Deploying Decision Tree Ensembles to DBMS

Fianl Report, CSE 544 Winter 2018

Hyunsu Cho

Problem Description

The **decision tree ensemble** is one of the most well-known statistical models used in the field of machine learning. With a long history in reseach literature, the tree ensemble has been used with great effect, showing impressive predictive performance on many tabular datasets[1].

The goal of this project is to **deploy** tree ensemble models to a modern database system for the prediction task. Existing packages require that the data be moved into the prediction code, which may be costly. We will instead move the prediction code into the database as a **user-defined function (UDF)**. It is often said that it is better to move code to where data is than to move data to where code is[2]; let's see if it's true for decision trees as well.

Project Description

In order to compile decision tree models into UDFs, I extended **treelite** (`http://treelite.io`), a package for decision tree deployment. Treelite grew out of earlier ideas presented in my quals paper. It lets users compile any decision tree models into a C program, which then gets compiled further into machine code. A modular design (Figure 1) provides for a clear separation between the front-end and back-end modules: the front-end ingests models from various libraries and translates it in a common model schema; the back-end then reads the schema and produces a C file containing prediction logic.

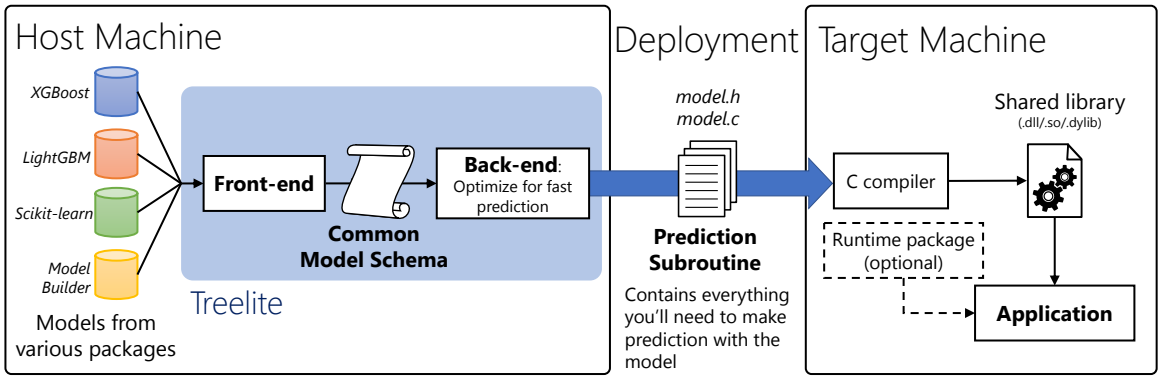

Figure 1: Existing workflow of Treelite

UDF Wrapper for PostgreSQL

For the course project, **treelite was extended to produce a user-defined function for PostgreSQL** (Figure 2). The prediction subroutine is now wrapped by a wrapper function that parses incoming data from PostgreSQL tables. The two functions together constitutes the UDF. Each tuple (data row) is passed to the wrapper and then parsed into a set of feature values. For instance, the following piece of code attempts to read the field `feature_0` from the incoming tuple and sets the corresponding array element `inst[0]` appropriately:

```
HeapTupleHeader t = PG_GETARG_HEAPTUPLEHEADER(0);  /* read tuple header */
Datum d = GetAttributeByName(t, "feature_0", &isnull);
```

[1] XGBoost, a well-optimized package implementing the training algorithm for decision tree ensembles, has been extensively used in winning solutions of various data science competitions. See `https://github.com/dmlc/xgboost/blob/master/demo/README.md#machine-learning-challenge-winning-solutions` for an example.

[2] A nugget of wisdom cited from "What Goes Around Comes Around," by Stonebreaker and Hellerstein.

```
if (isnull) {
  inst[0].missing = -1;                    /* field missing */
} else {
  inst[0].fvalue = (float)DatumGetFloat4(d); /* field present */
}
```

The prediction subroutine is invoked once all fields are read. Once the subroutine returns a predicted label, it gets returned back to the server as follows:

```
float res = predict_margin(inst);
if (!pred_margin) { /* whether to return probabilities or raw scores */
  pred_transform(&res);
}
PG_RETURN_FLOAT4((float4)res);
```

The wrapper function will include all necessary header files from the PostgreSQL installation.

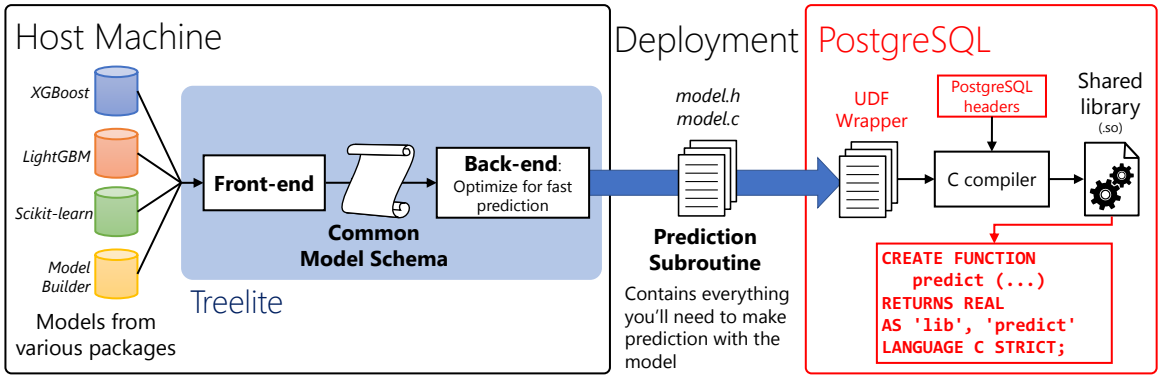

Figure 2: New workflow, with UDF deployed to PostgreSQL

The UDF is loaded into PostgreSQL via the SQL query

```
CREATE FUNCTION predict(my_table, BOOLEAN) RETURNS REAL
AS 'DIRECTORY/lib', 'predict'
LANGUAGE C STRICT;
```

We can use the loaded UDF by simply writing

```
SELECT predict(my_table, FALSE) FROM my_table LIMIT 20;  -- predict probabilities
SELECT predict(my_table, TRUE)  FROM my_table LIMIT 20;  -- predict raw scores
```

It takes only a few API calls to export a tree model as a UDF:

```
import treelite
model = treelite.Model.load('my_model.model', 'xgboost')
model.compile(dirpath='./my_model', params={'export_udf':1})
treelite.generate_makefile(dirpath='./my_model', platform='osx',
                           toolchain='gcc-7', export_udf=True)
# Now simply run `make` in the directory `my_model` and we're done
```

In order to test the functionality of the UDF extension, I imported the HIGGS dataset[3] as a table in PostgreSQL. Since the dataset was originally in LibSVM format [3], I first had to convert it into comma-separated values internally before it could be imported into PostgreSQL:

```
import psycopg2
import pandas
from io import StringIO
from fast_svmlight_loader import load_svmlight
```

[3] "HIGGS Data Set" https://archive.ics.uci.edu/ml/datasets/HIGGS

```
# load HIGGS dataset (libsvm format)
data = load_svmlight('higgs.train.libsvm', verbose=True)
X, y = data['data'], data['labels']
# convert to pandas dataframe
df = pandas.DataFrame(X.toarray())
df.columns = ['feature_{}'.format(i) for i in range(29)]
# connect to DB
conn = psycopg2.connect(dbname='chohyu01', user='chohyu01')
cur = conn.cursor()
# prepare a table schema
command = 'CREATE TABLE higgs ({});'\
          .format(','.join(['{} float(24)'.format(x) for x in df.columns]))
cur.execute(command)
conn.commit()
# convert HIGGS into comma-separated values
sio = StringIO()
sio.write(df.to_csv(index=False, header=False))
sio.seek(0)
# import comma-separated values into DB
cur.copy_from(sio, 'higgs', columns=df.columns, sep=',')
conn.commit()
# add primary key
cur.execute('ALTER TABLE higgs ADD COLUMN id BIGSERIAL PRIMARY KEY;')
conn.commit()
cur.close()
conn.close()
```

It took about 20 minutes to import 10 million rows. I then ran `CREATE FUNCTION` to load the compiled UDF into memory. Here is the output of a sample query involving the UDF:

```
chyunsu=# SELECT id, predict(higgs, TRUE)  AS score,
                     predict(higgs, FALSE) AS probability
          FROM higgs
          LIMIT 10;
 id |   score    | probability
----+------------+-------------
  1 |    1.37856 |     0.79876
  2 |   0.926516 |    0.716368
  3 |    3.75283 |    0.977086
  4 |    0.64713 |    0.656363
  5 | -0.0376431 |     0.49059
  6 |   -3.21826 |   0.0384842
  7 |    4.49237 |     0.98893
  8 |    3.94098 |    0.980941
  9 |    5.40972 |    0.995547
 10 |    4.64762 |    0.990507
(10 rows)
```

Support for Categorical Features

Despite my earlier plans, I ended up spending most of my time writing logic for categorical features. Categorical features present challenges because they are usually stored as string in the database but machine learning packages only understand numbers. A typical response is to use dataframe libraries like Pandas[4] to convert categorical feature values into numerical values before training decision tree models. The existence of categorical features also affects your choice about the tree libraries to train your model. I chose to use LightGBM[5] for this

[4] https://pandas.pydata.org/
[5] https://github.com/Microsoft/LightGBM

task because, unlike XGBoost[6], LightGBM provides more seamless support for categorical features[7].

To properly handle data with categorical features, the UDF wrapper keeps track of one-to-one mapping between category names and corresponding category (integer) codes. This mapping is created by Pandas before the model is trained, and we must use the same mapping at testing time. The wrapper has a lookup table to find integer codes for any category names, as follows:

```
static const
std::unordered_map<unsigned, std::unordered_map<std::string, int>>
category_map{
  {2, {
    {"A", 0}, {"AB", 1}, {"AC", 2}, {"AD", 3}, {"AE", 4}, {"AF", 5}, ... }},
  {14, {
    {"A", 0}, {"B", 1}, {"C", 2},   }},
  {16, {
    {"A", 0}, {"B", 1}, {"C", 2}, {"D", 3}, {"E", 4}, {"F", 5},   }},
  ...
};
```

In addition, the UDF wrapper needs to map feature names into feature indices. This is because columns typically have human-readable names (e.g. `Sales`). The mapping is similarly hard-coded into the wrapper. The UDF extension adds a feature to treelite that automatically generates this wrapper, since it is too cumbersome to write the wrapper by hand. The mapping for feature names and category names is discovered from saved model files.

To evaluate the categorical feature handling, I used dataset from Allstate Claim Prediction Challenge[8]. It consists of past claim payouts from a bodily liability insurance. Unlike higgs, the Allstate dataset has many categorical features; this is expected in many real-world datasets.

Writing SQL queries with model predicton

One interesting application of deploying tree models for prediction is the ability to write analytic SQL queries. A large ensemble of decision trees is effectively a black box, meaning we don't have a good intuitive interpretation of how the model makes decision for a given example. So a practitioner may want to ask questions about its behavior such as: on which examples in the dataset does this model makes most mistakes? If one or two features are constrained, does the model make better or worse predictions? For instance, the query

```
SELECT AVG(claim_amount) AS avg_true_label,
       AVG(predict(x)) AS avg_predicted_label,
       ABS(AVG(claim_amount) - AVG(predict(x))) AS overall_disparity
       FROM allstate x
       GROUP BY blind_make
       ORDER BY overall_disparity DESC;
```

will list car manufacturers whose cars the given tree model mispredict the most, as measured by the average difference between the predicted and true payouts.

Related Works

Last month, I presented treelite at an industry conference [1]. The poster presentation there does not include the UDF portion that was developed in this project. The poster shows performance benchmark results that compare prediction throughput of Treelite to that of the prediction method of a popular package XGBoost (`xgboost.Booster.predict()`).

Similar to this work, Ordonez [2] also uses UDF for machine learning tasks. He first derives a few sufficient statistics that are common to regression, principal component analysis (PCA), clustring, and Naive Bayes. Then he writes UDFs to efficiently compute those sufficient statistics, most of the time requiring only a single scan over the data table. Unlike Ordonez, I defer the training task to a standard machine learning package. This is because

[6] `https://github.com/dmlc/xgboost`

[7] See `http://lightgbm.readthedocs.io/en/latest/Quick-Start.html?highlight=categorical#categorical-feature-support` for more information.

[8] https://www.kaggle.com/c/ClaimPredictionChallenge

decision tree learning is not as easily reducible to a few sufficient statistics and thus would require several passes over the table. My work will be useful in scenarios where one trains a decision tree model on a relatively small amount of data and then uses the model to score a large table. Such scenario is not too uncommon in document scoring and ad ranking.

Closer to home, Brandon Myers, a recent graduate of Paul G. Allen School of Computer Science at University of Washington, investigates query compilation for high-performance computing applications [4]. Not only UDFs but also queries themselves are getting compiled into parallel programs. A notable fact is the use of columnar storage rather than the usual row-oriented storage. Column-oriented storage makes insertion and deletion harder but allows for better compression and faster read accesses.

Conclusion and Future Works

Since the UDF currently runs with a single thread, it is quite slow. It took nearly one hour to evaluate one million rows! The reason is that the query optimizer in PostgreSQL has no idea what to do with the UDF and simply defaults to sequential scan. So the opportunity for intra-query parallelism is lost. The UDF support still has merits for usability standpoints, but for better performance, a tighter integration with the database is necessary. At the very least, there needs to be a way to provide the query optimizer necessary information about the UDF so that it can choose parallel scans for the prediction task. This would require us to revise the source code of PostgreSQL. Another possibility is to store model information as database tables and express the prediction task as a set of SELECTs, JOINs, and aggregates. This has a major complication because the semantics for predicting with tree models is highly branch-dependent. We are yet unsure about whether the task can be cleanly mapped to SQL queries. Lastly, we might be able to write Datalog programs.

References

[1] Hyunsu Cho and Mu Li. Treelite: toolbox for decision tree deployment. SysML 2018.

[2] Carlos Ordonez. "Statistical Model Computation with UDFs," in *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 12, pp. 1752-1765, December 2010. Accessed at `http://ieeexplore.ieee.org/document/5432172/`

[3] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: a library for support vector machines. ACM Transactions on Intelligent Systems and Technology, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[4] Brandon Myers. Integrating query processing with parallel languages. IEEE HPEC, September 2015.