, however, may result in empty cells, i.e. hyperrectangles that contain no points at all. Therefore there is no bound on the depth of the tree or the number of nodes in it.

Hybrid Strategies Several strategies (including, but not limited to, the ones enumerated above) may be mixed and matched. For example, the popular Approximate Nearest Neighbor (ANN) library [8] defines construction strategies that e.g. attempt first to split across the midpoint and then move the midpoint such that no empty cells result. The best strategy to choose in each situation is, of course, dependent on the nature of the data set at hand.

5.6.2.2 Searching a kd-tree

The search process—as described by Friedman, Bentley, and Finkel [85]—first descends the tree recursively searching the point in the same (and smallest) bounding box as the query point $\mathbf{z}_q \in \mathbb{R}^K$. If the bounding boxes as created by the tree building algorithm are reasonably small in volume and aspect ratio, then a good approximation of the nearest neighbor has already been found. Keeping that point as a running candidate $\mathbf{z}_c$ (with the corresponding candidate distance $d(\mathbf{z}_q, \mathbf{z}_c) = r_c \in \mathbb{R}_+$), the algorithm climbs back the tree as it unwinds recursion. At each node climbed, if the intersection between the hypersphere centered at $\mathbf{z}_c$ of radius r_c and the other (as of yet unvisited) hyperrectangle of that node is non-null, the other child of the current node is also searched in a similar manner. The candidate $\mathbf{z}_c$ is replaced whenever a better candidate is found.

Two aspects are key to a good search performance. One is finding a good candidate in the descent phase with a small distance r_c. A good candidate eliminates the need for searching most or all unvisited subtrees in the unwinding phase, and leads to completing the search in logarithmic time. This aspect is dependent on the statistics of the data, the position of the query point relative to the point set, and the kd-tree build process. The other important aspect is ensuring that the intersection between the hypersphere centered at $\mathbf{z}_c$ of radius r_c and a hyperplane is cheaply computable. Achieving this goal depends on the characteristics of the distance function used.

Algorithm 6 (KDSEARCH) shows a definition of the search algorithm. It uses two subroutines that we will discuss in detail in short order. In brief, BOUNDSOVERLAPBALL$(R, \mathbf{z}, r)$ checks whether the rectangle R and the hypersphere (ball) centered in $\mathbf{z}$ with radius r have a non-null intersection. Second, the function BOUNDSENCLOSEBALL$(R, \mathbf{z}, r)$ checks whether the hypersphere centered in $\mathbf{z}$ with radius r is contained *entirely* within the hyperrectangle R.[2] The latter function is an optimization that is not present in some tutorial introductions to kd-trees [166], but was proposed alongside with the algorithm proposed by Friedman, Bentley, and Finkel [85]. To put Algorithm 6

[2]The subroutine was originally called "BALL-WITHIN-BOUNDS" [85], but we chose "BOUNDSENCLOSEBALL" because it has the same natural parameter order as "BOUNDSOVERLAPBALL."

in relation with that algorithm, the original definition used an augmented language instruction **done** to terminate recursion immediately; in contrast, Algorithm 6 adds a Boolean value c to the returned tuple and checks it after each recursive call. This makes the algorithm's definition marginally more complicated but also closer to a direct implementation.

Algorithm 6: KDSEARCH. Searching a kd-tree for the nearest neighbor of a query point [85].

Input: kd-tree T; bounding hyperrectangle $R = \infty$; query point $\mathbf{z}_q$; candidate distance $r_c = \infty$; candidate point $\mathbf{z}_c =$ *undefined*.
Output: Tuple of nearest point, smallest distance, and completion information $\langle\langle \mathbf{z}_n, r_n, c \rangle\rangle$.

if *isLeaf*(T) **then**
  $r \leftarrow \min_{\mathbf{z} \in T} d(\mathbf{z}_q, \mathbf{z})$;
  if $r < r_c$ **then**
    $\langle\langle \mathbf{z}_c, r_c \rangle\rangle \leftarrow \langle\langle \mathbf{z}_T, r \rangle\rangle$;
    if BOUNDSENCLOSEBALL$(R, \mathbf{z}_q, r_c)$ **then return** $\langle\langle \mathbf{z}_c, r_c, \textbf{true} \rangle\rangle$;
  end
else
  if $(\mathbf{z}_q)_{[d_T]} < c_T$ **then**
    $T' \leftarrow$ *left*(T);
    $T'' \leftarrow$ *right*(T);
    $R' \leftarrow$ *leftCut*(R, d_T, c_T);
    $R'' \leftarrow$ *rightCut*(R, d_T, c_T);
  else
    $T' \leftarrow$ *right*(T);
    $T'' \leftarrow$ *left*(T);
    $R' \leftarrow$ *rightCut*(R, d_T, c_T);
    $R'' \leftarrow$ *leftCut*(R, d_T, c_T);
  end
  $\langle\langle \mathbf{z}'_c, r'_c, c \rangle\rangle \leftarrow$ KDSEARCH$(T', R', \mathbf{z}_q, r_c)$;
  if c **then return** $\langle\langle \mathbf{z}'_c, r'_c, \textbf{true} \rangle\rangle$;
  if $r'_c < r_c$ **then**
    $\langle\langle \mathbf{z}_c, r_c \rangle\rangle \leftarrow \langle\langle \mathbf{z}'_c, r'_c \rangle\rangle$;
  end
  if BOUNDSOVERLAPBALL$(R'', \mathbf{z}_q, r_c)$ **then**
    $\langle\langle \mathbf{z}''_c, r''_c, c \rangle\rangle \leftarrow$ KDSEARCH$(T'', R'', \mathbf{z}_q, r_c)$;
    if c **then return** $\langle\langle \mathbf{z}''_c, r''_c, \textbf{true} \rangle\rangle$;
    if $r''_c < r_c$ **then**
      $\langle\langle \mathbf{z}_c, r_c \rangle\rangle \leftarrow \langle\langle \mathbf{z}''_c, r''_c \rangle\rangle$;
    end
  end
end
return $\langle\langle \mathbf{z}_c, r_c, \textbf{false} \rangle\rangle$;

Before discussing the subroutines BOUNDSOVERLAPBALL and BOUNDSENCLOSEBALL, let us note several refinements and improvements that can be effected in Algorithm 6. These include:

- *Using buckets:* Instead of storing exactly one point in each leaf, $b > 1$ points can be stored, thus making each leaf a bucket. Inside a bucket, brute-force search is used. Bucketing may save on tree allocation and navigation.

- *Using simplified distances:* A simple observation is that instead of the distance function, any monotonically-increasing function of the distance is allowed because it will yield the same nearest neighbors. This possibility can be used to reduce computational needs. For example, instead of computing Euclidean distance, the algorithm can operate on squared distances throughout; generalizing to Minkowski distances of order p, the algorithm can operate on distances raised to the power p.

- *Incremental distance calculation:* Arya and Mount [9] have proposed an ingenious technique to save computation when calculating the distance between the query point and the hyperrectangles R' and R''. Using their technique makes the complexity of BOUNDSOVERLAPBALL constant, whereas the canonical implementation of the function takes $\mathcal{O}(K)$ time. However, their improvement only applies to Minkowski distances, so it is not of interest to Jensen-Shannon divergence.

- *Searching for k nearest neighbors:* The algorithm can be readily adapted to find not only the closest neighbor, but the k nearest neighbors. This can be easily done by manipulating a priority list (e.g. binary heap) of tuples in lieu of the tuple $(\mathbf{z}_c, r_c)$. Whenever Algorithm 6 compares a potential replacement against the current best candidate, it must be changed to compare against the top of the heap (the worst match of the best k matches). If the potential replacement is better, it will replace the worst match in the heap. The complexity of the function grows by a factor of $\mathcal{O}(\log k)$, which is usually negligible.

5.6.2.3 Defining Core Routines. Distance Requirements

BOUNDSENCLOSEBALL and BOUNDSOVERLAPBALL form the core of the algorithm and also impose specific requirements on the distance measure. This section discusses these requirements and verifies that d_{JS} satisfies them.

An arbitrary distance function would lead to an arbitrarily complex definition for the two primitives, leading to failure of the entire approach. In their paper analyzing kd-trees [85], Friedman, Bentley, and Finkel remarked that the distance measure does not need to be a metric, but instead must obey a different set of requirements, also discussed by Reiss et al. [188].

The entire kd-tree method relies on the assumption that the function $d(\mathbf{z}, \mathbf{z}')$ grows monotonously with $\left|\mathbf{z}_{[i]} - \mathbf{z}'_{[i]}\right|$ in any dimension i. A simplifying step is to restrict analysis to distance functions of the form:

$$d(\mathbf{z}, \mathbf{z}') = D\left(\sum_{i=1}^{K} d_i\left(\mathbf{z}_{[i]}, \mathbf{z}'_{[i]}\right)\right) \tag{5.31}$$

$$d_i : \mathbb{K}^2 \to \mathbb{K}' \tag{5.32}$$

$$D : \mathbb{K}' \to \mathbb{R} \tag{5.33}$$

$$\mathbb{K}, \mathbb{K}' \subseteq \mathbb{R} \tag{5.34}$$

Although imposing this form to d seems rather restrictive, most distance functions naturally come in this form. For example, for Minkowski distances of order p defined as $L^p(\mathbf{z}, \mathbf{z}') \triangleq$

$\left(\sum_{i=1}^{F}\left|\mathbf{z}_{[i]}-\mathbf{z}'_{[i]}\right|^{p}\right)^{1/p}$ (also refer to § 3.2.1 and Eq. 3.3), we have $D(x)=x^{1/p}$ and $d_i(x,x')=\left|x-x'\right|^{p}$. However, the cosine distance function (§ 3.2.1, Eq. 3.5) notably does not fit this mold. (However, cosine distance being the square of a metric, other algorithms such as metric trees can be used in conjunction with it.)

Given the form in Eq. 5.31, Friedman et al. [85] defined the following restrictions the distance function's components in order to be usable with kd-trees:

1. All of d_i are symmetric:

$$d_i(x,x')=d_i(x',x) \tag{5.35}$$

2. The partial applications $d_i|_{x'=x_0}:\mathbb{K}\to\mathbb{R}$, $d_i|_{x'=x_0}(x)\triangleq d_i(x,x_0)$ have exactly one nonnegative local minimum at $x=x_0$ for all $x_0\in\mathbb{K}$:

$$x_0\le x\le x'\Rightarrow d_i(x_0,x)\le d_i(x_0,x') \tag{5.36}$$
$$x_0\ge x\ge x'\Rightarrow d_i(x_0,x)\ge d_i(x_0,x') \tag{5.37}$$

3. The distance between identical points is zero:

$$d_i(x,x)=0\quad\forall x\in\mathbb{K} \tag{5.38}$$

   (The original paper analyzing the distance requirements for kd-trees by Friedman, Bentley, and Finkel [85] does not mention the requirement in Eq. 5.38. However, that requirement is necessary, as clarified by the two theorems below. The omission might have gone unnoticed because virtually all distance measures between identical points have zero components in all dimensions.)

4. The function D is monotonically increasing:

$$x\le x'\Rightarrow D(x)\le D(x') \tag{5.39}$$

We now define BOUNDSENCLOSEBALL and BOUNDSOVERLAPBALL to take advantage of these restrictions. The following two algorithm definitions and their associated correctness proofs are similar to those introduced by Bentley et al. [85].

BOUNDSENCLOSEBALL The invocation BOUNDSENCLOSEBALL$(R,\mathbf{z},r)$ (Algorithm 7) returns **true** if the hyperrectangle R engulfs completely and strictly (no tangent points) the hypersphere centered at $\mathbf{z}$ of radius r, and **false** otherwise. The function carries the task by simply evaluating in each dimension whether the extrema of the hypersphere fall outside of the extrema of the hyperrectangle. The hyperrectangle is represented as two points: the corner with the lowest coordinates, denoted as $R^{\min}$, and the corner with the highest coordinates, denoted as $R^{\max}$. The

Algorithm 7: The BOUNDSENCLOSEBALL subroutine returns **true** if and only if hyperrectangle R completely engulfs hypersphere centered at $\mathbf{z}$ of radius r. The precondition is that $\mathbf{z} \in R$.

Input: Hyperrectangle $R \in \mathbb{R}^{K \times 2}$, point $\mathbf{z} \in \mathbb{R}^K$, radius $r \in \mathbb{R}_+$.
Output: Boolean indicating whether the hypersphere of radius r centered in $\mathbf{z}$ is completely enclosed inside the hyperrectangle R.

```
for i = 1 .. K do
    if D(d_i(z_[i], R^min_[i])) ≤ r then
        return false;
    end
    if D(d_i(z_[i], R^max_[i])) ≤ r then
        return false;
    end
end
return true;
```

comparisons are non-strict, i.e. a hypersphere tangent to a face of the hyperrectangle is conservatively assumed to not be enclosed. This is because there might be points right at the intersection that belong to a different branch of the kd-tree.

Whenever it is computationally more advantageous to work with the inverse of D, inequalities may be expressed and computed in terms of D^{-1}, for example $d_i\left(\mathbf{z}_{[i]}, R^{\min}_{[i]}\right) \leq D^{-1}(r)$.

Theorem 5.6.1 (Friedman et al. [85])**.** BOUNDSENCLOSEBALL *is correct under the assumptions in Eqs. 5.31–5.39.*

Proof. Assume that the function BOUNDSENCLOSEBALL returns **true** but there is still a point $\mathbf{z}^*$ inside of the hypersphere but outside the bounds:

$$\sum_{i=1}^{K} d_i\left(\mathbf{z}_{[i]}, \mathbf{z}^*_{[i]}\right) \leq D^{-1}(r) \tag{5.40}$$

By definition of R, $\mathbf{z}^*$ is outside of it if for at least one dimension j, $\mathbf{z}^*_{[j]} \leq R^{\min}_{[j]}$ or $R^{\max}_{[j]} \leq \mathbf{z}^*_{[j]}$. Given (by the precondition) that $R^{\min}_{[j]} \leq \mathbf{z}_{[j]} \leq R^{\max}_{[j]}$, by Eq. 5.37, either (respective to the two cases) $d_j\left(\mathbf{z}_{[j]}, (\mathbf{z}^*)_{[j]}\right) \geq d_j\left(\mathbf{z}_{[j]}, R^{\min}_{[j]}\right)$, or $d_j\left(\mathbf{z}_{[j]}, \mathbf{z}^*_{[j]}\right) \geq d_j\left(\mathbf{z}_{[j]}, R^{\max}_{[j]}\right)$. Given that BOUNDSENCLOSEBALL returns **true**, $d_j\left(\mathbf{z}_{[j]}, R^{\min}_{[j]}\right) > D^{-1}(r)$ and $d_j\left(\mathbf{z}_{[j]}, R^{\max}_{[j]}\right) > D^{-1}(r)$. So $d_j\left(\mathbf{z}_{[j]}, (\mathbf{z}^*)_{[j]}\right) > D^{-1}(r)$, which implies $d\left(\mathbf{z}, \mathbf{z}^*\right) > D^{-1}(r)$ (as sum of nonnegative terms per Eq. 5.38), contradicting the hypothesis.

Conversely, assume BOUNDSENCLOSEBALL returns **false**. Therefore there is at least one dimension j satisfying $d_j\left(\mathbf{z}_{[j]}, R^{\min}_{[j]}\right) \leq D^{-1}(r)$. (The other case involving $R^{\max}_{[j]}$ is similarly handled.) Let us choose the point $\mathbf{z}^*$ with the same coordinates as $\mathbf{z}$ except in dimension j where the

coordinate is $\mathbf{z}'_{[j]} = R^{\min}_{[j]}$. That point is on the surface of R so it is not enclosed inside of it. The distance between $\mathbf{z}$ and $\mathbf{z}^*$ has all terms equal to zero (Eq. 5.38) except in dimension j, leading to the value $d(\mathbf{z}, \mathbf{z}^*) = D\left(d_{i0}\left(\mathbf{z}_{[j]}, (\mathbf{z}^*)_{[j]}\right)\right) \leq r$. So the point $\mathbf{z}^*$ is in the ball but not enclosed in the hyperrectangle. □

BOUNDSOVERLAPBALL The subroutine BOUNDSOVERLAPBALL (Algorithm 8) is only slightly more complicated than BOUNDSENCLOSEBALL. It returns **true** if and only if there is some non-empty intersection between the hyperrectangle R and the hypersphere centered at $\mathbf{z}$ of radius r. (The original definition [85] returns the complement, i.e. **true** if there is no intersection.) In spite of the fact that the intersection itself may have a complicated shape, getting the yes/no answer is simple. The point on the surface of R closest to the sphere is calculated. That point may be a corner, an edge, or a face of the hyperrectangle. Regardless of the placement, each coordinate of that point is easy to compute on a by-case basis. Note that although the closest point is computed using simple inequalities, the actual distance to it is computed using the accurate d function. Comparing that distance against the sphere's radius yields the final result. This function makes the same assumptions about d as BOUNDSENCLOSEBALL.

Algorithm 8: BOUNDSOVERLAPBALL returns **true** if and only if there exists a non-empty intersection between the hyperrectangle R and the hypersphere centered at $\mathbf{z}$ of radius r.

Input: Hyperrectangle $R \in \mathbb{R}^{K\times 2}$, point $\mathbf{z} \in \mathbb{R}^K$, radius $r \in \mathbb{R}_+$.
Output: Boolean indicating whether the hypersphere of radius r centered in $\mathbf{z}$ intersects the hyperrectangle R.

```
1  s ← 0;
2  for i = 1 .. K do
3      if z_[i] < R^min_[i] then
4          s ← s + d_i(z_[i], R^min_[i]);
5      else if z_[i] > R^max_[i] then
6          s ← s + d_i(z_[i], R^max_[i]);
7      else
8          continue;
9      end
10     if s ≥ D^{-1}(r) then
11         return false;
12     end
13 end
14 return true;
```

Theorem 5.6.2 (Friedman et al. [85])**.** BOUNDSOVERLAPBALL *is correct under the assumptions in Eqs. 5.31–5.39.*

Proof. The proof relies on showing that the point implicitly chosen in the loop during the computation of s is the closest to $\mathbf{z}$ on the surface of R. If there was any closer point, it would have to have

at least one of the component distances smaller than that chosen in the loop (by monotonicity of d_i in all dimensions). But BOUNDSOVERLAPBALL already chooses the extremum of each coordinate that is closest to $\mathbf{z}$, so by Eq. 5.37 that choice also minimizes d_i in each dimension. □

5.6.2.4 Adapting kd-tree Search to Jensen-Shannon Space

Interestingly, although kd-trees are almost always used with distances d derived from a norm (by defining $d(x, x') = \|x - x'\|$), the restrictions in Eqs. 5.35–5.39 do not require d to be norm-based, which is relieving because Jensen-Shannon divergence cannot be easily expressed as the norm of a difference.[3] The reader interested in the norm associated with d_{JS} is encouraged to peruse Topsøe's work [217], which shows that a norm does exist for d_{JS} but does not have an analytic form. For our purposes, we only need to prove that d_{JS} is also suitable for use with kd-trees, for which reason we will prove the following theorem.

Theorem 5.6.3. *The function* $f : [0, 1] \times [0, 1] \to \mathbb{R}$

$$f(x, x') = x \log \frac{2x}{x + x'} + x' \log \frac{2x'}{x + x'} \tag{5.41}$$

is symmetric and has a partial application $f|_{x'=x_0}$ *with exactly one local minimum in* $x = x_0$ *equal to zero.*

Proof. Symmetry is immediate by renaming x to x' and vice versa; we obtain the same function. To find minima of the partial application $f|_{x'=x_0}$, let us take its first derivative:

$$(f|_{x'=x_0})' = \log \frac{2x}{x + x_0} + x \cdot \frac{x + x_0}{2x} \cdot \frac{2(x + x_0) - 2x}{(x + x_0)^2} - x_0 \cdot \frac{x + x_0}{2x_0} \cdot \frac{2x_0}{(x + x_0)^2} \tag{5.42}$$

$$= \log \frac{2x}{x + x_0} \tag{5.43}$$

The fraction $\dfrac{2x}{x + x_0}$ is positive and monotonically increasing on $[0, 1]$ and therefore $(f|_{x'=x_0})'$ is also monotonically increasing. The point at which $(f|_{x'=x_0})'$ intersects the $y = 0$ axis is where $\dfrac{2x}{x + x_0} = 1 \Rightarrow x = x_0$. At that point the function reaches its only local extremum. It is trivial to verify that the extremum is a minimum with value $f|_{x'=x_0}(x_0) = 0$, so the function is also nonnegative, which concludes the proof. □

The connection between Theorem 5.6.3 and our goal is that the function f described therein is one term of the Jensen-Shannon divergence function d_{JS}. For d_{JS}, D is simply scaling the sum of the components by $\dfrac{1}{2}$: $D(x) = \dfrac{x}{2}$. Theorem 5.6.3 proves that we can use d_{JS} directly with kd-trees and achieve correct results.

[3] However, this means we are foregoing some optimization opportunities, such the aforementioned incremental distance calculation [9].

5.7 Scalable Hyperparameter Tuning for the Gaussian Kernel

The data-driven method discussed in Chapter 3 defines distances on the outputs of a first-pass classifier. The resulting feature space consists of probability distributions over the desired classes, and probability divergence measures with well-understood statistical properties can thus be used as distance measures. Our experiments use a neural network with softmax output, trained on the original MFCC features, as the first-pass classifier, and Jensen-Shannon divergence as a distance measure.

The distance $d_{\mathtt{JS}}$ is converted to a similarity measure by using a Gaussian kernel of parameterized width (Eq. 2.1):

$$\mathtt{w}_{ij} = \exp\left[-\frac{d_{\mathtt{JS}}(\mathtt{x}_i, \mathtt{x}_j)^2}{\alpha^2}\right] \tag{5.44}$$

The quality of the similarity hinges on finding a good value for the hyperparameter α. Choosing the optimal α is an open research question; several heuristic methods have been used in practice. Zhu [238] optimizes α to yield a labeling of minimum entropy, subject to the constraint that the labeling must respect the labels of the training set, and also discusses a heuristics based on the Minimum Spanning Tree (MST). The MST-based method entails choosing $\alpha = \frac{d^0}{3}$, where d^0 is the smallest distance between two labeled points bearing different labels. This method is extremely sensitive to noise, as one or one pair of outlying samples is enough to influence the choice of α decisively. Optimizing α for minimum entropy is more robust, but carrying the optimization (by e.g. gradient descent) for each utterance would add considerable overhead to the classification time.

Corpora with only low amounts of labeled data ($\mathtt{t} \ll \mathtt{u}$) make the issue of effective hyperparameter training particularly difficult because there is little or no development data to tune parameters against. We propose an efficient and scalable method of calibrating α that works offline (only uses the training data) and is inspired by maximum margin techniques. As such, our method enjoys the usual properties of maximum-margin techniques such as robustness to noise and good separation capabilities.

First, we compute the average intra-class distance (d_{intra}) and inter-class distance (d_{inter}):

$$d_{\text{intra}} = \frac{\sum_{i,j:\ i\neq j, \mathtt{y}_i = \mathtt{y}_j} d(\mathtt{x}_i, \mathtt{x}_j)}{\operatorname{card}\left\{\{i,j\} \in \{1,\ldots,\mathtt{t}\}^2 \;\middle|\; i \neq j, \mathtt{y}_i = \mathtt{y}_j\right\}} \tag{5.45}$$

$$d_{\text{inter}} = \frac{\sum_{i,j:\ \mathtt{y}_i \neq \mathtt{y}_j} d(\mathtt{x}_i, \mathtt{x}_j)}{\operatorname{card}\left\{\{i,j\} \in \{1,\ldots,\mathtt{t}\}^2 \;\middle|\; \mathtt{y}_i \neq \mathtt{y}_j\right\}} \tag{5.46}$$

where N_{intra} and N_{inter} are the counts of the respective terms. Ideally, $d_{\text{intra}} > d_{\text{inter}}$ by a large margin, otherwise the data has poor separability. In fact, comparing d_{intra} and d_{inter} gives a good gauge of the quality of the feature selection and distance measure. We then choose α such that two samples distanced at $\frac{d_{\text{inter}} + d_{\text{intra}}}{2}$ have a similarity of 0.5:

$$\exp\left[-\frac{(d_{\text{intra}} + d_{\text{inter}})^2}{4\alpha^2}\right] = \frac{1}{2} \Rightarrow \alpha = \frac{d_{\text{intra}} + d_{\text{inter}}}{2\sqrt{\ln 2}} \tag{5.47}$$

The intuition behind this choice is that, given that both distance and similarity have range $[0, 1]$, two samples placed at the most ambiguous distance should be midway in terms of similarity as well.

Computing the average distances d_{intra} and d_{inter} would necessitate $O(\mathtt{t}^2)$ distance computations, one for each pair of training samples. A time-efficient approach we choose in practice is to do a random sampling: two samples $\mathtt{x}_i$ and $\mathtt{x}_j$ are randomly chosen from the training set, their distance is computed and considered for d_{intra}, if $\mathtt{y}_i = \mathtt{y}_j$, or for d_{inter} otherwise. We used 2.5% of the data in five successive trials; α varied by no more than 1% among trials. This is encouraging in scenarios where ($\mathtt{t} \gg \mathtt{u}$).

Choosing α in this manner yielded much better performance in our tests than grid search and a method based on the Minimum Spanning Tree [238, § 7.3].

5.8 Speed Measurements for Unstructured Classification

We have measured the run time of the Vocal Joystick experiment enhanced with kd-trees for nearest neighbors estimation, graph reduction, and in-place label propagation. The moderate corpus size allowed us to run, for comparison purposes, the brute-force nearest neighbors algorithm for graph construction and also the classic iterative label propagation (Algorithm 1). (We were unable to construct the full graph in memory so there is no comparison point for full graph versus reduced graph memory consumption; this is admittedly an obvious point that needs no experimental evidence.) We then measured the run time of the proposed approach using kd-trees for accelerating nearest neighbors search, and our in-place label propagation (Algorithm 1).

In both brute-force and kd-tree experiments, one graph was built for each test utterance (in keeping with the approach to measuring accuracy). Therefore two nearest-neighbor searches must be performed. First, the training set must be searched for each sample in the current utterance (for the labeled-unlabeled connections forming $\mathtt{P}_{\mathtt{UL}}$). Second, the samples in the current utterance must be also cross-searched (for the labeled-unlabeled connections forming $\mathtt{P}_{\mathtt{UU}}$). Again in both cases, we observed the common practice of keeping the list of current nearest neighbor candidates in a binary heap [52] in order to not let the size of the list add to the complexity. The number of nearest neighbors retained was $k = 10$.

In the brute-force experiments, we computed the nearest neighbors by linear search against both sets. In the kd-tree experiments, we first built one kd-tree for the entire training set. The tree was then reused across all test utterances. Then, for each test utterance, we built a separate kd-tree. This second tree is needed to compute the unlabeled-to-unlabeled connections.

The same systems-level optimizations have been applied to both implementations. The system accuracy has been the same in both cases, although the individual soft labels have been slightly different on occasion due to different order in which floating point operations have been carried.

We conducted five timing experiments, one at a time, on the same machine and computed average and standard deviation in each case. The computer used was an AMD64 machine with 2 GHz clock speed and 8 GB RAM. All data was stored on a local disk. The results shown in Table 5.1 reveal two facts. First, kd-trees bring over two orders of magnitude improvement ($127\times$) in terms of speed. Second, the classification time is dominated by graph construction; although our proposed in-place label propagation is significantly faster than the state-of-the-art alternative, there is little improvement in total runtime.

The in-place label propagation was faster because it converged faster than classic iterative la-

Step	Run time (seconds)
Graph construction (brute force nearest neighbors)	24703.07 ± 414.56
Graph construction (kd-trees)	193.77 ± 0.57
Label propagation iterative per [238]	6.45 ± 1.72
Label propagation in-place proposed in § 5.3.1	2.59 ± 0.41

Table 5.1: Run time for brute force graph construction and original label propagation vs. kd-trees and in-place label propagation. Graph construction is improved by two orders of magnitude. Convergence speed is also largely improved, but has a relatively small contribution to the overall run time.

bel propagation. Over the 49 graphs constructed for the dev set, classic iterative label propagation [238] took on average 21.45 steps to converge. The proposed in-place label propagation took on average 8.86 steps to converge. The complexity of each approach is the same. The absolute improvements in runtime depend on the density of the graphs.

5.9 Fast Graph Construction in String Spaces

Chapter 4 discusses graph-based learning using string kernels. Let us discuss scalability considerations when string kernels are used as a similarity measure. The costliest operation involved in creating the graph is computing the similarities $\sigma(\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle, \langle\langle \mathtt{x}_j, \mathtt{y}_j \rangle\rangle)$ for all pairs of pairs (sic) $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$, $\langle\langle \mathtt{x}_j, \mathtt{y}_j \rangle\rangle$ in the test set. Recall that for structured learning with string kernels we use a hypothesis-based approach that relies on an external generator to create several hypotheses for each unlabeled sample, so the semi-supervised (sub)system needs to regress a scoring function for pairs $\langle\langle \mathtt{x}_i, \mathtt{y}_i \rangle\rangle$. On the source side ($\mathtt{x}$), the total number of kernel computations is (after eliminating all unnecessary computations $\kappa(\mathtt{x}_i, \mathtt{x}_i)$ and taking the symmetry $\kappa(\mathtt{x}_i, \mathtt{x}_j) = \kappa(\mathtt{x}_j, \mathtt{x}_i)$ into account):

$$C_{\mathtt{x}} = \mathtt{u} \cdot \mathtt{t} \frac{\mathtt{u}(\mathtt{u}-1)}{2} \tag{5.48}$$

This scales poorly with $\mathtt{u}$ so we need to improve on that, particularly when we consider that each kernel evaluation takes $\mathcal{O}(|\mathtt{x}_i| \cdot |\mathtt{x}_j|)$ time (in a dynamic programming implementation). Furthermore, on the target side ($\mathtt{y}$), the total number of kernel evaluations in a naïve/greedy approach is

$$C_{\mathtt{y}} = \frac{\left(\sum_{i=1}^{\mathtt{u}} \operatorname{card} \mathcal{Y}(\mathtt{x}_i)\right)\left[\left(\sum_{i=1}^{\mathtt{u}} \operatorname{card} \mathcal{Y}(\mathtt{x}_i)\right) - 1\right]}{2} \tag{5.49}$$

because there is one kernel evaluation for each pair of hypotheses, and there are $\sum_{i=1}^{\mathtt{u}} \operatorname{card} \mathcal{Y}(\mathtt{x}_i)$ total hypotheses.

In the following we will focus on the general problem of computing all similarities between two sets of strings.

5.9.1 Inverted Index

Given that kernel evaluations against pairs of strings is relatively expensive, there is a strong motivation for finding fast, inexpensive approximations of the real value. This technique is akin to finding the nearest neighbors when constructing the graph: instead of operating on a very connected graph, we approximate it by only keeping the strongest edges. A good approximation of the string kernel would eliminate highly dissimilar string pairs, which form the bulk, and keep the most similar strings. Ideally the approximation method would have good recall such that no significantly similar pairs are lost. The precision influences speed because a low precision forces many unneeded or low-yield kernel evaluations.

One widely known data structure for approximating string similarities is known as *inverted index*. The inverted index (also called inverted file by Knuth [123, Vol. 3, § 6.5] or postings file) is a data structure dating to way before the beginnings of automated computing. Book indexes are some of the earliest examples of systematic creation and use of inverted indexes. Inverted indexes have been a mainstay in computing and have seen a revived interest with the advent of data mining, information extraction, and Internet-scale search engines.

An inverted index [25] is a general structure applicable to generic strings ("string" as defined in § 4.4.5, Definition 4.4.1). We formally define an inverted index below.

Definition 5.9.1. Given a collection of strings $S = \langle\langle s_1, \ldots, s_n \rangle\rangle$ over an alphabet Σ, an *inverted index* is an associative array I that associates each element $w \in \Sigma$ to the set

$$I(w) = \left\{ s_i \in S \,|\, \exists j \in \{1, \ldots, |s_i|\}, w = (s_i)_{[j]} \right\} \tag{5.50}$$

that maps w to the subset of strings in which w occurs. An element $I(w)$ of I is called an *inverted list*.

In our application to Machine Translation and similar structured problems, a string s_i is a sentence and the collection S is a document (e.g. a training set). If a balanced tree is used as an intermediate data structure, building an inverted index involves scanning S sequentially and appending for each sentence s_i a sentence identifier (e.g. the sentence number i) to the node of the tree corresponding to each word in s_i. That means $\mathcal{O}(|s_i| \log \Sigma)$ time for sentence s_i. The total time for constructing an inverted index for S is therefore $\mathcal{O}\left(\left(\sum_{s \in S} |s|\right) \log \Sigma\right)$, which is satisfactory even using a straightforward algorithm. Sorting the obtained string list for each word is useful and adds $\mathcal{O}\left(\left(\sum_{s \in S} |s|\right) \log \left(\max_{s \in S} |s|\right)\right)$ time. Finding all potentially similar sentences given the inverted index is, however, a nontrivial algorithm, for which reason we discuss it in detail. (However, we don't consider it original to this work as similar and more involved techniques are to be found in the literature [241].)

5.9.1.1 Normalization by String Length

If the number of words in common is used as approximation for string similarity, then long training strings would be at an unfair advantage: long strings contain many words, and therefore they will appear similar to many short strings. We have already met a similar problem when discussing string

kernels (§ 4.3.1.1) where it was revealed that normalization is needed to obtain unbiased kernels. Therefore we define an approximated similarity for strings by normalizing by the geometric mean of the number of words in the strings, as follows.

Definition 5.9.2. Given two non-empty strings s and t and denoting with $W(s) \triangleq \{w \in \Sigma \,|\, \exists i \in \{1, \ldots, |s|\}, w = s_i\}$ and $W(t) \triangleq \{w \in \Sigma \,|\, \exists i \in \{1, \ldots, |t|\}, w = t_i\}$ the sets of (distinct) elements in s and respectively t, we define the normalized bag-of-words similarity of s and t as:

$$\sigma_b(s,t) = \frac{\text{card}(W(s) \cap W(t))}{\sqrt{\text{card}(W(s)) \cdot \text{card}(W(t))}} \tag{5.51}$$

Similarity σ_b is bounded within $[0, 1]$ and can be considered an approximation of all normalized string kernels discussed in § 4.3.1.3. This is because those kernels rely on common words and also on the relative ordering of words; σ_b does measure word commonality, but ignores their ordering. As such, σ_b may return higher similarities than the actual kernels, but also lower similarities because it does not account for repeated words.

The plan is to devise a fast approximate method for finding the most similar strings with high likelihood with a relatively small computational effort. After this step, the precise kernel computes the actual similarities starting from the trimmed candidate list.

5.9.1.2 Algorithm for Approximating Most Similar Strings using an Inverted Index

Given some string s and an inverted index I, our aim is to quickly find the strings that contain the most words in common with s. This will not yield a precise ranking of the most similar strings according to the kernel because it only focuses on 1-grams and ignores word order and hence all higher-order n-grams. However, it does provide a reasonable approximation to any string kernel. Algorithm 9 shows how the most similar strings can be efficiently found in a numeric inverted index.

The algorithm first selects a subset C of the index corresponding to the words contained in the string, ignoring all the rest (section starting at line 2). This step takes $\mathcal{O}(|s|)$ time and is where most computational savings will occur, assuming $|s|$ is small relative to $|\Sigma|$. After this step, C is systematically and exclusively used for counting the number of common strings.

The algorithm makes use of two binary heaps [123, Vol. 3, § 5.2]. The first heap, H_R, is a min-heap of pairs $\langle\langle m, k \rangle\rangle$ containing string IDs and the number of common words they share with the query string s. The binary min-heap is ordered by projection of its pairs on the second member (k, the occurrence count; the string ID is irrelevant to ordering). Therefore, searching the heap for the string *least similar* to s is done in $\mathcal{O}(1)$ time. Inserting a new string in the heap takes $\mathcal{O}(\log |H_R|) = \mathcal{O}(\log n)$ time.

The second binary min-heap, H_C, is a less usual construct. It organizes arrays of inverted lists, and as such care must be exercised when reading Algorithm 9 so as to not confuse elements of this heap with elements of each inverted list stored in the heap. For example, *top*(H_C) is the top of the heap (consisting of one entire list of string IDs), whereas $(\textit{top}(H_C))_{[1]}$ is the leftmost string ID in the list at the top of the heap. The unusual element is the ordering induced by the heap: two heap elements (i.e., two inverted lists) a and b found in H_C are ordered by the relation:

$$\textit{frontOrder}(a, b) \triangleq a_{[1]} < b_{[1]} \tag{5.52}$$

Algorithm 9: Finding the strings sharing most words with a given string in an inverted index.

Input: String s without repeated words; inverted index I, each $I(w)$ is a sorted array of numeric string identifiers; n, the limit for the most similar strings.
Output: The top n strings containing the most words in common with s.

```
   /* Create the searched subset of I                                    */
2  C ← ∅;
3  for w ∈ s do
4      if I(w) ≠ ∅ then C ← C I(w)
5  end
   /* Search C transversally maintaining the result heap                 */
6  H_R ← makeEmptyHeap();
7  H_C ← makeFrontHeap(C);
8  while |H_C| > 0 do
       /* Select minimum m and its count k off index's head              */
9      m ← (top(H_C))_1;
10     k ← 0;
11     repeat
12         k ← k + 1;
13         top(H_C) ← (top(H_C))_[2..|top(H_C)|];
14         if |top(H_C)| = 0 then
16             pop(H_C);
17         else
19             percolateDown(H_C);
20         end
21     until |H_C| = 0 ∨ (top(H_C))_[1] ≠ m ;
23     if length(H_R) < n then
24         push(H_R, ⟨⟨m, k⟩⟩ );
25     else if top(H_R).k < k then
27         replaceTop(H_R, ⟨⟨m, k⟩⟩ );
28     end
29     if |H_C| ≤ top(H_R).k ∨ top(H_R).k = |s| then
30         break while;
31     end
32 end
33 return H_R;
```

In other words, H_C orders inverted lists by the ID of the first string. Given that the inverted lists are sorted in ascending order by string ID, H_C introduces ordering by the *globally smallest string ID* present in the index. This means that H_C offers $\mathcal{O}(1)$ access to the lowest string ID in the entire set C. As items are removed from the heap (and C), heap maintenance preserves this property in only $\mathcal{O}(\log |H_C|)$ time. That is not the cumulated length of all lists, but instead the relatively small number of inverted lists. $|H_C|$ is initially equal to $|s|$ (the length of the sought string) and decreases as elements are removed from H_C. Also note that swapping elements in H_C does not entail swapping entire inverted lists, but instead swapping indirect pointers to the lists. As such, swapping two elements of H_C is $\mathcal{O}(1)$ and therefore operations on H_C obey the usual complexity bounds.

The outer **while** loop counts, in each pass, the total number of words shared by s and the indexed string with the globally smallest ID found in C. The approach is to repeatedly eliminate the first ID

Primitive	Complexity	Comments
makeFrontHeap(C)	$\mathcal{O}(\|C\|)$	Organizes elements in C as a heap using Eq. 5.52 as the ordering relation. No additional storage is necessary; C is organized in situ by swapping its elements in-place. Empty lists in C are not put in the heap.
$\|H\|$	$\mathcal{O}(1)$	Number of elements in heap H.
top(H)	$\mathcal{O}(1)$	Returns the element at the top of heap H. (Usually that element is stored at the first position in the array underlying the heap.)
pop(H)	$\mathcal{O}(\log \|H\|)$	Removes the top of heap H while preserving the heap property.
percolateDown(H)	$\mathcal{O}(\log \|H\|)$	Assuming the top of the heap has mutated, re-establishes the heap property by swapping that element appropriately. Rönngren and Ayani [192] argue that the practical average insertion time is $\mathcal{O}(1)$.
replaceTop(H, e)	$\mathcal{O}(\log \|H\|)$	Replaces the top of the heap with e and then re-establishes the heap property. Technically not a primitive: evaluates *replaceTop*$(H) \leftarrow e$ followed by *percolateDown*(H).

Table 5.2: Heap primitives used by Algorithm 9. General texts on algorithms and data structures [123, 52] cover implementation of heap primitives in detail.

in the top string in heap H_C. After each such operation, the inverted list might have become empty (in which case it is removed off the list, line 16) or it still contains elements, in which case the heap property must be preserved (line 19). The count of successful ID extraction operations (i.e., passes through the **repeat** loop) is exactly the number of (distinct) words that query string s and string m have in common.

Lines 23 through 27 perform the insertion in the result heap. In a manner common to top-n algorithms, insertion is done with "saturation:" we are only interested in the top n matches so if $|H_R| = n$ and a match was found better than the worst match seen so far, we just replace that worst match with the found one. Recall that H_R is a min-heap of which top is the string having the *fewest* words in common with s.

Complexity Analysis The innermost **repeat** loop makes one step for each training string that has at least one word in common with the test string. Due to the heap management, that step takes $\log |H_C|$ time. In turn, $|H_C|$ decreases as elements are consumed off H_C, but in the worst case it is no greater than $|s|$. So each pass through the **repeat** loop takes $\log |s|$ time.

Adding (or replacing) one element in H_R takes $\mathcal{O}(n)$ time, but is only done on average once every $\overline{k}$ steps, where $\overline{k}$ is the average number of words that s has with a string in C. In the worst case, we have many train strings each sharing only one word with s.[4] So in the worst case at each

[4]Due to the way C was created, any string in it has at least one word in common with s.

pass through the outer loop we are taking $\mathcal{O}(\log(n \cdot |s|))$ time.

The outer loop ceases when C has been exhausted entirely, which totals as many steps as accumulated occurrences in C of words in s:

$$T = \mathcal{O}\left(\log(n \cdot |s|) \sum_{w \in s} \mathit{count}(w)\right) \tag{5.53}$$

where the *count* function is the number of occurrences of word w in the inverted index.

This is the worst case complexity. The worst case situation occurs when each training string has exactly one word in common with the test string, and when C contains a large fraction of the corpus, i.e. s is a long string containing many distinct words. In practice this seldom happens, but at any rate any skewing would affect the $\log(n \cdot |s|)$ factor, which is small to begin with.

Scalability Considerations Algorithm 9 is scalable to large systems because in addition to its good theoretical complexity it also enjoys a number of properties relevant for practical implementations. The inverted lists are scanned strictly sequentially and only their current element needs to be in memory in order to be organized in heap H_C. The inverted index is therefore friendly to external storage. Cache locality is not very good, however, because the lists are spanned in lockstep, therefore a long searched sentence could fill the cache lines such that memory thrashing will occur. In the worst case, the sentence IDs are distributed evenly across the inverted lists; a more cache-favorable case is to have long running sequences of IDs that belong to a minority of lists.

NLP-Specific Complexity Considerations In NLP applications, usually the sought string is a sentence and the inverted index maps words (or word tags) to sentences or documents in which the word (or tag) occurs. The larger factor is the occurrence count of the searched string's elements in the training set. If the distribution of vocabulary elements in the corpus were approximately uniform, the count would be proportional to $|s|$ and to the number of strings in the corpus. However, words in natural language sentences are Zipf-distributed [240, 130, 143] (the frequency of a word is roughly double the frequency of the next less-frequent word). The distribution is skewed towards the extremes, i.e. the most few frequent words decay slower and the least frequent word frequencies decay faster [137]. If we assume that the Zipf distribution applies to individual strings as well, then longer test sentences have an exponentially decreasing overlap with training strings because they contain less and less frequent distinct words. So we can practically consider that $\sum_{w \in s} \mathit{count}(w)$ depends on the size of the training data but not on the length of s.

Following the Zipf law, the inverted index itself is very jagged (the number of elements in the inverted lists drops exponentially). The most frequent word is "the," occurring in 6.2%-7% of all sentences [137, List 1.2]. The frequency decays to under 1% by the eighth ranked word ("is"). This means that after stop word elimination, Algorithm 9 can perform approximate similarity searches based on an inverted index by only accessing less that 1% of the training set sentences.

5.9.1.3 Loss of Inverted Index Compared to the Gapped String Kernel

In using an inverted index for approximating the most similar sentences (where the desired exact similarity would be computed by a gapped string kernel), there are two sources of inaccuracy:

- *Word Repetition:* In the inverted index, at least as implemented herein, repeated occurrences of the same word within a sentence are not recorded; the index only tells whether a word occurs in a sentence at least once. Stop words appear repeatedly in sentences more often than meaningful words, so stop word elimination should limit this source of noise. Another useful technique for reducing the effect of word repetition is to use sentence segmentation by breaking compound and complex sentences into simple sentences.

- *Word Order:* The inverted index does not retain an important source of information—the order of words in the original sentence. For example, the sentences "work smart not hard" and "work hard not smart" are put in the same equivalence class by the inverted index.

If the inverted index is used as a pre-filter to limit computation of the expensive kernel to only the top n estimated similar sentences and ignoring all others, the filtering effected by the inverted index may lose some of the most similar sentences (according to the real, expensive-to-compute kernel) and introduces others, not as similar, sentences in the top-n list. We want to estimate the loss introduced, which we will do in two ways:

1. *By counts:* For each test sentence s_i, determine the top n similar sentences $U_i = \langle\langle u_{i_i}, \ldots, u_{i_n} \rangle\rangle$ by using the actual similarity measure we are interested in, and the top n similar sentences $U_i' = \langle\langle u'_{i_1}, \ldots, u'_{i_n} \rangle\rangle$ by using the inverted index. Then the loss is computed as the average relative disagreement of the two sets:

$$\mathcal{L}_c(n) = \frac{\sum_{i=1}^{\mathtt{u}} \left[1 - \frac{\mathrm{card}(U_i \cap U_i')}{\mathrm{card}(U_i)}\right]}{\mathtt{u}} \tag{5.54}$$

   as a value in $[0, 1]$.

2. *By accumulated similarity:* Another approach to loss measurement takes into account the fact that even the sentences mistakenly considered in top n (according to the inverted index) are not uniformly undesirable because some may be in fact close in similarity. To make that distinction, we compare the accumulated similarity of the top n matches as guessed by the inverted index, with the accumulated similarity of the true top n matches according to the string kernel:

$$\mathcal{L}_s(n) = \frac{\sum_{i=1}^{\mathtt{u}} \left(1 - \frac{\sum_{j=1}^{n} \sigma_b(u'_{ij}, \mathtt{x}_i)}{\sum_{j=1}^{n} \sigma_b(u_{ij}, \mathtt{x}_i)}\right)}{\mathtt{u}} \tag{5.55}$$

   Again, the loss is in $[0, 1]$; in the case of a perfect match, there is no loss as the top approximate similarities are the same as the true similarities so the fraction in the enumerator is always 1. This measure is more informative than $\mathcal{L}_c$ because it directly reflects the loss of good connections in the graph that is ultimately built using these most similar sentences.

One simple technique that can be used to reduce the loss when using an inverted index is to *over-allocate* the inverted index results, i.e., having the inverted index method select the top $n_o > n$ estimates, i.e. more than the top n estimated similar sentences. After that step, the precise method inspects those estimates and retains the true top n similar samples found. For example, the inverted index select the top 100 most similar sentences, and then the precise similarity measure is computed for those and only the top 10 are retained. The speed of the approach degrades to a controlled degree, but the gain in precision may be justified. We want to measure to what extent over-allocation helps.

We have measured the loss of using an inverted index to find the most similar sentences out of the Europarl [124] training data for each sentence in the IWSLT 2006 [127] development set. The original set was passed through a statistical chunker to split sentences into smaller chunks, resulting in a total of 3902 chunks. We built an inverted index built from Europarl's training data. After sentence chunking, the train set size was 1,478,564 chunks (which we can consider sentences for practical purposes and we will call them as such). The size of the vocabulary is 72,480 words. The lengths of the sentences varied between 1 and 40 words, with an average length of 12.6 words.

The reference used was the gapped string kernel in § 4.3.1.3 with penalty $\lambda = 0.5$. Because computing the actual top similarities for the entire corpus would have been prohibitively expensive, we approximated the loss on 5 uniform random subsamples of the development set, each totaling 100 samples (about 2.56% of the test set size), and then took the average and standard deviation.

The first experiment computed $\mathcal{L}_c(n)$ and $\mathcal{L}_s(n)$ for various values of n. Table 5.3 displays the results.

n	Count loss $\mathcal{L}_c(n)$ (%)	Similarity loss $\mathcal{L}_s(n)$ (%)
10	66.46±3.01	13.13±2.39
20	65.68±3.23	12.78±2.51
30	65.84±3.97	12.84±2.75
40	65.90±3.31	12.79±2.67
50	65.72±2.42	12.68±2.46

Table 5.3: Loss in the inverted index depending on the cutoff for most similar sentences. The fractional numbers $\mathcal{L}_c(n)$ (Eq. 5.54) and $\mathcal{L}_s(n)$ (Eq. 5.55) are multiplied by 100 to obtain percentages.

The count loss is high for the entire measured range of values of n. On average, there was less than 35% agreement between the n most similar sentences as predicted by the inverted index and the reference string kernel. However, the similarity loss was relatively low, indicating that even when the inverted index did not find the most similar sentences, it did find sentences in the same neighborhood.

The second experiment measured the effect of over-allocation. We kept $n = 10$ and varied n_o between 20 and 1280, in geometric progression. Table 5.4 displays the results.

The count loss $\mathcal{L}_c$ and especially the similarity loss $\mathcal{L}_s$ are seeing a dramatic improvement with growth of n_o. Note that values of n_o that are relatively large compared to n are not degrading speed significantly. This is because n_o compares against $\mathtt{t} = 1,478,564$, the size of the training corpus. Even at $n_o = 1280$, less than one thousandth of kernel evaluations are performed compared to the

n_o	Count loss $\mathcal{L}_c(n)$ (%)	Similarity loss $\mathcal{L}_s(n)$ (%)
20	58.04±2.96	7.45±1.37
40	49.00±3.10	4.62±0.80
80	40.54±2.53	2.86±0.46
160	31.88±1.93	1.84±0.30
320	26.10±1.57	1.27±0.22
640	19.50±1.31	0.75±0.25
1280	14.22±1.73	0.47±0.18

Table 5.4: Dependency of loss on over-allocation. Out of n_o samples selected by using the inverted index, the top $n = 10$ have been retained using the string kernel.

brute force method.

5.9.2 Fast Cross-Product String Kernel Computation

In our application of graph-based learning for Statistical Machine Translation (Chapter 4), even after counting in the benefits of the pre-filtering done by the inverted index, computing the similarities between two hypothesis sets remains a time-consuming step. Recall from Definition 5.5.1 (last construction step) that once two hypothesis sets have been decided to be similar on the source language side, all cross-product similarities between sentences in the two sets must be computed on the target language side. The maximum size of a hypothesis set can be controlled, but that means the search space $\mathcal{Y}(\mathcal{X})$ is truncated, which negatively impacts rescoring. Contemporary Statistical Machine Translation systems use hypothesis sets on the order of 10^3, so computing similarities across two hypothesis sets entails 10^6 kernel evaluations. We set out to improve on that number. One key observation is that hypotheses in any given set are remarkably similar with one another, often differing only by one word or by the order of words.

We formulate the problem as follows: given two sets of strings $S = \{s_1, \ldots, s_{|S|}\}$ and $T = \{t_1, \ldots, t_{|T|}\}$, compute all kernel values $\kappa(s_i, t_j)\ \forall i \in \{1, \ldots, |S|\}, j \in \{1, \ldots, |T|\}$. We are ultimately interested in the normalized kernel values $\hat{\kappa}(s_i, t_j)$ (§ 4.3.1.1), but computing the normalization factors $\kappa(s_i, s_i)$ and $\kappa(t_j, t_j)$ is a linear $\mathcal{O}(|S| + |T|)$ process that can be made part of preprocessing. The bulk of kernel computations is computing kernel values for the Cartesian product $S \times T$. The kernel function of interest may be the n-length gap-weighted string kernel or the all-lengths gap-weighted string kernel, both described in § 4.3.1.3. We will start with the latter as it is easier to discuss and implement; the former follows a similar pattern.

Yin et al. [230] proposed a dynamic programming algorithm to compute the all-lengths gap-weighted string kernel for two strings s and t in time $\mathcal{O}(|s| \cdot |t|)$. We are interested in discussing the actual procedure, so Algorithm 10 (next page) shows it as originally proposed.

The algorithm maintains two bi-dimensional matrices $DPS, DPV \in \mathbb{R}_+^{|s| \times |t|}$ and computes their elements at indices (i, j) from elements at smaller indices, in a classic dynamic programming manner. Before introducing an algorithm for computing kernels over multiple strings, let us notice one fact of interest: The value $DPS(i, j)$ is computed in the inner loop and used immediately;

Algorithm 10: All-lengths gap-weighted kernel as proposed by Yin et al. [230]

Input: Strings s, t; gap penalty $\lambda \in \mathbb{R}$.
Output: All-lengths gap-weighted similarity $K \in \mathbb{R}$

```
1  DPS(1 : |s|, 1 : |t|) = 0;
2  DPV(0, 0 : |t|) = 0;
3  DPV(1 : |s|, 0) = 0;
4  K = 0;
5  for i = 1 : |s| do
6      for j = 1 : |t| do
7          if s_i = t_j then
8              DPS(i, j) ← 1 + DPV(i − 1, j − 1);
9              K ← K + DPS(i, j);
10         end
11         DPV(i, j) ← DPS(i, j)+λDPV(i, j−1)+λDPV(i−1, j)−λ^2 DPV(i−1, j−1);
12     end
13 end
14 return K;
```

past values of DPS are never used. We could eliminate DPS entirely, but let us only modify the algorithm slightly to store a more useful matrix $DPSS$ defined as

$$DPSS(i,j) \triangleq \sum_{k=1}^{j} DPS(i,k) \tag{5.56}$$

So $DPSS$ stores partial sums of columns in DPS. Algorithm 11 (next page) shows the modified algorithm definition. We replaced the matrix DPS with one transitory value $DPSij$, and introduced the $DPSS$ matrix.

In the modified algorithm, the update of K has been hoisted out of the inner loop to the outer loop. This modification does not have optimization consequences, as the inner loop does the same amount of work by updating the $DPSS$ matrix. The more important effect obtained is that now the matrix $DPSS$ enjoys a useful property (along with DPV). If strings s, s' share a prefix of length l_s and strings t, t' share a prefix of length l_t, then let us denote the matrices resulting after the kernel values have been computed for (s,t) and (s',t') respectively as $(DPV, DPSS)$ and $(DPV', DPSS')$. Then

$$DPV(0 : l_s, 0 : l_t) = DPV'(0 : l_s, 0 : l_t) \tag{5.57}$$
$$DPSS(1 : l_s, 1 : l_t) = DPSS'(1 : l_s, 1 : l_t) \tag{5.58}$$

In other words, the matrices share a rectangular region in the top-left corner. The width of the rectangular region depends on the length of the shared prefix between (t,t'), whereas its height depends on the length of the shared prefix between (s,s'). So one simple idea to accelerate computation of all similarities between two sets of strings $S = \{s_1, \ldots, s_{|S|}\}$ and $T = \{t_1, \ldots, t_{|T|}\}$ is to

Algorithm 11: Modified all-lengths gap-weighted kernel

Input: Strings s, t; gap penalty $\lambda \in \mathbb{R}_+$.
Output: All-lengths gap-weighted similarity $K \in \mathbb{R}_+$

$DPSS(1:|s|, 1:|t|) = 0$;
$DPV(0, 0:|t|) = 0$;
$DPV(1:|s|, 0) = 0$;
$K = 0$;
for $i = 1:|s|$ **do**
    for $j = 1:|t|$ **do**
        if $s_i = t_j$ **then**
            $DPSij \leftarrow 1 + DPV(i-1, j-1)$;
            $DPSS(i, j) \leftarrow DPSS(i, j) + DPSij$;
        else
            $DPSij \leftarrow 0$;
        end
        $DPV(i, j) \leftarrow DPSij + \lambda DPV(i, j-1) + \lambda DPV(i-1, j) - \lambda^2 DPV(i-1, j-1)$;
    end
    $K \leftarrow K + DPSS(i, |t|)$;
end
return K;

exploit this property by making the matrices $DPSS$ and DPV persistent (i.e., outlasting one kernel evaluation) and then ordering the Cartesian product $S \times T$ such that consecutive string pairs share as long a prefix as possible. Then, for each kernel computation, only a fraction of the matrices DPV and $DPSS$ must be evaluated.

We could attempt to build structure over the set $S \times T$ directly. However, that set has a large cardinality so it would be preferable to avoid operating on it directly (in all likelihood, handling $S \times T$ would exhibit the high complexity that we wanted to avoid in the first place); a better approach is to induce structure over S and T separately. To do so, let us make an observation derived from Eq. 5.57: for a given string t, two strings s and s' sharing a prefix of length l_s will share the first l_s rows of $DPSS$ and DPV. To compute similarities between s and s' on the left hand side, and t on the right-hand side, we do not need to compute full matrices for each kernel computation; the first l_s rows only need be computed once.

To benefit of such savings, we arrange the strings in S in a *trie* [123, Vol. 3, § 6.3: Digital Searching, pp. 492] and we distribute the rows of the matrices $DPSS$ and DPV along the nodes of the trie. A trie (also known as retrieval tree or prefix tree) provides a compact representation of strings with shared prefixes, which is exactly what is needed. For example, given the sentences set S:

Mary has a praline
Mary has a candy bar
Mary has chocolate

the corresponding word-level trie is shown is shown in Fig. 5.9.2 (next page).

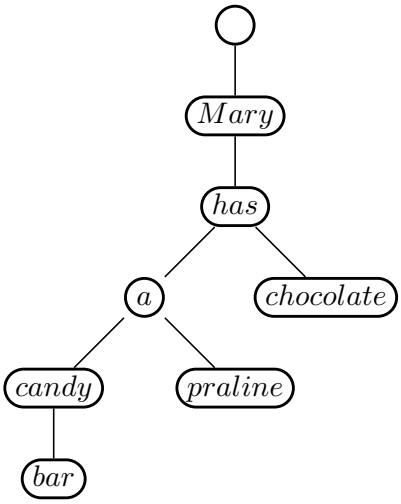


Figure 5.1: Three sentences organized in a trie. The shared prefixes are collapsed together.

Consider that we organize the entire set S in a trie. Key to the proposed algorithm is that we distribute the rows of $DPSS$ and DPV along the nodes of the trie: a node at depth i in the trie stores the i^{th} row of $DPSS$ and DPV. This is correct because those rows would have the same value anyway due to the shared prefix. What we effectively obtained is a compact way to store many $DPSS$ and DPV matrices, one for each string in S.

Consider now the set T containing only the sentence:

Mary has a little lamb

To compute similarities of that sentence with *all* three sentences in S, we perform any root-first traversal of the trie (either depth-first or breadth-first). At each node we compute the entire i^{th} row of $DPSS$ and DPV by using the already-computed row in the parent node. The savings come from the fact that rows for common prefixes in S only need to be computed once. Instead of filling $|t| \sum_{i=1}^{|S|} |s_i|$ rows, only $N \cdot |t|$ rows need to be filled, where $N \leq \sum_{i=1}^{|S|} |s_i|$ is the number of nodes in the trie (except for the root node). Algorithm 12 (next page) computes all similarities of a sentence against a set of sentences. We use the notation $a.b$ to denote "property b associated with entity a," as is the case with many of today's programming languages. Also, we avail ourselves of high-level primitives with obvious implementation, such as $buildTrie$ and $preOrder$.

Algorithm 12 still has a large inefficiency: it exploits common prefixes on the left-hand side, but not on the right-hand side. Consider the right-hand side set T:

Mary has a little lamb

Mary has a tiny lamb

For each of the two strings in T, and for each node r in the trie constructed from S, the vectors $r.DPSS$ and $r.DPV$ are filled from scratch, even though their first three columns are identical. We would like to also avoid repeated computation on the right-hand side. It would appear that organizing T in a trie would yield similar benefits to those obtained for S, but a simpler method that is just as efficient is to simply sort T in lexicographic order. Lexicographical sorting is a well-studied

Algorithm 12: All-lengths gapped kernel of a string against a set of strings

Input: String set $S = \{s_1, \ldots, s_{|S|}\}$; string t; gap penalty $\lambda \in \mathbb{R}_+$.

Output: All-lengths gap-weighted similarities $K \in \mathbb{R}_+^{|S|}$

```
root = buildTrie(S);
root.DPSS(1 : |t|) = 0;
root.DPV(0 : |t|) = 0;
for r ∈ preOrder(root) do
    r.DPSS(1 : |t|) = 0;
    r.DPV(0) = 0;
    for j = 1 : |t| do
        if r.key = t_j then
            DPSij ← 1 + r.parent.DPV(j − 1);
            r.DPSS(i, j) ← r.DPSS(i, j) + DPSij;
        else
            DPSij ← 0;
        end
        r.DPV(i, j) ←
        DPSij + λ(r.DPV(j − 1) + r.parent.DPV(j)) − λ²r.parent.DPV(j − 1);
    end
    r.K ← r.parent.K + r.DPSS(i, |t|);
end
return CollectKFromLeaves(root);
```

problem with efficient algorithms. Incidentally, a good lexicographical sorting method relies on a trie [123, Vol. 3, Ch. 5]. After sorting, consecutive strings in T will always have the longest possible common prefix. If we then use information about the common prefix of the current and last string in T, we can only compute a fraction of the columns in the $r.DPSS$ and $r.DPV$ at each pass through the trie. Algorithm DYNTRIE 13 (page 118) realizes this idea.

A few details about Algorithm 13 are worth noting. Instead of yielding a matrix $K_{ij} \in \mathbb{R}_+^{|S|\times|T|}$, the algorithm writes the results sequentially to a tape. This is to emphasize that the output is incremental and there is no need to hold the entire output in memory, which is an important detail because otherwise the memory consumption of the algorithm would be considerably higher. Discounting the tape, the actual memory requirements of the algorithm is $\mathcal{O}(N \cdot \max_T |t|)$, where N is the number of nodes in the trie. (The trie's management overhead amounts to a constant factor.) Had the algorithm used an output matrix, that would have taken additional $\mathcal{O}(|S| \cdot |T|)$ space unless additional measures are taken to make the matrix sparse. Using a tape for output clarifies that no additional memory is needed beyond the trie.

5.9.3 Collecting Results

For graph construction we are interested in the strongest edges, i.e., the largest normalized similarities. To efficiently collect the highest kernel values, we use a classic top-N-copy algorithm that

Algorithm 13: DYNTRIE: All-lengths gapped kernel of a set of strings against a set of strings

Input: String set $S = \{s_1, \ldots, s_{|S|}\}$; string set $T = \{t_1, \ldots, t_{|T|}\}$; gap penalty $\lambda \in \mathbb{R}_+$; output tape τ.
Output: All-lengths gap-weighted similarities $K \in \mathbb{R}_+^{|S|\times|T|}$ are written to tape τ.

```
root = buildTrie(S);
root.DPSS(1 : |t|) = 0;
root.DPV(0 : |t|) = 0;
t^prev = ϵ;
for t ∈ LexicographicalSort(T) do
    l ← CommonPrefixLength(t^prev, t);
    t^prev ← t;
    for r ∈ preOrder(root) do
        if l > 0 then
            r.DPSS(l + 1 : |t|) = r.DPSS(l);
        else
            r.DPSS(1 : |t|) = 0;
            r.DPV(0) = 0;
        end
        for j = l + 1 : |t| do
            if r.key = t_j then
                DPSij ← 1 + r.parent.DPV(j − 1);
                r.DPSS(i, j) ← r.DPSS(i, j) + DPSij;
            else
                DPSij ← 0;
            end
            r.DPV(i, j) ← DPSij + λ(r.DPV(j − 1) + r.parent.DPV(j)) − λ²r.parent.DPV(j − 1);
        end
        r.K ← r.parent.K + r.DPSS(i, |t|);
        if r.IsLeaf then Write(τ, r.string, t, r.K);
    end
end
```

uses a binary heap [52] to efficiently store the best similarities seen so far. Algorithm 14 (page 119) shows the heap-based algorithm that is connected to the output tape τ of Algorithm 13.

The complexity of the top-N-copy algorithm is $\mathcal{O}(|\tau|\cdot\log N)$, where $|\tau|$ is the length of the input tape. The dominant operation inside the loop (assuming $N \ll |S|\times|T|$) is the *replaceTop* operation which takes time logarithmic in N. As discussed, the self-similarities $\kappa(s_i, s_i)$ and $\kappa(t_j, t_j)$ needed for normalization are computed once and kept separately.

5.9.4 Complexity

One individual kernel evaluation against strings s and t takes $\mathcal{O}(|s|\cdot|t|)$ elementary operations. A brute force evaluation against two sets of strings S and T therefore has time complexity:

$$\mathcal{C}_{\text{brute}}(S,T) \triangleq \mathcal{O}\left(\sum_{i=1}^{|S|}\sum_{j=1}^{|T|}|s_i|\cdot|t_j|\right) = \mathcal{O}\left(\left(\sum_{i=1}^{|S|}|s_i|\right)\cdot\left(\sum_{j=1}^{|T|}|t_j|\right)\right) \tag{5.59}$$

Algorithm 14: Obtaining the top kernel values.

Input: Input tape τ; Maximum values kept N; Criterion function $BetterThan$.
Output: Array of largest similarities h.

$h \leftarrow makeEmptyHeap$;
for $\langle\langle s, t, \kappa(s,t)\rangle\rangle \in Read(\tau)$ **do**
  $\hat{\kappa} \leftarrow \dfrac{\kappa(s,t)}{\sqrt{\kappa(s,s)\kappa(t,t)}}$;
  if $h.size < N$ **then**
    $h.push(\langle\langle s, t, \hat{\kappa}\rangle\rangle)$;
  else
    if $betterThan(\langle\langle s, t, \hat{\kappa}\rangle\rangle, top(h))$ **then**
      $replaceTop(h, \langle\langle s, t, \hat{\kappa}\rangle\rangle)$;
    end
  end
end
return h;

To calculate the complexity of DYNTRIE, let us introduce an auxiliary function:

$$\text{prefixes} : \mathcal{F}(\Sigma^*) \times \mathbb{N}^* \to \mathbb{N}^* \tag{5.60}$$

$$\text{prefixes}(A, n) = \text{card}\left(\{x \in \Sigma^n \mid \exists a \in A, a(1:n) = x\}\right) \tag{5.61}$$

where $a(1:n)$ is the substring from 1 to n of string a, and $\mathcal{F}(X)$ (also defined in Eq. 4.6) is the finite power set of some set X:

$$\mathcal{F}(X) = \{A \in \mathcal{P}(X) \mid \text{card}(A) < \infty\} \tag{5.62}$$

Colloquially, $\text{prefixes}(S, n)$ is the number of distinct prefixes of length n in string set S. When Algorithm 13 executes, at each depth i in the trie it will compute $\text{prefixes}(S, i)$ rows for the matrices $DPSS$ and DPV. However, not all columns are computed every pass; the first column is computed $\text{prefixes}(T, 1)$ times, the second is computed $p(T, 2)$ times, ..., the j^{th} column is computed $\text{prefixes}(T, j)$ times. So the number of elementary operations at depth i is

$$o_i = \text{prefixes}(S, i) \sum_{j \geq 1} \text{prefixes}(T, j) \tag{5.63}$$

Summing over all levels we obtain the overall complexity:

$$\mathcal{C}_{\text{DYNTRIE}}(S, T) = \mathcal{O}\left(\left(\sum_{i \geq 1} \text{prefixes}(S, i)\right) \cdot \left(\sum_{j \geq 1} \text{prefixes}(T, j)\right)\right) \tag{5.64}$$

It is trivially shown that the two sums are in fact equal to the number of nodes in the tries that would be built out of S and T (excluding the root node). This is in keeping with intuition: the

more prefixes are shared, the more compact the tries are, and the more computation can be saved compared to the brute force approach.

The worst-case complexity is attained when no two strings share the same prefix and is the same as the complexity of the brute-force approach. If vocabulary size is taken into account, another bound exists because there can be no more than $|\Sigma|^n$ distinct prefixes of length n, which limits the sum of $\mathrm{prefixes}$ in a set S to no more than $|\Sigma|^{\max_{s \in S} |s|}$; however, the exponential nature of that possible bound makes it inoperative beyond very small vocabularies and very short strings.

5.9.5 System-level Optimizations

Practical algorithm implementations must not only faithfully follow the definition of the algorithm, but should also account for the many details that can influence speed and memory consumption, sometimes to a surprisingly large extent or even subverting the algorithm's theoretical complexity.[5]

We implemented the all-strings gap-weighted kernel algorithm for the Cartesian product of two sets (Algorithm 13) and carried timing measurements against the hypotheses sets in a real medium-sized corpus for Machine Translation, Europarl [124]. The next section describes in detail the experimental setup. For now, we show how various system-level optimizations influenced the final timings of the implementation of the proposed approach in Table 5.5.

#	Optimization	Improvement
1	Mostly contiguous allocation of the trie nodes	6%
2	Avoid reallocation (don't shrink, keep the largest blocks allocated so far)	27%
3	Use one vector of pairs instead of two vectors for DPV, $DPSS$	12%
4	Use unchecked pointers instead of indexed access in the inner loop	6%
5	Cache on the stack all indirectly-accessed values in the inner loop	5%
	Total reduction in run time by	**56%**

Table 5.5: System-level optimizations in implementing Algorithm 13 and their influence on the timing results. The optimizations have been applied in the order shown, so optimizations towards the bottom may experience a diminished effect. The percents shown are absolute run time improvements compared to the unoptimized implementation of the same algorithm.

The improvements are highly system-dependent and we present them for informative purposes only. It is likely that on a different system the relative participation of each optimization would differ. Also, changing the order in which optimizations are applied would lead to different percentages. For example, optimization #5 brings a 5% absolute improvement when all other optimizations are already in effect; measuring its effect before all others may improve its measured participation level.

The section below compares the proposed algorithm against a brute-force evaluation of $|S| \cdot |T|$ kernel values. It should be noted that all of the above optimizations have also been carried

[5] A classic example is a $\mathcal{O}(n)$ loop transformed into an $\mathcal{O}(n^2)$ one by a poor implementation of an array append operation that is system-provided and assumed to be correct.

in the brute-force implementation where applicable, including two others that are not available to the trie-based version: (a) keeping only the last row of DPV and eliminating DPS entirely (see Algorithm 10); and (b) swapping inputs appropriately such that the inner loop always operates on the shorter of the two strings. Combined, these two measures lead to a memory consumption of only $\mathcal{O}(\min(|s|, |t|))$ for the brute force algorithm.

5.9.6 Timing Measurements

To gauge the improvements brought by the proposed method, we timed the kernel computations on hypotheses in the Europarl [124] corpus, starting from the same setup as that described in § 5.9.1.3. We generated up to 100-best hypotheses per chunk, resulting in an average of 72.8 hypotheses for each chunk. (Short sentences have fewer than 100 hypotheses.) Only unique (distinct) hypotheses have been generated for each hypothesis set; duplicated hypotheses would unfairly favor the proposed approach because the incremental cost of kernel evaluation for duplicated sentences is null (which is nonetheless an important property of the DynTrie algorithm). All things considered, about 20.7 million distinct kernel evaluations would need to be made if the Cartesian product of all hypothesis pairs would be evaluated. Practical approaches would avoid such computation by, for example, only computing the Cartesian product for hypotheses that are sufficiently similar on the source side and consider the rest dissimilar. However, the savings of the DynTrie method have effect for each pair of hypothesis sets, so the comparison is meaningful.

We measured the time to completion of a brute-force approach against the proposed algorithm. Sorting the input and normalization were not considered part of the process and were not timed. However, the time needed to build the trie was included in the timing of DynTrie, and collection of the top hypotheses using Algorithm 14 (the binary heap-based top-N-copy) was considered part of the process and was included in both timings.

The plot in Figure 5.2 (next page) reveals considerable improvements brought by the proposed algorithm for all input sizes. Figure 5.3 (page 123) displays the improvement factor of the proposed approach over the brute-force implementation. The improvements stay in the 3x range and do not degrade for large values of N. We should note, however, that this experiment is somewhat favorable to the trie-based approach: hypothesis sets are highly similar (albeit never identical) so they are likely to share prefixes more than e.g. randomly-chosen sentences. The proposed approach would not yield notable improvements if there is no significant prefix sharing across inputs (e.g. short strings randomly drawn from a large alphabet.)

5.9.7 Considerations on Parallelization

The brute-force approach has an obvious path towards parallelization—simply divide either or both sides of the computation in batches and deliver them to separate computation units. Each of these deposits results in a synchronized queue that feeds a top-N-copy collector.

The trie-based approach is also parallelizable. A naïve approach would be to exploit the property that at any branching point in the trie, there is no data sharing below it. Therefore, computation can be forked onto different units at any branching point in the trie. However, the subtrees resulting after branching can be very unequal in size, leading to an uneven distribution of computation. Furthermore, once a computing unit is done, there is no obvious point at which it could restart work on a different part of the trie.

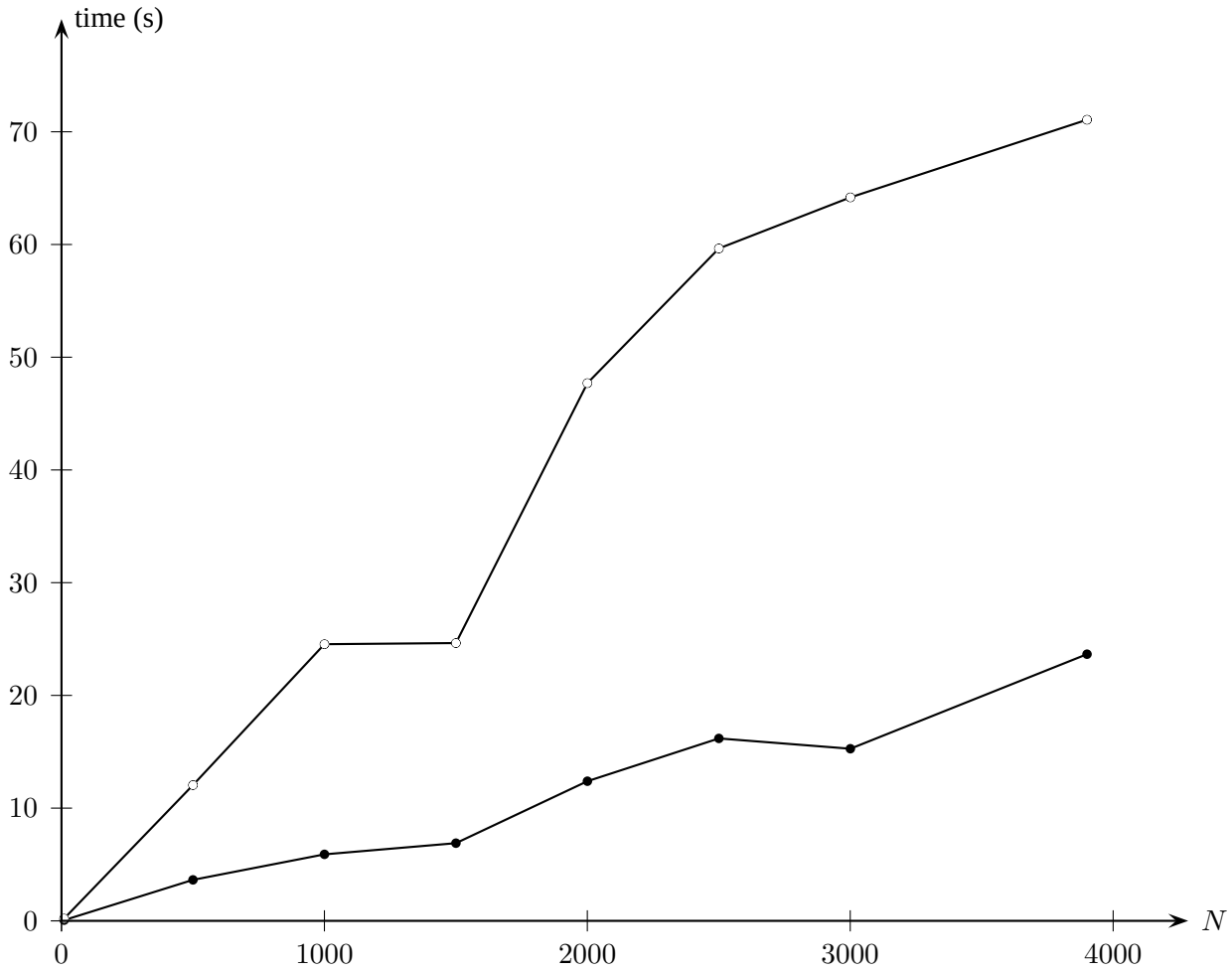


Figure 5.2: Timing comparison of brute force kernel computation (hollow dots) vs. trie-based dynamic programming computation (full dots). The graph displays the time to completion for comparing one hypothesis set consisting of 73 hypotheses on one side, against N hypothesis sets on the other side. The average number of hypotheses per set is 72.8.

A worklist-based approach is better suited: initially, the root's children are put in a worklist containing trie nodes. Each computing unit takes one node off the worklist, calculates that node's DPV, $DPSS$, and K, and puts that node's children back onto the list (save perhaps for one so it can continue computing without consulting the worklist). Once a computing unit is done, it again fetches any node off the worklist and resumes work. That way the worklist is continuously populated with nodes in the trie for which kernel computation can immediately proceed (as the parent computation has finished). Computation has finished when all threads are idle and the worklist is empty.

The worklist must be properly synchronized, but the overhead on contemporary architectures is low; the order of processing worklist items does not matter and singly-linked lists with prepending as the fundamental insertion operation can be implemented with lock-free guarantees [220, 81].

A different approach to parallelization can exploit characteristics of the data set used. For ex-

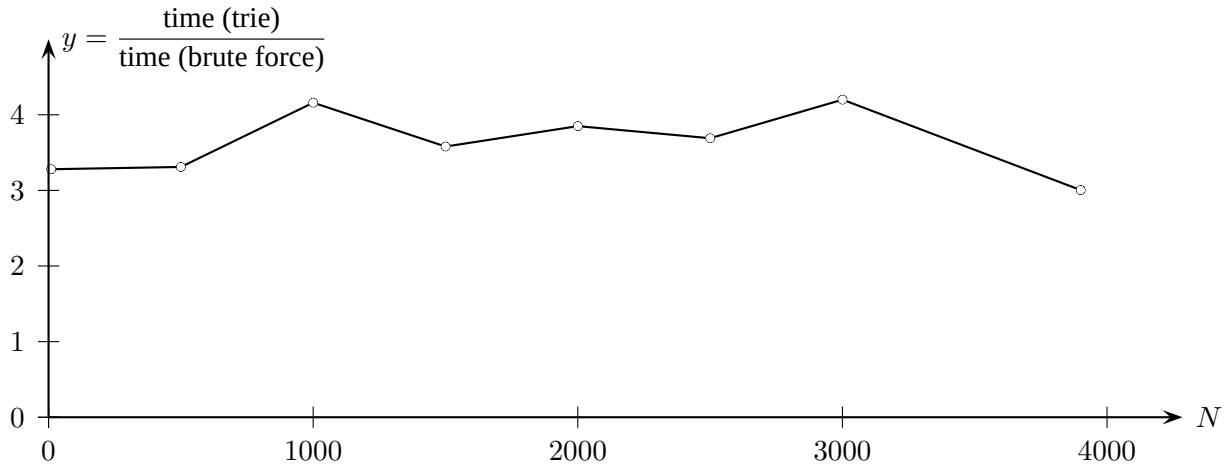


Figure 5.3: The variation of the improvement factor of the proposed algorithm over a brute force implementation on the same experiment as in Fig. 5.2.

ample, in the SMT scenario, the hypothesis sets provide a natural means of batching data. Also, the batching is highly effective because sentences in a hypothesis sets tend to be similar. Our approach is to build one trie out of each hypothesis set and distribute its computation to one processing unit.

5.10 Batching via Path Closures for GBL with Structured Inputs and Outputs

We describe below a method for reducing graph sizes with no or small loss in accuracy for graph-based learning with structured inputs and outputs following the formalism presented in Chapter 4. The reduction is important when there are very large amounts of unlabeled data and memory consumption becomes a concern. Our proposed solution trades consumed memory for computation; instead of a large graph it builds and uses several smaller graphs, which contain different portions of interest of the large graph. Depending on the original graph's connectivity, there could be no loss or a controllable tradeoff between loss and occupied memory.

Recall from Chapter 4 that the size of a fully constructed graph is $\mathtt{u} \cdot \overline{h} + 2$, where $\overline{h}$ is the average number of hypotheses per unlabeled sample. We have partly solved the size problem already by having all train data occupying only two vertices in the in-core graph, so the size of the representation is essentially independent of the training set size. We still need to take measures when scaling up the approach to large test sets. The number of hypotheses is to some extent controllable, but if $\mathtt{u}$ is large there is the risk that the graph becomes too large to be manageable. This creates the need for *batching*, i.e., devising a means to compute scores on one subset of the unlabeled set at a given time. That way several smaller graphs are used instead of a large one.

A principled way to achieve a good semi-supervised effect without operating on the entire graph at once is to work only on one test sample's hypotheses at any given time. We keep only the subgraph of interest for that test sample, which needs only to include the vertices reachable from that test sample's hypotheses. It is worth noting that reducing the graph does not change the graph, so the learning process is still global; only the portions of the graph not relevant to computing certain scores are removed. We will formally prove that below, but first let us define a path in a graph as a

sequence of distinct connected vertices that links two given vertices.

Definition 5.10.1. Given the undirected graph (V, E), a *path* between vertices $v \in V$ and $v' \in V$ is a sequence of vertices $\langle\langle v, v_1, \ldots, v_n, v' \rangle\rangle$ satisfying:

$$\{v, v_1\}, \{v_n, v'\} \in E \qquad \text{(connected start/endpoint)} \qquad (5.65)$$

$$\{v_{i-1}, v_i\} \in E \; \forall i \in \{2, \ldots, n\} \qquad \text{(connected consecutive vertices)} \qquad (5.66)$$

$$i \neq j \Leftrightarrow v_i \neq v_j \; \forall i, j \in \{1, \ldots, n\} \qquad \text{(distinct inner vertices)} \qquad (5.67)$$

$$v, v' \notin \{v_1, \ldots, v_n\} \qquad \text{(inner vertices distinct from start/endpoint)} \qquad (5.68)$$

The sequence $\langle\langle v, v' \rangle\rangle$ is also a path between v and v' if and only if $\{v, v'\} \in E$. A path is a *cycle* if $v = v'$ and *acyclic* otherwise. We denote the set of all paths between the two nodes as $\mathrm{Paths}_{(V,E)}(v, v')$.

Our study is only concerned with acyclic paths, but the definition above allows cycles in order to stay in keeping with the definition of "path" in established graph terminology and thus avoid confusion. Acyclic paths may consolidate Eq. 5.65 with Eq. 5.66 and Eq. 5.67 with Eq. 5.68.

Definition 5.10.2. Given the undirected graph (V, E) and a subset $V' \subseteq V$, we denote $(V, E) \setminus V'$ as the graph obtained from (V, E) after removing all $v \in V'$ and all edges that have at least one end in V':

$$(V, E) \setminus V' \triangleq \left(V \setminus V', \left\{\{v, v'\} \in E \mid v \notin V' \wedge v' \notin V'\right\}\right) \qquad (5.69)$$

Theorem 5.10.3. *Consider a similarity graph (V, E) constructed as per Definition 5.5.1 for the structured learning problem defined by features $\mathtt{X} = \langle\langle \mathtt{x}_1, \ldots, \mathtt{x}_{\mathtt{t}+\mathtt{u}} \rangle\rangle \subseteq \mathcal{X}$, training labels $\mathtt{Y} = \langle\langle \mathtt{y}_1, \ldots, \mathtt{y}_{\mathtt{t}} \rangle\rangle \subseteq \mathcal{Y}$, similarity function $\sigma : (\mathcal{X} \times \mathcal{Y}) \times (\mathcal{X} \times \mathcal{Y}) \to [0, 1]$, and hypothesis generator function $\chi : \mathcal{X} \to \mathcal{F}(\mathcal{Y})$. Given vertices $v, v' \in V \setminus \{v_+, v_-\}$ with $v \neq v'$, if $\mathrm{Paths}_{(V,E)\setminus\{v_+,v_-\}}(v, v') = \emptyset$, then removing vertex v' from the graph does not affect $s(v)$ computed by label propagation.*

Proof (by contradiction). Assume that the score computed for v in the graph $(V, E) \setminus \{v'\}$ is different from the score computed for v in the graph (V, E). Then, under the random walk interpretation of label propagation, this means there is at least one path from v to either v_+ or v_- passing through v'. That path influences the probability of the random walk starting at v and ending in v_+ or v_-, and hence the score $s(v)$. Then the sub-path from v up to v', which does not include either v_+ or v_- (by the definition of a path), contradicts the hypothesis that $\mathrm{Paths}_{(V,E)\setminus\{v_+,v_-\}}(v, v') = \emptyset$. □

We are now in the position of defining a smaller graph on which to compute scores for one given hypothesis. The unlabeled vertices needed for the precise score computation of hypotheses $\chi(\mathtt{x})$ are exactly those for which a path exists from some hypothesis to them. We formalize that set as a *path closure*.

Definition 5.10.4. Given the undirected graph (V, E) and a subset of its vertices $V' \subseteq V$, we define the *path closure of* (V, E) *over* V' as the graph $\mathrm{Paths}^*_{(V,E)}(V') \triangleq (V'', E'')$, where:

$$V'' = \left\{v \in V \mid \exists v' \in V', \mathrm{Paths}(E, v, v') \neq \emptyset\right\} \qquad (5.70)$$

$$E'' = \left\{\{v, v'\} \in E \mid v, v' \in V''\right\} \qquad (5.71)$$

Corollary 5.10.5. *To compute correct scores for the hypotheses of sample* $\mathtt{x}_i$ *in the similarity graph* (V, E)*, the subgraph* $\mathrm{Paths}^*_{(V,E)\setminus\{v_+,v_-\}}(\chi(\mathtt{x}_i))$ *is sufficient.*

Proof. Immediate from Definition 5.10.4 and Theorem 5.10.3. After removing (without affecting scores) all vertices in (V, E) with no paths from some hypothesis in $\chi(\mathtt{x}_i)$, what is left is by definition the path closure of $\chi(\mathtt{x}_i)$. □

So the transitive closure of the edge set over a subset of vertices $V' \subseteq V$ is the smallest component of the original graph (V, E) containing all vertices reachable from some vertex in V'. This smaller graph does not affect the outcome of the learning process for the focal test sample. In the worst theoretical case, the path closure could comprehend the entire graph, but in practice the edge set is almost never that dense. To counter for the possible worst-case scenario, we use a cutoff C that limits the number of vertices in the subgraph. The vertex set is computed starting from the vertices of the hypothesis and expands from there. This growth strategy is based on the heuristic that faraway nodes connected through low-weight edges have less influence on the result. We use a simple embodiment of this heuristic in a work-list approach implemented by Algorithm 15.

Algorithm 15: Batching via Path Closure with Cutoff

Input: Focal sample $\mathtt{x}_f$, its hypotheses $\chi(\mathtt{x}_f)$, edge set E, and cutoff $C \in \mathbb{N}^*$.
Output: Graph (V_f, E_f) for the similarity graph dedicated to computing scores for hypotheses of $\mathtt{x}_f$.

$V_f \leftarrow \{\langle\!\langle \mathtt{x}_f, y\rangle\!\rangle \mid y \in \chi(\mathtt{x}_f)\} \cup \{v_+, v_-\}$;
$E_f \leftarrow \{\{v, v'\} \in E \mid v, v' \in V_f\}$;
$c \leftarrow$ **true**;
while c **do**
    $c \leftarrow$ **false**;
    foreach $\{v', v''\} \in E$ **do**
        if $v' \in V_f \wedge v'' \notin V_f \cup \{v_+, v_-\}$ **then**
            $V_f \leftarrow V_f \cup \{v''\}$;
            $E_f \leftarrow E_f \cup \{v', v''\}$;
            $c \leftarrow$ **true**;
        else if $v'' \in V_f \wedge v' \notin V_f \cup \{v_+, v_-\}$ **then**
            $V_f \leftarrow V_f \cup \{v'\}$;
            $E_f \leftarrow E_f \cup \{v', v''\}$;
            $c \leftarrow$ **true**;
        end
        if $\mathrm{card}(V_f) = C$ **then**
            break while;
        end
    end
end
return (V_f, E_f);

Starting from the nodes of interest (hypotheses for the focal sentence), we expand the closure starting with the direct neighbors, which have the largest influence; then add their neighbors, which have less influence, and so forth. A threshold C on the number of added vertices limits undue expansion while capturing either the entire closure or an approximation of it. The algorithm makes iteration over the edge set E explicit, to clarify that E does not have to reside in core memory at any point throughout the algorithm.

Another practical computational advantage of portioning work in batches is that graphs for different hypothesis sets can be trivially created and used in parallel, whereas distributing large matrix-vector multiplication is much more difficult [48]. The disadvantage is that overall redundant computations are being made: incomplete estimates of s are computed for the ancillary nodes in the transitive closure and then discarded.

Chapter 6

CONCLUSIONS

This dissertation has shown that Machine Learning methods based on global similarity graphs can be used successfully against realistically-sized Human Language Technology tasks addressing problems in Natural Language Processing, Automatic Speech Recognition, and Machine Translation.

We have addressed a number of challenges in applying graph-based learning to HLT tasks. We summarize our contributions below.

Two-pass classifier for unstructured classification To address the heterogeneous, mixed, high-dimensional nature of features in unstructured HLT classification problems, we have introduced a two-pass system (Chapter 3). A first-pass classifier, which can be chosen to better suit the nature of the features, serves as a feature transformation mechanism. In the proposed setup, interestingly, the graph-based learner operates on the same space for input and output: probability distribution space. The input space is organized using a distance measure, which is easier to choose than a distance in the original heterogeneous feature space. We have experimentally confirmed that Jensen-Shannon divergence is the best distance measure to use in a variety of HLT applications. Furthermore, Jensen-Shannon divergence enjoys mathematical properties that make it suitable for fast nearest neighbor algorithms. We have proved that Jensen-Shannon divergence fulfills the requirements for being used with the kd-trees fast searching data structure, and implemented it measuring a speed gain of two orders of magnitude in Chapter 5. Metric-based search structures can be also used because Jensen-Shannon divergence is the square of a metric. We illustrate data-driven graph construction with experiments on lexicon learning, word sense disambiguation (both in Chapter 3), and phone classification (Chapter 5).

Structured learning through regression with kernel functions The formalization is widely applicable and relies on a hypothesis generator function χ (e.g. a generative learner with good recall and low precision) and a real-valued similarity function σ that returns a real number comparing two input/output pairs for similarity. An important category of similarity functions are kernel functions, among which string kernels are of particular interest to HLT applications. We demonstrate an application of graph-based learning with string kernels for Machine Translation.

Scalability A common theme in application of graph-based learning to large tasks is scalability. Graphs require the entire data set (training plus test) to be resident in working memory and connected through similarity edges. This proposition raises obvious scalability concerns in terms of sheer size and also in terms of time required to build the graph and then to run label propagation to completion. Naturally, scalability is an important focus of our work. We attack the scalability problem on all fronts.

Graph Construction As far as the graph size is concerned, we prove and implement a graph reduction technique that reduces the labeled sample size to one vertex per distinct label (§ 5.4) without affecting learning results. This reduction has a huge positive impact on both working set size and learning time. Also, we model additional information source without adding extra vertices by only manipulating edge weights. This technique effects density gradients without adding to the size of the graph (§ 3.8.1). In addition, as mentioned above, the use of a two-pass classifier allows us to use probability divergence measures with good properties, conducive to use of fast nearest-neighbors algorithms (such as our choice, kd-trees). For structured learning, discrete algorithms are an alternative to nearest-neighbors algorithms. We propose an algorithm called DYNTRIE, which combines traditional matrix-based dynamic programming with the trie data structure to mark additional savings in duplicate computations when computing cross-product kernel similarities over two sets of strings. Experiments with MT data show that the proposed method is three times faster than existing approaches.

Learning Speed To improve propagation speed, we introduce (§ 5.3.1) an in-place label propagation algorithm that uses an improved model parameter as soon as it was computed, as opposed to computing an entire batch of improved parameters in one epoch. Compared with the classic iterative algorithm, in-place propagation consumes half the memory and is faster (experimentally converges in roughly one third of the number of steps). We also provide the theoretical proof and implementation sketch of a multicore label propagation algorithm that uses parallel processing and benign data races to distribute work on label propagation. The number of cores can be arbitrarily high, up to the number of unlabeled samples. In our experiments, graph construction has always dominated total learning time, so improving propagation proper might seem of secondary interest. However, continuous learning systems would derive a large benefit from improved propagation times.

6.1 Future Directions

We see several directions in which our work can be continued and extended. One would concern improving the learning process *per se*, regardless of the problem it is being applied to. The two-pass classifier is currently trained in an open loop, i.e. there is no feedback from the graph-based engine to the first-pass classification engine. We do recognize that smoothness of the distributions of the first-pass classifier is essential for the good functioning of the graph-based learner and we regularize the first-pass classifier accordingly, but we believe that a closed-loop, joint training of the two classifiers would be closer to optimal. A simple example would be to optimize a neural network learner by introducing smoothness in the epoch-level decision on keeping or reducing the learning rate of the network. A more direct coupling is to offer back-propagation information with errors output by the graph-based learner, not (only) the neural network proper. That approach would work directly on minimizing the bottom-line goal.

Using other kernels than the Gaussian kernel (§ 3.2) or string kernels (§ 4.3.1.3) for computing similarity is a direction worth exploring. Especially when HLT applications with structured data are concerned, the option of using tree and graph kernels (§ 4.3) is very attractive; trees and graphs naturally occur in linguistics (e.g. syntax trees or semantic graphs). Using such kernels would put an even higher emphasis on scalability and efficiency. It may be worth exploring extending the DYNTRIE algorithm to tree or graph matching, and also combining it with approximation bounds

for obtaining fast approximate matches. Using other nearest-neighbor techniques aside from kd-trees are a possible direction in exploring scalability.

As we have already hinted above, the fast convergence time obtained by the proposed algorithm suggests applicability to continuous learning systems and incremental learners where results are needed at the same rate as input samples. Systems can be envisioned that maintain a fixed-size graph with historical samples and their connections, that changes slowly as new samples are seen and old samples are discarded.

Finally, applications far removed from HLT can be attempted for scalable graph-based learning. The battery of proposed techniques extend applicability of graph-based learning beyond problems in which a notion of similarity could be easily defined.

Appendix A

TWO THEORETICAL BOUNDS FOR SPEED OF CONVERGENCE IN LABEL PROPAGATION

We have computed two theoretical results that put upper bounds on the number of steps to convergence within a given tolerance τ without actually running the label propagation algorithms. Our implementation does not use these bounds but they may be useful for graph analysis and for improving the graph construction step.

What constitutes a "good" matrix $\mathtt{P_{UU}}$ that leads to quick convergence, and what bounds can be derived about the number of steps to convergence? We compute such bounds depending on features of $\mathtt{P_{UU}}$. Our practical implementations do not use these theoretical bounds, but they are useful to assess the quality of a graph before performing iterative label propagation against it. We also hope that this will inspire future work aimed at finding tighter bounds.

Let us recall the iteration core

$$\mathtt{f_U} \leftarrow \mathtt{f'_U} \tag{A.1}$$

$$\mathtt{f'_U} \leftarrow \mathtt{P_{UU}}\mathtt{f_U} + \mathtt{P_{UL}}\mathtt{Y_L} \tag{A.2}$$

This reveals that iterative label propagation is a repeated application of the function

$$Q : [0,1]^{\mathtt{u}\times\ell} \rightarrow [0,1]^{\mathtt{u}\times\ell}, \quad Q(X) = \mathtt{P_{UU}}X + \mathtt{P_{UL}}\mathtt{Y_L} \tag{A.3}$$

The approach we will take to estimating the number of steps to convergence is to define a metric space over $[0,1]^{\mathtt{u}\times\ell}$ and then use the fixed point theorem [107, Ch. 7] to bound the steps to of convergence of Q. Let us endow the set $[0,1]^{\mathtt{u}\times\ell}$ with the distance measure

$$d_{\max}(A,B) : [0,1]^{\mathtt{u}\times\ell} \rightarrow \mathbb{R}_+ \tag{A.4}$$

$$d_{\max}(A,B) = \max_{\substack{i\in\{1,\ldots,\mathtt{u}\}\\ j\in\{1,\ldots,\ell\}}} |A_{ij} - B_{ij}| \tag{A.5}$$

It is trivial to verify that the space $S_{\max} = \left([0,1]^{\mathtt{u}\times\ell}, d_{\max}\right)$ verifies the conditions for being metric and complete. ($d_{\max}$ is in fact the Minkowski distance of infinite order.) This sets the stage for the following theorem.

Theorem A.1. *If* $\max_{i\in\{1,\ldots,\mathtt{u}\}} \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik} = \gamma_{\max} < 1$*, then function* Q *is a contraction in the space* $S_{\max} = \left([0,1]^{\mathtt{u}\times\ell}, d_{\max}\right)$.

Proof. To prove that Q is a contraction we need to show $\exists q \in (0,1)$ such that $d_{\max}(Q(A), Q(B)) \leq q \cdot d_{\max}(A, B)\ \forall A, B \in [0,1]^{\mathtt{u}\times\ell}$.

$$d_{\max}(Q(A), Q(B)) = \max_{i\in\{1,\ldots,\mathtt{u}\}} \max_{j\in\{1,\ldots,\ell\}} \left|(\mathtt{P_{UU}}A + \mathtt{P_{UL}}\mathtt{Y_L} - \mathtt{P_{UU}}B - \mathtt{P_{UL}}\mathtt{Y_L})_{ij}\right| \tag{A.6}$$

$$= \max_{i\in\{1,\ldots,\mathtt{u}\}} \max_{j\in\{1,\ldots,\ell\}} \left|[\mathtt{P_{UU}}\,(A-B)]_{ij}\right| \tag{A.7}$$

$$= \max_{i\in\{1,\ldots,\mathtt{u}\}} \max_{j\in\{1,\ldots,\ell\}} \left|\sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik}(A-B)_{kj}\right| \tag{A.8}$$

$$\leq \max_{i\in\{1,\ldots,\mathtt{u}\}} \max_{j\in\{1,\ldots,\ell\}} \sum_{k=1}^{\mathtt{u}} \left[(\mathtt{P_{UU}})_{ik}\left|(A-B)_{kj}\right|\right] \tag{A.9}$$

$$\leq \max_{i\in\{1,\ldots,\mathtt{u}\}} \max_{j\in\{1,\ldots,\ell\}} \sum_{k=1}^{\mathtt{u}} \left[(\mathtt{P_{UU}})_{ik} \max_{k\in\{1,\ldots,\mathtt{u}\}} |(A-B)_{kj}|\right] \tag{A.10}$$

$$= d_{\max}(A,B) \cdot \max_{i\in\{1,\ldots,\mathtt{u}\}} \sum_{k=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik} = \gamma_{\max} d_{\max}(A,B) \tag{A.11}$$

So Q is a contraction in a complete metric space, and the sought-after constant q is $\gamma_{\max}$. □

It follows by Banach's fixed point theorem [107, Ch. 7] that Q has a unique fixed point that can be reached by repeated application starting from an arbitrary element in $[0,1]^{\mathtt{u}\times\ell}$.

This result is similar to that of Theorem 2.3.1 obtained by Zhu [238] and with the same restriction on $\mathtt{P_{UU}}$, but this form provides a bound for the speed of convergence. If we denote $\mathtt{f_U}^{\text{step } 0}$ as the initial value of $\mathtt{f_U}$, $\mathtt{f_U}^{\text{step } t}$ as the value of $\mathtt{f_U}$ after the t^{th} step, and $\mathtt{f_U}^{\text{step } \infty}$ as the fixed point, then [107, Ch. 7]

$$d_{\max}(\mathtt{f_U}^{\text{step } \infty}, \mathtt{f_U}^{\text{step } t}) \leq \frac{\gamma_{\max}^t}{1-\gamma_{\max}} \cdot d_{\max}\left(\mathtt{f_U}^{\text{step } 1}, \mathtt{f_U}^{\text{step } 0}\right) \tag{A.12}$$

so at each step the distance from the solution decreases by at least a factor of $\gamma_{\max}$. Although $\mathtt{f_U}^{\text{step } 0}$ can be an arbitrary element in $[0,1]^{\mathtt{u}\times\ell}$, its choice does affect speed of convergence and monotonicity. Algorithm 1 chooses $\mathtt{f_U}^{\text{step } 0} = 0$, therefore

$$d_{\max}(\mathtt{f_U}^{\text{step } \infty}, \mathtt{f_U}^{\text{step t}}) \leq \frac{\gamma_{\max}^t}{1-\gamma_{\max}} \cdot \max_{\substack{i\in\{1,\ldots,\mathtt{u}\}\\ j\in\{1,\ldots,\ell\}}} (\mathtt{P_{UL}}\mathtt{Y_L})_{ij} \tag{A.13}$$

We can now get a bound on the number of steps to convergence for a label propagation algorithm that uses $d_{\max}(\mathtt{f_U}^{\text{step } t+1}, \mathtt{f_U}^{\text{step } t})$ as its termination condition with tolerance τ:

$$t \leq \log_{\gamma_{\max}} \frac{\tau(1-\gamma_{\max})}{\max\limits_{\substack{i\in\{1,\ldots,\mathtt{u}\}\\ j\in\{1,\ldots,\ell\}}} (\mathtt{P_{UL}}\mathtt{Y_L})_{ij}} \tag{A.14}$$

$$= \frac{\ln\left(\tau(1-\gamma_{\max})\right) - \ln \max\limits_{\substack{i\in\{1,\ldots,\mathtt{u}\}\\ j\in\{1,\ldots,\ell\}}} (\mathtt{P_{UL}}\mathtt{Y_L})_{ij}}{\ln \gamma_{\max}} \tag{A.15}$$

In fact, $\gamma_{\max}$ as just computed is also the lowest bound in the space $S_{\max} = \left([0,1]^{u\times\ell}, d_{\max}\right)$ for function Q, also called the Lipschitz constant [107]. This means that at least in this particular space, $\gamma_{\max}$ is the best bound on the convergence speed for Q.

Theorem A.2. *The bound $\gamma_{\max}$ is the Lipschitz constant for Q in space $S_{\max} = \left([0,1]^{u\times\ell}, d_{\max}\right)$.*

Proof. We will show that for certain values A and B, the inequalities A.9 and A.10 (page 131) turn into equalities. For equation A.9, the inequality becomes equality if $A_{kj} \geq B_{kj}\ \forall i \in \{1,\ldots,\mathtt{u}\}, j \in \{1,\ldots,\ell\}$. For equation A.10, the inequality becomes equality if matrix $A - B$ has all elements equal to one another. So

$$A - B = a^{\mathtt{u}\times\ell} \Rightarrow d_{\max}(Q(A), Q(B)) = \gamma_{\max} \cdot d_{\max}(A, B) = a \tag{A.16}$$

which concludes the proof because any choice smaller than $\gamma_{\max}$ would invalidate the inequality. □

In order to achieve rapid convergence, a small $\gamma_{\max}$ and a strong maximum element in $\mathtt{P_{UL}}$ are desirable; both describe, unsurprisingly, a graph that has strong connections between labeled and unlabeled nodes.

One problem with the bound computed above is that the restriction on $\mathtt{P_{UU}}$ is quite harsh: each unlabeled point must be directly connected to at least one labeled point such that after normalization, the total weight connecting it to other unlabeled nodes is strictly less than 1. It is worth searching for a different theoretical bound. To that end, we define a different metric over the same matrix set:

$$d_\Sigma(A, B) = \sum_{i=1}^{\mathtt{u}} \sum_{j=1}^{\ell} |A_{ij} - B_{ij}| \tag{A.17}$$

The resulting space $S_\Sigma = \left([0,1]^{\mathtt{u}\times\ell}, d_\Sigma\right)$ allows a different bound and a different restriction on $\mathtt{P_{UU}}$. This time sums over columns (as opposed to rows) of elements in $\mathtt{P_{UU}}$ are involved.

Theorem A.3. *If $\max_{i\in\{1,\ldots,\mathtt{u}\}} \sum_{i=1}^{\mathtt{u}} (\mathtt{P_{UU}})_{ik} = \gamma_\Sigma < 1$, then function Q is a contraction in space $S_\Sigma = \left([0,1]^{\mathtt{u}\times\ell}, d_\Sigma\right)$.*

Proof.

$$d_\Sigma(Q(A), Q(B)) = \sum_{i=1}^{\mathtt{u}} \sum_{j=1}^{\ell} \left| (\mathtt{P}_{\mathtt{UU}} A + \mathtt{P}_{\mathtt{UL}} \mathtt{Y}_{\mathtt{L}} - \mathtt{P}_{\mathtt{UU}} B - \mathtt{P}_{\mathtt{UL}} \mathtt{Y}_{\mathtt{L}})_{ij} \right| \tag{A.18}$$

$$= \sum_{i=1}^{\mathtt{u}} \sum_{j=1}^{\ell} \left| \left[\mathtt{P}_{\mathtt{UU}} (A - B) \right]_{ij} \right| \tag{A.19}$$

$$= \sum_{i=1}^{\mathtt{u}} \sum_{j=1}^{\ell} \left| \sum_{k=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} (A - B)_{kj} \right| \tag{A.20}$$

$$\leq \sum_{i=1}^{\mathtt{u}} \sum_{j=1}^{\ell} \sum_{k=1}^{\mathtt{u}} \left[(\mathtt{P}_{\mathtt{UU}})_{ik} \left| (A - B)_{kj} \right| \right] \tag{A.21}$$

$$= \sum_{j=1}^{\ell} \sum_{k=1}^{\mathtt{u}} \left[\left| (A - B)_{kj} \right| \sum_{i=1}^{\mathtt{u}} (\mathtt{P}_{\mathtt{UU}})_{ik} \right] \tag{A.22}$$

$$\leq \gamma_\Sigma d_\Sigma(A, B) \tag{A.23}$$

So Q is a contraction in S_Σ with γ_Σ as a bound for its contraction constant. □

We can derive similar bounds on speed of convergence and maximum number of steps for γ_Σ as we did for $\gamma_{\max}$. (However, γ_Σ is not easily shown as the Lipschitz constant.) Theorem A.1 tracks the largest error in each iteration, whereas Theorem A.3 characterizes global convergence by tracking the sum of all errors.

Appendix B

EXPONENTIAL SPEEDUP OF LABEL PROPAGATION

In the following we show how speed of convergence in the label propagation algorithm can be accelerated exponentially. Our experiments do not use this definition and instead use in-place label propagation which has a smaller working set. However, the algorithm below may be of interest when the graphs have relatively few vertices but are densely connected. On a given input, if original label propagation would converge in n steps, the algorithm presented below converges in approximately $\log n$ steps.

Let us consider an already reduced graph with ℓ labeled nodes and $\mathtt{u}$ unlabeled nodes, and define matrix S as follows:

$$S = \begin{bmatrix} \mathbb{1}^{\ell} & 0^{\ell \times \mathtt{u}} \\ \mathtt{P}_{\mathtt{UL}} & \mathtt{P}_{\mathtt{UU}} \end{bmatrix} \tag{B.1}$$

where $\mathbb{1}^{\ell}$ is the identity matrix of size ℓ and $0^{\ell \times \mathtt{u}}$ is a matrix of size $\ell \times \mathtt{u}$ containing zeros. It is easy to verify that raising S to the power of t yields

$$S^t = \begin{bmatrix} \mathbb{1}^{\ell} & 0^{\ell \times \mathtt{u}} \\ \left(\sum_{i=0}^{t} \mathtt{P}_{\mathtt{UU}}^t \right) \mathtt{P}_{\mathtt{UL}} & \mathtt{P}_{\mathtt{UU}} \end{bmatrix} \tag{B.2}$$

The bottom-left quadrant of S^t is exactly $\mathtt{f}_{\mathtt{U}}$ after t iterations of Zhu's label propagation algorithm starting from $\mathtt{f}_{\mathtt{U}} = 0$, as shown in eq. 2.8. This means computing powers of S is an alternate way of converging to the solution. Then the harmonic function would be the bottom-left quadrant of $S^{\infty} = \lim_{t \to \infty} S^t$. Such a way of implementing label propagation would not be more attractive for large data sets given that the matrices involved are larger, were it not for a simple but crucial observation: large powers of S can be computed exponentially faster by repeatedly squaring the intermediate result, as opposed to just multiplying the intermediate result by S. That way, by using t matrix multiplications, we can compute S^{2^t} instead of S^t—an exponential speedup.

$$S^{2^t} = \left(\left(\left(\left(S^2 \right)^2 \right)^2 \right)^2 \dots \right)^2 \tag{B.3}$$

This algorithm for computing large powers over a field was known as far back as ancient Egypt and is described in detail in Knuth's treatise [123, Vol. 2, pp. 465–481]. So the core iteration to convergence is

$$S \leftarrow S^2 \tag{B.4}$$

which can be rewritten as

$$S_{SW} \leftarrow (S_{SE} + \mathbb{1})S_{SW} \tag{B.5}$$

$$S_{SE} \leftarrow S_{SE}^2 \tag{B.6}$$

where S_{SW} is the bottom-left quadrant of S (initially $\mathtt{P}_{\mathtt{UL}}$) and S_{SE} is the bottom-right quadrant of S (initially $\mathtt{P}_{\mathtt{UU}}$). The cost of the exponential speedup is that, in addition to the matrix multiplication between a $\mathtt{u} \times \mathtt{u}$ matrix and a $\mathtt{u} \times \ell$ matrix (same cost as for the other algorithms), there is a need to also perform a squaring of a $\mathtt{u} \times \mathtt{u}$ matrix. If $\mathtt{P}_{\mathtt{UU}}$ is dense, in a straight implementation of matrix product, the complexity of the algorithm jumps from $\mathcal{O}(\ell \cdot \mathtt{u}^2)$ to $\mathcal{O}(\mathtt{u}^3)$, which is an important change because we pursue scalability across $\mathtt{u}$ while ℓ is often considered a constant. However, if $\mathtt{u}$ is relatively small or if $\mathtt{P}_{\mathtt{UU}}$ is sparse by using a nearest-neighbors method of graph construction, the benefits of exponential speedups can be enjoyed at an affordable extra cost per step.

BIBLIOGRAPHY

[1] Steven Abney. *Semisupervised Learning for Computational Linguistics*. Chapman & Hall/CRC, 2007. ISBN 978-1584885597.

[2] Marcel R. Ackermann, Johannes Blömer, and Christian Sohler. Clustering for metric and non-metric distance measures. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 799–808, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics. URL `http://portal.acm.org/citation.cfm?id=1347082.1347170`.

[3] Shivani Agarwal. Ranking on graph data. In *ICML '06: Proceedings of the 23rd international conference on Machine learning*, pages 25–32, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-383-2. URL `http://doi.acm.org/10.1145/1143844.1143848`.

[4] Joshua Albrecht and Rebecca Hwa. A re-examination of machine learning approaches for sentence-level MT evaluation. In *Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics*, pages 880–887, Prague, Czech Republic, June 2007. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P/P07/P07-1111`.

[5] E. Alpaydin and C. Kaynak. Cascading classifiers. *Kybernetika*, 34:369–374, 1998.

[6] Y. Altun, I. Tsochantaridis, T. Hofmann, et al. Hidden Markov Support Vector Machines. In *Machine Learning International Workshop then Conference*, volume 20, page 3, 2003.

[7] Y. Altun, D. McAllester, and M. Belkin. Maximum margin semi-supervised learning for structured variables. In *Proceedings of NIPS 18*, 2005.

[8] S. Arya and D. Mount. ANN: library for approximate nearest neighbor searching. URL `http://www.cs.umd.edu/~mount/ANN/`, 2005.

[9] S. Arya and D.M. Mount. *Algorithms for fast vector quantization*. University of Maryland, 1993.

[10] F. Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Computing surveys*, 23(3), 1991.

[11] R.G. Bartle. *The elements of integration and Lebesgue measure*. Wiley, 1995.

[12] M. Belkin, I. Matveeva, and P. Niyogi. Regularization and semi-supervised learning on large graphs. *COLT*, 2004. URL `http://www.cse.ohio-state.edu/tilde/mbelkin/reg_colt.pdf`.

[13] M. Belkin, P. Niyogi, and V. Sindhwani. On Manifold Regularization. *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTAT 2005)*, 2005.

[14] Y. Bengio, R. Ducharme, and P. Vincent. A neural probabilistic language model. In *NIPS*, 2000.

[15] Yoshua Bengio. Probabilistic neural network models for sequential data. In *IJCNN '00: Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN'00)-Volume 5*, pages 50–79, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0619-4.

[16] J.L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23:214–229, 1980.

[17] J.L. Bentley and J.H. Friedman. A Survey of Algorithms and Data Structures for Range Searching. *ACM Computing Surveys*, 11:397–409, 1979.

[18] J.L. Bentley and M.I. Shamos. Divide-and-conquer in multidimensional space. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 220–230. ACM New York, NY, USA, 1976.

[19] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975. URL `http://doi.acm.org/10.1145/361002.361007`.

[20] A.L. Berger, V.J. Della Pietra, and S.A. Della Pietra. A maximum entropy approach to natural language processing. *Computational Linguistics*, 22(1):39–71, 1996.

[21] P. Berkhin. Survey of clustering data mining techniques, 2002.

[22] P. Bernaola-Galvan, R. Roman-Roldan, and J.L. Oliver. Compositional segmentation and long-range fractal correlations in DNA sequences. *Physical Review E*, 53(5):5181–5189, 1996.

[23] P. Bernaola-Galván, I. Grosse, P. Carpena, J.L. Oliver, R. Román-Roldán, and H.E. Stanley. Finding borders between coding and noncoding DNA regions by an entropic segmentation method. *Physical Review Letters*, 85(6):1342–1345, 2000.

[24] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM New York, NY, USA, 2006.

[25] Paul E. Black. 'inverted index', from Dictionary of Algorithms and Data Structures, 2007. URL `http://www.nist.gov/dads/HTML/invertedIndex.html`.

[26] Paul E. Black. 'trie', from Dictionary of Algorithms and Data Structures, 2007. URL `http://www.nist.gov/dads/HTML/trie.html`.

[27] A. Blum and S. Chawla. Learning from labeled and unlabeled data using graph mincuts. *Proc. 18th International Conf. on Machine Learning*, pages 19–26, 2001.

[28] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM New York, NY, USA, 1998.

[29] Avrim Blum, John Lafferty, Mugizi Robert Rwebangira, and Rajashekar Reddy. Semi-supervised learning using randomized mincuts. In *ICML '04: Proceedings of the twenty-first international conference on Machine Learning*, page 13, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-828-5. URL `http://doi.acm.org/10.1145/1015330.1015429`.

[30] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, and R.E. Tarjan. Time bounds for selection. *JCSS*, 7(4):448–461, 1973.

[31] B.P. Bogert, MJR Healy, and J.W. Tukey. The quefrency alanysis of time series for echoes: Cepstrum, pseudo-autocovariance, cross-cepstrum and saphe cracking. In *Proceedings of the Symposium on Time Series Analysis*, pages 209–243, 1963.

[32] B.E. Boser, I. Guyon, and V. Vapnik. A Training Algorithm for Optimal Margin Classifiers. *Computational Learing Theory*, pages 144–152, 1992.

[33] H. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, 1994.

[34] A. Bowyer. Computing Dirichlet tessellations. *The Computer Journal*, 24(2):162–166, 1981.

[35] Thorsten Brants and Alex Franz. *Web 1T 5-gram Version 1*. Linguistic Data Consortium, Philadelphia, 2006. ISBN 1-58563-397-6.

[36] JS Bridle and MD Brown. An experimental automatic word recognition system. *JSRU Report*, 1003, 1974.

[37] J. Briët, P. Harremoës, and F. Topsøe. Properties of Classical and Quantum Jensen-Shannon Divergence. *ArXiv e-prints*, June 2008.

[38] C. Callison-Burch, M. Osborne, and P. Koehn. Re-evaluating the role of BLEU in machine translation research. In *Proceedings of EACL*, 2006.

[39] W.M. Campbell, J.P. Campbell, D.A. Reynolds, E. Singer, and P.A. Torres-Carrasquillo. Support vector machines for speaker and language recognition. *Computer Speech & Language*, 20(2-3):210–229, 2006.

[40] R. Cattoni, M. Danieli, V. Sandrini, and C. Soria. ADAM: the SI-TAL Corpus of Annotated Dialogues. In *Proceedings of LREC 2002, Las Palmas, Spain*, 2002.

[41] M. Cettolo and M. Federico. Minimum error training of log-linear translation models. In *Proc. of the International Workshop on Spoken Language Translation*, pages 103–106, 2004.

[42] O. Chapelle, B. Schölkopf, and A. Zien. *Semi-Supervised Learning*. MIT Press, 2006.

[43] Nick Chater and Paul Vitányi. The generalized universal law of generalization. *Journal of Mathematical Psychology*, 47:346–369, 2003.

[44] S. Chen, T. Gu, X. Tao, and J. Lu. Application based distance measurement for context retrieval in ubiquitous computing. In *Mobile and Ubiquitous Systems: Networking & Services, 2007. MobiQuitous 2007. Fourth Annual International Conference on*, pages 1–7, 2007.

[45] S.F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech and Language*, 13(4):359–394, 1999.

[46] K. Cheng. Shepard's Universal Law Supported by Honeybees in Spatial Generalization. *Psychological Science*, 11(5):403–408, 2000.

[47] David Chiang, Steve Deneefe, Yee S. Chan, and Hwee T. Ng. Decomposability of translation metrics for improved evaluation and efficient algorithms. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 610–619, Honolulu, Hawaii, October 2008. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/D08-1064`.

[48] Jaeyoung Choi. A new parallel matrix multiplication algorithm on distributed-memory concurrent computers. *Concurrency: Practice and Experience*, 10(8):655–670, 1998. URL `citeseer.ist.psu.edu/article/choi97new.html`.

[49] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the International Conference on Very Large Data Bases*, pages 426–435. Institute of Electrical & Electronics Engineers (IEEE), 1997.

[50] P. Clarkson and R. Rosenfeld. Statistical language modeling using the CMU-Cambridge toolkit. In *Fifth European Conference on Speech Communication and Technology*. ISCA, 1997.

[51] M. Collins and N. Duffy. Convolution kernels for natural language. *Advances in Neural Information Processing Systems*, 1:625–632, 2002.

[52] T.H. Cormen. *Introduction to Algorithms*. MIT press, 2001.

[53] C. Cortes, P. Haffner, and M. Mohri. Rational kernels: Theory and algorithms. *The Journal of Machine Learning Research*, 5:1035–1062, 2004.

[54] D. Coughlin. Correlating Automated and Human Assessments of Machine Translation Quality. In *Proceedings of MT Summit IX*, pages 63–70, 2001.

[55] T. Cover and P. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[56] F. Cozman, I. Cohen, and M. Cirelo. Semi-supervised learning of mixture models and bayesian networks. In *ICML '03: Proceedings of the twenty-second international conference on Machine Learning*, 2003. URL `http://citeseer.ist.psu.edu/cozman03semisupervised.html`.

[57] Fabio G. Cozman and Marcelo C. Cirelo. Semisupervised learning of classifiers: Theory, algorithms, and their application to human-computer interaction. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(12):1553–1567, 2004. ISSN 0162-8828. URL `http://dx.doi.org/10.1109/TPAMI.2004.127`.

[58] Fabio G. Cozman and Ira Cohen. Unlabeled data can degrade classification performance of generative classifiers. In *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, pages 327–331. AAAI Press, 2002. ISBN 1-57735-141-X.

[59] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines and Other Kernel-based Learning Methods*. Cambridge University Press, March 2000. ISBN 0521780195.

[60] A. Culotta and J. Sorensen. Dependency tree kernels for relation extraction. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics Morristown, NJ, USA, 2004.

[61] F.J. Damerau. A technique for computer detection and correction of spelling errors. *Communications of the ACM*, 7(3):171–176, 1964.

[62] Hal Daumé III. From Zero to Reproducing Kernel Hilbert Spaces in Twelve Pages or Less, 2004.

[63] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 28(4):357–366, 1980.

[64] O. Delalleau, Y. Bengio, and N. Le Roux. Efficient non-parametric function induction in semi-supervised learning. *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTAT 2005)*, 2005.

[65] Olivier Delalleau, Yoshua Bengio, and Nicolas Le Roux. Large-scale algorithms. In Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, editors, *Semi-Supervised Learning*, pages 333–341. MIT Press, 2006.

[66] J.R. Deller, J.G. Proakis, and J.H.L. Hansen. *Discrete-time processing of speech signals*. Macmillan New York, 1993.

[67] Guo dong Guo, Anil K. Jain, Wei ying Ma, Hong jiang Zhang, and Senior Member. Learning similarity measure for natural image retrieval with relevance feedback. *IEEE Transactions on Neural Networks*, 13:811–820, 2002.

[68] Peter G. Doyle and Laurie J. Snell. Random walks and electric networks, Jan 2000. URL `http://arxiv.org/abs/math.PR/0001057`.

[69] Kai-bo Duan and S. Sathiya Keerthi. Which is the best multiclass SVM method? An empirical study. In *Proceedings of the Sixth International Workshop on Multiple Classifier Systems*, pages 278–285, 2005.

[70] I.S. Duff, AM Erisman, and J.K. Reid. *Direct methods for sparse matrices*. Oxford University Press, USA, 1986.

[71] K. Duh and K. Kirchhoff. Beyond Log-Linear Models: Boosted Minimum Error Rate Training for N-best Re-ranking. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics (ACL), Columbus, Ohio, June*, 2008.

[72] David S. Dummit and Richard M. Foote. *Abstract Algebra*. Wiley, 3 edition, July 2003. ISBN 0471433349.

[73] Bernardt Duvenhage. Using an implicit min/max kd-tree for doing efficient terrain line of sight calculations. In *AFRIGRAPH '09: Proceedings of the 6th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, pages 81–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-428-7. URL `http://doi.acm.org/10.1145/1503454.1503469`.

[74] Raymond G. Gordon (editor). *Ethnologue: Languages of the World*. Summer Inst of Linguistics; 15th edition, 2005.

[75] G. Ekman. Dimensions of color vision. *Journal of Psychology*, 38:467–474, 1954.

[76] A. El Isbihani, S.K.O. Bender, and H. Ney. Morpho-syntactic Arabic Preprocessing for Arabic-to-English Statistical Machine Translation. In *Human Language Technology Conf./North American Chapter of the Assoc. for Computational Linguistics Annual Meeting (HLT-NAACL), Workshop on Statistical Machine Translation*, pages 15–22, 2006.

[77] D. M. Endres and J. E. Schindelin. A new metric for probability distributions. *Information Theory, IEEE Transactions on*, 49(7):1858–1860, 2003. URL `http://www.st-andrews.ac.uk/~dme2/density_metric.pdf`.

[78] Zheng-Yu Niu et al. Word sense disambiguation using label propagation based semi-supervised learning method. In *Proceedings of ACL*, pages 395–402, 2005.

[79] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. E cient and effective querying by image content. *Journal of Intelligent Information Systems*, 3 (3):231–262, 1994.

[80] D.A. Field. Implementing Watson's algorithm in three dimensions. In *Proceedings of the second annual symposium on Computational geometry*, pages 246–259. ACM New York, NY, USA, 1986.

[81] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM New York, NY, USA, 2004.

[82] S. Foo and H. Li. Chinese word segmentation and its effect on information retrieval. *Information Processing and Management*, 40(1):161–190, 2004.

[83] Cameron S. Fordyce. Overview of the IWSLT 2007 Evaluation Campaign. In *IWSLT*, Trento, Italy, October 2007.

[84] Robert Frederking and Sergei Nirenburg. Three heads are better than one. In *In Proceedings of the fourth Conference on Applied Natural Language Processing (ANLP-94*, pages 95–100, 1994.

[85] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[86] M. Gales and P.C. Woodland. Mean and variance adaptation within the MLLR framework. *Computer, Speech and Language*, 1996.

[87] A. Ganapathiraju and J. Picone. Hybrid SVM/HMM architectures for speech recognition. In *Proceedings of NIPS*, 2000.

[88] T. Gartner, P. Flach, and S. Wrobel. On graph kernels: Hardness results and efficient alternatives. *Lecture notes in computer science*, pages 129–143, 2003.

[89] J.L. Gauvain and C.H. Lee. Maximum a-posteriori estimation for multivariate gaussian mixture observations of markov chains. *IEEE Transactions on Speech and Audio Processing*, 2: 291–298, 1994.

[90] M.I. Gil'. A nonsingularity criterion for matrices. *Linear Algebra and Its Applications*, 253 (1-3):79–87, 1997.

[91] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *In Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[92] J. Goddard, AE Martinez, FM Martinez, and HL Rufiner. A comparison of string kernels and discrete hidden Markov models on a Spanish digit recognition task. In *Engineering in Medicine and Biology Society, 2003. Proceedings of the 25th Annual International Conference of the IEEE*, volume 3, 2003.

[93] A. Goldberg and J. Zhu. Seeing stars when there aren't many stars: Graph-based semi-supervised learning for sentiment categorization. In *HLT-NAACL Workshop on Graph-based Algorithms for Natural Language Processing*, 2006.

[94] J.F. Gómez-Lopera, J. Martínez-Aroza, A.M. Robles-Pérez, and R. Román-Roldán. An analysis of edge detection by using the Jensen-Shannon divergence. *Journal of Mathematical Imaging and Vision*, 13(1):35–56, 2000.

[95] C. Grover and R. Tobin. Rule-based chunking and reusability. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC 2006)*, 2006.

[96] C. Grozea. Finding optimal parameter settings for high performance word sense disambiguation. *Proceedings of Senseval-3 Workshop*, 2004.

[97] D. Gusfield. Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, 1997.

[98] N. Guttman and H. I. Kalish. Discriminability and stimulus generalization. *J Exp Psychol*, 51:79–88, Jan 1956. ISSN 0022-1015.

[99] N. Habash, B. Dorr, and C. Monz. Challenges in Building an Arabic-English GHMT System with SMT Components. In *Proceedings of the 11th Annual Conference of the European Association for Machine Translation (EAMT-2006)*, pages 56–65, 2006.

[100] G. Hanneman and A. Lavie. Decoding with Syntactic and Non-Syntactic Phrases in a Syntax-Based Machine Translation System. *SSST-3*, page 1, 2009.

[101] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani. *The elements of statistical learning*. Springer New York, 2001.

[102] D. Haussler. Convolution kernels on discrete structures. *In Technical Report UCS-CRL-99-10. UC*, 1999.

[103] T. Hofmann, B. Schölkopf, and A.J. Smola. Kernel methods in machine learning. *ANNALS OF STATISTICS*, 36(3):1171, 2008.

[104] Q. Hu, D. Yu, and Z. Xie. Neighborhood classifiers. *Expert systems with applications*, 34(2): 866–876, 2008.

[105] X. Huang and H.W. Hon. *Spoken Language Processing: A Guide to Theory, Algorithm, and System Development*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[106] K. Imamura. Application of translation knowledge acquired by hierarchical phrase alignment for pattern-based MT. *Proceedings of TMI*, pages 74–84, 2002.

[107] Vasile I. Istratescu. *Fixed Point Theory, An Introduction*. D. Reidel, the Netherlands, 1981. ISBN 90-277-1224-7.

[108] E. T. Jaynes. *Probability Theory: The Logic of Science*. Cambridge University Press, April 2003. ISBN 0521592712.

[109] T. Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European Conference on Machine Learning (ECML)*, pages 137–142, Berlin, 1998. Springer.

[110] T. Joachims. Transductive inference for text classification using support vector machines. In *Sixteenth International Conference on Machine Learning*, 1999.

[111] Thorsten Joachims. Transductive inference for text classification using support vector machines. In Ivan Bratko and Saso Dzeroski, editors, *Proceedings of ICML-99, 16th International Conference on Machine Learning*, pages 200–209, Bled, SL, 1999. Morgan Kaufmann Publishers, San Francisco, US. URL `http://citeseer.ist.psu.edu/joachims99transductive.html`.

[112] Thorsten Joachims. *Learning to Classify Text Using Support Vector Machines: Methods, Theory and Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2002. ISBN 079237679X.

[113] B.H. Juang, L. Rabiner, and J. Wilpon. On the use of bandpass liftering in speech recognition. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(7):947–954, 1987.

[114] D. Karger and M. Ruhl. Finding nearest neighbors in growth-restricted metrics, 2002. URL `citeseer.ist.psu.edu/karger02finding.html`.

[115] H. Kashima and A. Inokuchi. Kernels for graph classification. In *ICDM Workshop on Active Mining*, volume 2002, 2002.

[116] H. Kashima, K. Tsuda, and A. Inokuchi. Kernels for graphs. *Kernel methods in computational biology*, pages 155–170, 2004.

[117] A.M. Kibriya and E. Frank. An empirical comparison of exact nearest neighbour algorithms. *Lecture Notes in Computer Science*, 4702:140, 2007.

[118] K. Kilanski, J. Malkin, X. Li, R. Wright, and J. Bilmes. The Vocal Joystick data collection effort and vowel corpus. In *Interspeech*, September 2006.

[119] Margaret King. Evaluating natural language processing systems. *Commun. ACM*, 39(1): 73–79, 1996. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/234173.234208`.

[120] K. Kirchhoff and M. Yang. The University of Washington Machine Translation System for the IWSLT 2007 Competition. In *Proc. of the International Workshop on Spoken Language Translation*, 2007.

[121] J.M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 599–608. ACM New York, NY, USA, 1997.

[122] Kevin Knight and Ishwar Chander. Automated postediting of documents. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 779–784, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence. ISBN 0-262-61102-3.

[123] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1998.

[124] P. Koehn. Europarl: A parallel corpus for statistical machine translation. In *Machine Translation Summit X*, pages 79–86, Phuket, Thailand, 2005.

[125] P. Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. *Washington DC*, 2004.

[126] P. Koehn, F.J. Och, and D. Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 48–54. Association for Computational Linguistics Morristown, NJ, USA, 2003.

[127] Philipp Koehn and Christof Monz, editors. *Proceedings on the Workshop on Statistical Machine Translation*. Association for Computational Linguistics, New York City, June 2006. URL `http://www.aclweb.org/anthology/W/W06/W06-15`.

[128] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, Chris Dyer, Ondrej Bojar, Alexandra Constantin, and Evan Herbst. Moses: Open source toolkit for statistical machine translation. In *ACL*, 2007.

[129] Tuomo Korenius, Jorma Laurikkala, and Martti Juhola. On principal component analysis, cosine and Euclidean measures in information retrieval. *Information Sciences*, 177(22):4893–4905, 2007. ISSN 0020-0255. URL `http://www.sciencedirect.com/science/article/B6V0C-4NS2GMR-3/2/5392027024723e691b0c624af9ea5f2f`.

[130] H. Kucera and W.N. Francis. *Computational analysis of present-day American English*. Brown University Press Providence, 1967.

[131] J. Lafferty, X. Zhu, and Y. Liu. Kernel conditional random fields: representation and clique selection. In *Proceedings of the twenty-first international conference on Machine learning*. ACM New York, NY, USA, 2004.

[132] Elina Lagoudaki. Translation Memory systems: Enlightening users' perspective. Key finding of the TM Survey 2006 carried out during July and August 2006. Technical report, Imperial College London, Translation Memories Survey, 2006.

[133] P.W. Lamberti and A.P. Majtey. Non-logarithmic Jensen–Shannon divergence. *Physica A: Statistical Mechanics and its Applications*, 329(1-2):81–90, 2003.

[134] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[135] Y.K. Lee and H.T. Ng. An empirical evaluation of knowledge sources and learning algorithms for word sense disambiguation. In *Proceedings of EMNLP*, pages 41–48, 2002.

[136] Y.K. Lee, H.T. Ng, and T.K. Chia. Supervised Word Sense Disambiguation with Support Vector Machines and Multiple Knowledge Sources. *SENSEVAL-3*, 2004.

[137] G. Leech, P. Rayson, and A. Wilson. *Word frequencies in written and spoken English: based on the British National Corpus*. Longman, London, 2001.

[138] C. Leslie and R. Kuang. Fast Kernels for Inexact String Matching. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 114–128, 2003.

[139] C. Leslie, E. Eskin, and W. S. Noble. The spectrum kernel: a string kernel for svm protein classification. *Pac Symp Biocomput*, pages 564–575, 2002. ISSN 1793-5091. URL `http://view.ncbi.nlm.nih.gov/pubmed/11928508`.

[140] C. Leslie, E. Eskin, J. Weston, and W.S. Noble. Mismatch String Kernels for SVM Protein Classification. *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*, pages 1441–1448, 2003.

[141] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In *Soviet Physics Doklady*, volume 10, page 707, 1966.

[142] D.D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. *Lecture Notes in Computer Science*, 1398:4–18, 1998.

[143] W. Li. Random texts exhibit Zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38(6):1842–1845, 1992.

[144] Xiao Li. *Regularized Adaptation: Theory, Algorithms and Applications*. PhD thesis, University of Washington, 2007.

[145] H. T. Lin, C. J. Lin, and R. C. Weng. A Note on Platt's Probabilistic Outputs for Support Vector Machines. Technical report, National Taiwan University, May 2003. URL `http://work.caltech.edu/~htlin/publication/doc/plattprob.pdf`.

[146] J. Lin. Divergence measures based on the Shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[147] Marc Lipson. *Schaum's Easy Outline of Discrete Mathematics*. McGraw-Hill, 2002.

[148] Ning Liu, Benyu Zhang, Jun Yan, Qiang Yang, Shuicheng Yan, Zheng Chen, Fengshan Bai, and Wei-Ying Ma. Learning similarity measures in non-orthogonal space. In *CIKM '04: Proceedings of the thirteenth ACM international conference on Information and knowledge management*, pages 334–341, New York, NY, USA, 2004. ACM. ISBN 1-58113-874-1. URL `http://doi.acm.org/10.1145/1031171.1031240`.

[149] H. Lodhi, J. Shawe-Taylor, and N. Cristianini. Text classification using string kernels. In *Proceedings of NIPS*, 2002.

[150] W. Macherey, F.J. Och, I. Thayer, and J. Uszkoreit. Lattice-based minimum error rate training for statistical machine translation. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing (EMNLP), Honolulu*, 2008.

[151] Christopher D. Manning and Hinrich Schtze. *Foundations of Statistical Natural Language Processing*. MIT Press, June 1999. ISBN 0262133601.

[152] D. Marcu. Towards a unified approach to memory- and statistical-based machine translation. In *Proceedings of ACL*, 2001.

[153] D. Marcu and W. Wong. A phrase-based, joint probability model for statistical machine translation. In *Proceedings of EMNLP*, volume 2, 2002.

[154] Alvin Martin and Mark Przybocki. *2004 NIST Speaker Recognition Evaluation*. Linguistic Data Consortium, Philadelphia, 2006.

[155] A. Martins. String kernels and similarity measures for information retrieval. Technical report, Technical report, Priberam, Lisbon, Portugal, 2006.

[156] H. Maruyana and H. Watanabe. Tree cover search algorithm for example-based translation. In *Proceedings of TMI*, pages 173–184, 1992.

[157] W.J. McGuire. A multiprocess model for paired-associate learning. *Journal of Experimental Psychology*, 62:335–347, 1961.

[158] M.L. Menéndez, J.A. Pardo, L. Pardo, and M.C. Pardo. The Jensen-Shannon divergence. *Journal of the Franklin Institute*, 334(2):307–318, 1997. ISSN 0016–0032.

[159] J. Mercer. Functions of Positive and Negative Type, and Their Connection with the Theory of Integral Equations. *Proceedings of the Royal Society of London. Series A*, 83(559):69–70, 1909.

[160] María Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recogn. Lett.*, 15(1):9–17, 1994. ISSN 0167-8655. URL `http://dx.doi.org/10.1016/0167-8655(94)90095-7`.

[161] R. Mihalcea, T. Chklovski, and A. Killgariff. The Senseval-3 English Lexical Sample Task. In *Proceedings of ACL/SIGLEX Senseval-3*, 2004.

[162] George A. Miller and Patricia E. Nicely. An analysis of perceptual confusions among some english consonants. *The Journal of the Acoustical Society of America*, 27(2):338–352, 1955. URL `http://dx.doi.org/10.1121/1.1907526`.

[163] Thomas P. Minka. Bayesian inference, entropy, and the multinomial distribution, 2007. URL `http://citeseer.ist.psu.edu/171980.html`.

[164] S. Mohammad and T. Pedersen. Complementarity of Lexical and Simple Syntactic Features: The SyntaLex Approach to Senseval-3. *Proceedings of the SENSEVAL-3*, 2004.

[165] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[166] Andrew Moore. A tutorial on kd-trees. Technical Report 209, University of Cambridge, 1991. URL `http://www.cs.cmu.edu/~awm/papers.html`. Extract from PhD Thesis.

[167] Pedro J. Moreno, Purdy P. Ho, and Nuno Vasconcelos. A Kullback-Leibler divergence based kernel for SVM classification in multimedia applications. In *In Advances in Neural Information Processing Systems 16*. MIT Press, 2003.

[168] Arnold Neumaier. Solving ill-conditioned and singular linear systems: A tutorial on regularization. *SIAM Review*, 40:636–666, 1998.

[169] A.Y. Ng. Feature selection, L_1 vs. L_2 regularization, and rotational invariance. *ACM International Conference Proceeding Series*, 2004.

[170] NIST. Automatic evaluation of machine translation quality using n-gram co-occurrence statistics. *NIST*, 2002. URL `http://www.nist.gov/speech/tests/mt/doc/ngram-study.pdf`.

[171] N.Young. *An Introduction to Hilbert Spaces*. Cambridge University Press, 1988. ISBN 978–0521337175.

[172] F.J. Och. Minimum Error Rate Training in Statistical Machine Translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 160–167. Association for Computational Linguistics Morristown, NJ, USA, 2003.

[173] F.J. Och and H. Ney. Discriminative training and maximum entropy models for statistical machine translation. In *Proc. of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 8, 2002.

[174] F.J. Och and H. Ney. Giza++: Training of statistical translation models. *Disponible sur http://www. fjoch. com/GIZA++. html*, 2003.

[175] F.J. Och and H. Ney. The alignment template approach to statistical machine translation. *Computational Linguistics*, 30(4):417–449, 2004.

[176] F.J. Och, C. Tillmann, H. Ney, et al. Improved alignment models for statistical machine translation. In *Proc. of the Joint SIGDAT Conf. on Empirical Methods in Natural Language Processing and Very Large Corpora*, pages 20–28, 1999.

[177] M.T. Orchard. A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2297–2300, 1991.

[178] B. Pang and L. Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the ACL*, pages 115–124, 2005.

[179] B. Pang and L. Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts. *Proceedings of the ACL*, pages 271–278, 2004.

[180] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *ACL '02: Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 311–318, Morristown, NJ, USA, 2001. Association for Computational Linguistics.

[181] M. Paul, E. Sumita, and S. Yamamoto. Example-based Rescoring of Statistical Machine Translation Output. *Proc of the HLTNAACL, Companion Volume*, pages 9–12, 2004.

[182] J. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. *Advances in Large Margin Classifiers*, 6174, 1999.

[183] M. Przybocki, G. Sanders, and A. Le. Edit Distance: A Metric for Machine Translation Evaluation. In *Actes de LREC 2006 (5^{th} International Conference on Language Resources and Evaluation)*, pages 2038–2043, 2006.

[184] Taylor R. A users guide to measure-theoretic probability. *Journal of the American Statistical Association*, 98:493–494, January 2003. URL `http://ideas.repec.org/a/bes/jnlasa/v98y2003p493-494.html`.

[185] C. Rao. Differential Geometry in Statistical Interference. *IMS-Lecture Notes*, 10:217, 1987.

[186] A. Ratnaparkhi. A maximum entropy model for part-of-speech tagging. In *Proceedings of EMNLP*, pages 133–142, 1996. URL `http://citeseer.ist.psu.edu/ratnaparkhi96maximum.html`.

[187] M. Reed and B. Simon. *Functional Analysis. Revised and enlarged Edition.*, volume I of *Methods of Modern Mathematical Physics*. Academic Press, San Diego, 1980.

[188] JD Reiss, J. Selbie, and MB Sandler. OPTIMISED KD-TREE INDEXING OF MULTIMEDIA DATA. In *Digital Media Processing for Multimedia Interactive Services: Proceedings of the 4th European Workshop on Image Analysis for Multimedia Interactive Services*, page 47. World Scientific, 2003.

[189] K. Rieck, P. Laskov, and S. Sonnenburg. Computation of similarity measures for sequential data using generalized suffix trees. *Advances in Neural Information Processing Systems*, 19: 1177, 2007.

[190] Raùl Rojas. *Neural Networks: A Systematic Introduction*. Springer, 1996.

[191] R. Román-Roldán, P. Bernaola-Galván, and J.L. Oliver. Sequence compositional complexity of DNA through an entropic segmentation method. *Physical Review Letters*, 80(6):1344–1347, 1998.

[192] R. Rönngren and R. Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2): 157–209, 1997.

[193] F. Sadat, H. Johnson, A. Agbago, G. Foster, R. Kuhn, J. Martin, and A. Tikuisis. PORTAGE: A Phrase-based Machine Translation System. *Building and Using Parallel Texts: Data-Driven Machine Translation and Beyond*, 2005.

[194] G. Salton and M.J. McGill. *Introduction to modern information retrieval*. McGraw-Hill, Inc. New York, NY, USA, 1986.

[195] Gerard Salton. *Introduction to Modern Information Retrieval (McGraw-Hill Computer Science Series)*. McGraw-Hill Companies, September 1983. ISBN 0070544840.

[196] M. Santaholma. Grammar sharing techniques for rule-based multilingual NLP systems. In *Proceedings of NODALIDA*, pages 253–260. Citeseer, 2007.

[197] J. Schroeder and P. Koehn. The University of Edinburgh System Description for IWSLT 2007. In *Proc. of the International Workshop on Spoken Language Translation*, 2007.

[198] F. Sha and L. Saul. Large margin Gaussian mixture modeling for phonetic classification and recognition. In *Proceedings of ICASSP*, pages 265–268, 2006.

[199] Marvin Shapiro. The choice of reference points in best-match file searching. *Comm. ACM*, 20(5):339–343, 1977. ISSN 0001-0782. URL `http://doi.acm.org/10.1145/359581.359599`.

[200] D. Shen, J. Zhang, J. Su, G. Zhou, and C.L. Tan. Multi-criteria-based active learning for named entity recognition. In *Proceedings of the 42nd Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics Morristown, NJ, USA, 2004.

[201] R.N. Shepard. Toward a universal law of generalization for physical science. *Science*, 237: 1317–1323, 1987.

[202] Jonathon Shlens. A tutorial on principal component analysis, December 2005. URL `http://www.snl.salk.edu/~shlens/pub/notes/pca.pdf`.

[203] A.W.M. Smeulders and R. Jain. *Image databases and multi-media search*. World Scientific, 1997.

[204] Ivan Smith. Historical Notes about the Cost of Hard Drive Storage Space. URL `http://alts.net/ns1625/winchest.html`, 2004.

[205] A. Solomonoff, C. Quillen, and W.M. Campbell. Channel compensation for SVM speaker recognition. In *ODYSSEY04-The Speaker and Language Recognition Workshop*, 2004.

[206] R. Sproat and T. Emerson. The First International Chinese Word Segmentation Bakeoff. In *Proceedings of the Second SIGHAN Workshop on Chinese Language Processing*, pages 133–143. Sapporo, Japan: July, 2003.

[207] David McG. Squire. Learning a similarity-based distance measure for image database organization from human partitionings of an image set. *Applications of Computer Vision, IEEE Workshop on*, 0:88, 1998. URL `http://doi.ieeecomputersociety.org/10.1109/ACV.1998.732863`.

[208] GW Stewart. Gershgorin Theory for the Generalized Eigenvalue Problem Ax=λBx. *Mathematics of Computation*, pages 600–606, 1975.

[209] A. Stolcke. SRILM-an extensible language modeling toolkit. In *Seventh International Conference on Spoken Language Processing*. ISCA, 2002.

[210] A. Stolcke, F. Grezl, M-Y Hwang, X. Lei, N. Morgan, and D. Vergyri. Cross-domain and cross-language portability of acoustic features estimated by multilayer perceptrons. In *Proceedings of ICASSP*, pages 321–324, 2006.

[211] C. Strapparava, A. Gliozzo, and C. Giuliano. Pattern abstraction and term similarity for word sense disambiguation: IRST at SENSEVAL-3. *Proc. of SENSEVAL-3*, pages 229–234, 2004.

[212] J. Suzuki, Y. Sasaki, and E. Maeda. Kernels for structured natural language data. In *Proc. of the 17th Annual Conference on Neural Information Processing Systems (NIPS2003)*, 2003.

[213] Zhuoran Wang;John Shawe-Taylor;Sandor Szedmak. Kernel regression based machine translation. In *Proceedings of NAACL/HLT*, pages 185–188. Association for Computational Linguistics, 2007. URL `http://www.aclweb.org/anthology/N/N07/N07-2047`.

[214] Martin Szummer and Tommi Jaakkola. Partially labeled classification with Markov random walks. In *Advances in Neural Information Processing Systems*, volume 14, 2001. URL `http://citeseer.ist.psu.edu/szummer02partially.html`.

[215] B. Taskar, C. Guestrin, and D. Koller. Max-margin Markov networks. *Advances in neural information processing systems*, 16:51, 2004.

[216] O. Taussky. A recurring theorem on determinants. *American Mathematical Monthly*, pages 672–676, 1949.

[217] F. Topsøe. Jenson-Shannon divergence and norm-based measures of discrimination and variation. *Preprint*, 2003.

[218] Ioannis Tsochantaridis, Thomas Hofmann, Thorsten Joachims, and Yasemin Altun. Support vector machine learning for interdependent and structured output spaces. In *ICML '04: Proceedings of the twenty-first international conference on Machine Learning*, page 104, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-828-5. URL `http://doi.acm.org/10.1145/1015330.1015341`.

[219] Nicola Ueffing, Gholamreza Haffari, and Anoop Sarkar. Semi-supervised model adaptation for statistical machine translation. *Machine Translation*, 21(2):77–94, June 2007. URL `http://dx.doi.org/10.1007/s10590-008-9036-3`.

[220] J.D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM New York, NY, USA, 1995.

[221] T. Veale and A. Way. Gaijin: a template-based bootstrapping approach to example-based machine translation. In *Proceedings of News Methods in Natural Language Processing*, 1997.

[222] E. Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters*, 15(1):1–7, 1994.

[223] S. V. N. Vishwanathan and Alexander J. Smola. Fast kernels for string and tree matching. In *Advances in Neural Information Processing Systems 15*, pages 569–576. MIT Press, 2003.

[224] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.P. Seidel. Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5): 562–572, 2005.

[225] R. Wang, X. Wang, Z. Chen, and Z. Chi. Chunk Segmentation of Chinese Sentences Using a Combined Statistical and Rule-based Approach (CSRA). *Int. J. Comp. Proc. Orie. Lang.*, 20 (02n03):197–218, 2007.

[226] T. Watanabe and E. Sumita. Example-based decoding for statistical machine translation. In *Machine Translation Summit IX*, pages 410–417, 2003.

[227] DF Watson. Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes*. *The computer journal*, 24(2):167–172, 1981.

[228] K. Yamada and K. Knight. A decoder for syntax-based statistical MT. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, pages 303–310. Association for Computational Linguistics Morristown, NJ, USA, 2001.

[229] D. Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In *Proceedings of the 33rd annual meeting on Association for Computational Linguistics*, pages 189–196. Association for Computational Linguistics Morristown, NJ, USA, 1995.

[230] C. Yin, S. Tian, S. Mu, and C. Shao. Efficient computations of gapped string kernels based on suffix kernel. *Neurocomputing*, 71(4-6):944–962, 2008.

[231] H.P. Zhang, H.K. Yu, D.Y. Xiong, and Q. Liu. HHMM-based Chinese Lexical Analyzer ICTCLAS. In *Proceedings of Second SIGHAN Workshop on Chinese Language Processing*, pages 184–187, 2003.

[232] D. Zhou, O. Bousquet, T.N. Lal, J. Weston, and B. Schölkopf. Learning with local and global consistency. *Advances in Neural Information Processing Systems*, 16:321–328, 2004.

[233] Q. Zhu, A. Stolcke, B. Chen, and N. Morgan. Using MLP features in SRI's conversational speech recognition system. In *Proceedings of Eurospeech*, pages 2141–2144, 2005.

[234] X. Zhu and Z. Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical report, CMU-CALD-02, 2002. URL `http://citeseer.ist.psu.edu/article/zhu02learning.html`.

[235] X. Zhu and Z. Ghahramani. Towards semi-supervised classification with Markov random fields. Technical report, Technical Report CMU-CALD-02-106). Carnegie Mellon University, 2002.

[236] X. Zhu and Z. Ghahramani. Towards semi-supervised learning with Boltzmann machines. Technical report, Technical report, Carnegie Mellon University, 2002.

[237] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions. *ICML-03, 20th International Conference on Machine Learning*, 2003.

[238] Xiaojin Zhu. *Semi-Supervised Learning with Graphs*. PhD thesis, Carnegie Mellon University, 2005. CMU-LTI-05-192.

[239] Xiaojin Zhu. Semi-supervised learning literature survey. Technical Report 1530, Computer Sciences, University of Wisconsin-Madison, 2005. URL `http://www.cs.wisc.edu/~jerryzhu/pub/ssl_survey.pdf`.

[240] G. Zipf. Selective Studies and the Principle of Relative Frequencies in Language. *MIT Press*, 1932.

[241] Justin Zobel, Alistair Moffat, and Ron Sacks-Davis. An efficient indexing technique for full text databases. In *VLDB '92: Proceedings of the 18th International Conference on Very Large Data Bases*, pages 352–362, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc. ISBN 1-55860-151-1.