CSE546 Machine Learning, Autumn 2014: Homework 1

Due: Tuesday, October 14^{th}, beginning of class

1 Probability [10 points]

Let X_1 and X_2 be independent, continuous random variables uniformly distributed on $[0, 1]$. Let $X = \min(X_1, X_2)$. Compute

1. *(3 points)* $E(X)$.
2. *(3 points)* $Var(X)$.
3. *(4 points)* $Cov(X, X_1)$.

2 MLE [8 points]

This question uses a discrete probability distribution known as the Poisson distribution. A discrete random variable X follows a Poisson distribution with parameter λ if

$$\Pr(X = k) = \frac{\lambda^k}{k!} e^{-\lambda} \qquad k \in \{0, 1, 2, \dots\}$$

Imagine we have a gumball machine which dispenses a random number of gumballs each time you insert a quarter. Assume that the number of gumballs dispensed is Poisson distributed (i.i.d) with parameter λ. Curious and sugar-deprived, you insert 8 quarters one at a time and record the number of gumballs dispensed on each trial:

Trial	1	2	3	4	5	6	7	8
Gumballs	4	1	3	5	5	1	3	8

Let $G = (G_1, \dots, G_n)$ be a random vector where G_i is the number of gumballs dispensed on trial i:

1. *(3 points)* Give the log-likelihood function of G given λ.
2. *(4 points)* Compute the MLE for λ in the general case.
3. *(1 point)* Compute the MLE for λ using the observed G.

3 Linear Regression and LOOCV [16 points]

In class you learned about using cross validation as a way to estimate the true error of a learning algorithm. A solution that provides an almost unbiased estimate of this true error is *Leave-One-Out Cross Validation* (LOOCV), but it can take a really long time to compute the LOOCV error. In this problem you will derive an algorithm for efficiently computing the LOOCV error for linear regression using the *Hat Matrix*. [1] (This is the *cool trick* alluded to in the slides!)

[1] Unfortunately, such an efficient algorithm may not be easily found for other learning methods.

Assume that there are n training examples, $(X_1, y_1), (X_2, y_2), \ldots, (X_n, y_n)$, where each input data point X_i, has d real valued features. The goal of regression is to learn to predict y_i from X_i. The *linear* regression model assumes that the output y is a *linear* combination of the input features plus Gaussian noise with weights given by w.

We can write this in matrix form by stacking the data points as the rows of a matrix X so that $x_i^{(j)}$ is the i-th feature of the j-th data point. Then writing Y, w and ϵ as column vectors, we can write the matrix form of the linear regression model as:

$$Y = Xw + \epsilon$$

where:

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \ldots & x_d^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \ldots & x_d^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ x_1^{(n)} & x_2^{(n)} & \ldots & x_d^{(n)} \end{bmatrix}, w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}, \quad \text{and } \epsilon = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

Assume that ϵ_i is normally distributed with variance σ^2. We saw in class that the maximum likelihood estimate of the model parameters w (which also happens to minimize the sum of squared prediction errors) is given by the *Normal equation*:

$$\hat{w} = (X^T X)^{-1} X^T Y$$

Define $\hat{Y}$ to be the vector of predictions using $\hat{w}$ if we were to plug in the original training set X:

$$\begin{aligned} \hat{Y} &= X\hat{w} \\ &= X(X^T X)^{-1} X^T Y \\ &= HY \end{aligned}$$

where we define $H = X(X^T X)^{-1} X^T$ (H is often called the *Hat Matrix*).

As mentioned above, $\hat{w}$, also minimizes the sum of squared errors:

$$\text{SSE} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Now recall that the Leave-One-Out Cross Validation score is defined to be:

$$\text{LOOCV} = \sum_{i=1}^{n} (y_i - \hat{y}_i^{(-i)})^2$$

where $\hat{Y}^{(-i)}$ is the estimator of Y after removing the i-th observation (i.e., it minimizes $\sum_{j \neq i} (y_j - \hat{y}_j^{(-i)})^2$).

1. *(3 points)* What is the time complexity of computing the LOOCV score naively? (The naive algorithm is to loop through each point, performing a regression on the $n-1$ remaining points at each iteration.) *Hint:* The complexity of matrix inversion is $O(k^3)$ for a $k \times k$ matrix [2].
2. *(1 point)* Write $\hat{y}_i$ in terms of H and Y.
3. *(4 points)* Show that $\hat{Y}^{(-i)}$ is also the estimator which minimizes SSE for Z where

$$Z_j = \begin{cases} y_j, & j \neq i \\ \hat{y}_i^{(-i)}, & j = i \end{cases}$$

4. *(1 point)* Write $\hat{y}_i^{(-i)}$ in terms of H and Z. By definition, $\hat{y}_i^{(-i)} = Z_i$, but give an answer that is analogous to *2*.

[2] There are faster algorithms out there but for simplicity we'll assume that we are using the naive $O(k^3)$ algorithm.

5. *(3 points)* Show that $\hat{y}_i - \hat{y}_i^{(-i)} = H_{ii}y_i - H_{ii}\hat{y}_i^{(-i)}$, where H_{ii} denotes the i-th element along the diagonal of H.

6. *(4 points)* Show that

$$LOOCV = \sum_{i=1}^{n} \left(\frac{y_i - \hat{y}_i}{1 - H_{ii}} \right)^2$$

What is the algorithmic complexity of computing the LOOCV score using this formula?

Note: We see from this formula that the diagonal elements of H somehow indicate the impact that each particular observation has on the result of the regression.

4 Regularization Constants [16 points]

We have discussed the importance of regularization as a technique to avoid overfitting our models. For linear regression, we have mentioned both LASSO (which uses the L_1 norm as a penalty), and ridge regression (which uses the squared L_2 norm as a penalty). In practice, the scaling factor of these penalties has a significant impact on the behavior of these methods, and must often be chosen empirically for a particular dataset. In this problem, we look at what happens when we choose our regularization factor poorly.

For the following, recall that the loss function to be optimized under ridge regression is

$$E_R = \sum_{i=1}^{n} (y_i - (\hat{w}_0 + x^{(j)}\hat{w}))^2 + \lambda \|\hat{w}\|_2^2$$

where

$$\lambda \|\hat{w}\|_2^2 = \lambda \sum_{i=1}^{d} (\hat{w}_i)^2 \tag{1}$$

and λ is our regularization constant.

The loss function to be optimized under LASSO regression is

$$E_L = \sum_{i=1}^{n} (y_i - (\hat{w}_0 + x^{(j)}\hat{w}))^2 + \lambda \|\hat{w}\|_1$$

where

$$\lambda \|\hat{w}\|_1 = \lambda \sum_{i=1}^{d} |\hat{w}_i|. \tag{2}$$

1. *(8 points)* Discuss briefly how choosing too small a λ affects the magnitude of the following quantities. Please describe the effects for both ridge and LASSO, or state why the effects will be the same.

   (a) The error on the training set.

   (b) The error on the testing set.

   (c) The elements of $\hat{w}$.

   (d) The number of nonzero elements of $\hat{w}$.

2. *(8 points)* Now discuss briefly how choosing too large a λ affects the magnitude of the same quantities in the previous question. Again describe the effects for both ridge and LASSO, or state why the effects will be the same.

Algorithm 1: Coordinate Descent Algorithm for Lasso

while *not converged* **do**

$w_0 \leftarrow \sum_{i=1}^{N} \left(\mathbf{y}_i - \sum_j \mathbf{w}_j \mathbf{X}_{ij} \right) / N$

for $k \in \{1, 2, \cdots d\}$ **do**

$a_k \leftarrow 2 \sum_{i=1}^{N} \mathbf{X}_{ik}^2$

$c_k \leftarrow 2 \sum_{i=1}^{N} \mathbf{X}_{ik} \left(\mathbf{y}_i - (w_0 + \sum_{j \neq k} \mathbf{w}_j \mathbf{X}_{ij}) \right)$

$$\mathbf{w}_k \leftarrow \begin{cases} (c_k + \lambda)/a_k & c_k < -\lambda \\ 0 & c_k \in [-\lambda, \lambda] \\ (c_k - \lambda)/a_k & c > \lambda \end{cases}$$

end

end

5 Programming Question [50 points]

The Lasso is the problem of solving

$$\arg\min_{\mathbf{w}, w_0} \sum_i (\mathbf{X_i w} + w_0 - \mathbf{y}_i)^2 + \lambda \sum_j |\mathbf{w}_j| \tag{3}$$

Here $\mathbf{X}$ is an $N \times d$ matrix of data, and $\mathbf{X_i}$ is the i-th row of the matrix. $\mathbf{y}$ is an $N \times 1$ vector of response variables, $\mathbf{w}$ is a d dimensional weight vector, w_0 is a scalar offset term, and λ is a regularization tuning parameter. For the programming part of this homework, you are required to implement the coordinate descent method that can solve the Lasso problem.

This question contains three parts, 1) analyze and optimize the time complexity 2) test your code using toy examples 3) try your code on a real world dataset. The first part involves no programming, but is crucial for writing an efficient algorithm.

5.1 Time complexity and making your code fast [14 points]

In class, you derived the update rules for coordinate descent, which is shown in Algorithm 1 (including the update term for w_0). In this part of question, we will analyze the algorithm and discuss how to make it fast. There are two key points: utilizing sparsity and caching your predictions. Assume we are using a sparse matrix representation, so that a dot product takes time proportional to the number of non-zero entries. In the following questions, your answers should take advantage of the sparsity of $\mathbf{X}$ when possible.

1. *(2 points)* Define $\hat{\mathbf{y}}_i = \mathbf{X}_i \mathbf{w} + w_0$. Simplify the update rules for w_0 and the computation for c_k in Algorithm 1 using $\hat{\mathbf{y}}$. (Hint, there should no longer be a sum over j).

2. *(2 points)* Let $\|X\|_0$ be the number of nonzero entries in $\mathbf{X}$. What is the time complexity to compute $\hat{\mathbf{y}}$?

3. *(2 points)* What is the time complexity to update w_0 when $\hat{\mathbf{y}}$ is not already computed? What if $\hat{\mathbf{y}}$ is already computed? (assume you can access $\hat{\mathbf{y}}$ with no extra cost)

4. *(2 points)* Let $z_j = \sum_i I(\mathbf{X}_{ij} \neq 0)$ be the number of nonzero elements in j-th column of $\mathbf{X}$. What is the time complexity to update $\mathbf{w}_j$ when $\hat{\mathbf{y}}$ is already computed?

5. *(2 points)* Let $\hat{\mathbf{y}}_i^{(t)} = \mathbf{X}_i \mathbf{w}^{(t)} + w_0^{(t)}$, and assume we update $w_0^{(t)}$ to $w_0^{(t+1)}$ using the rule above. Let $\hat{\mathbf{y}}_i^{(t+1)} = \mathbf{X}_i \mathbf{w}^{(t)} + w_0^{(t+1)}$ be the new prediction after updating. Express $\hat{\mathbf{y}}^{(t+1)}$ in terms of $\hat{\mathbf{y}}^{(t)}$. What is the complexity to calculate $\hat{\mathbf{y}}^{(t+1)}$ when $\hat{\mathbf{y}}^{(t)}$ is already computed?

Algorithm 2: Efficient Coordinate Descent Algorithm

while *not converged* **do**
  $\hat{\mathbf{y}} \leftarrow \mathbf{X}\mathbf{w} + w_0$ (re-calculate $\hat{\mathbf{y}}$ each iteration to avoid numerical drift)
  apply update rule of w_0 you derived in Problem 5.1 Q1
  update $\hat{\mathbf{y}}$ using results in Problem 5.1 Q5
  for $k \in \{1, 2, \cdots d\}$ **do**
    apply update rule of $\mathbf{w}_k$ you derived in Problem 5.1 Q1
    update $\hat{\mathbf{y}}$ using results in Problem 5.1 Q6
  end
end

6. *(2 points)* Let $\hat{\mathbf{y}}_i^{(t)} = \mathbf{X}_i \mathbf{w}^{(t)} + w_0^{(t)}$, and assume we update $\mathbf{w}_k^{(t)}$ to $\mathbf{w}_k^{(t+1)}$ using the rule above. Let $\hat{\mathbf{y}}_i^{(t+1)} = \sum_{j \neq k} \mathbf{w}_j^{(t)} \mathbf{X}_{ij} + \mathbf{w}_k^{(t+1)} \mathbf{X}_{ik} + w_0^{(t)}$ be the new prediction after updating. Express $\hat{\mathbf{y}}^{(t+1)}$ in terms of $\hat{\mathbf{y}}^{(t)}$. What is the complexity to calculate $\hat{\mathbf{y}}^{(t+1)}$, when $\hat{\mathbf{y}}^{(t)}$ is already computed?

7. *(2 points)* Putting this all together, you get the efficient coordinate descent algorithm in Algorithm 2. What is per iteration complexity of this algorithm (cost of each round of the while loop)?

5.2 Implement coordinate descent to solve the Lasso

Now we are ready to implement the coordindate descent algorithm in Algorithm 2. Your code must

- Include an offset term w_0 that is not regularized
- Take optional initial conditions for $\mathbf{w}$ and w_0
- Be able to handle sparse $\mathbf{X}$. It is ok for your code not being able to handle dense $\mathbf{X}$. In Python, this means you can always assume input is `scipy.sparse.csc_matrix`.
- Avoid unnecessary computation (i.e take advantage of what you learned from Problem 5.1)

You may use any language for your implementation, but we recommend Python. Python is a very useful language, and you should find that Python achieves reasonable enough performance for this problem. You may use common computing packages (such as NumPy or SciPy), but please, do not use an existing Lasso solver.

Before you get started, here are some hints that you may find helpful:

- With the exception of computing objective values or initial conditions, the only matrix operations required are adding vectors, multiplying a vector by a scalar, and computing the dot product between two vectors. Try to use as much vector/matrix computation as possible.
- The most important check is to ensure the objective value is nonincreasing with each step.
- To ensure that a solution $(\hat{\mathbf{w}}, \hat{w}_0)$ is correct, you also can compute the value

$$2\mathbf{X}^T(\mathbf{X}\hat{\mathbf{w}} + \hat{w}_0 - \mathbf{y})$$

  This is a d-dimensional vector that should take the value $-\lambda \text{sign}(\hat{\mathbf{w}}_j)$ at j for each $\hat{\mathbf{w}}_j$ that is nonzero. For the zero indices of $\hat{\mathbf{w}}$, this vector should take values lesser in magnitude than λ. (This is similar to setting the gradient to zero, though more complicated because the objective function is not differentiable.)
- It is up to you to decide on a suitable stopping condition. A common criteria is to stop when no element of $\mathbf{w}$ changes by more than some small δ during an iteration. If you need your algorithm to run faster, an easy place to start is to loosen this condition.

- For several problems, you will need to solve the Lasso on the same dataset for many values of λ. This is called a regularization path. One way to do this efficiently is to start at a large λ, and then for each consecutive solution, initialize the algorithm with the previous solution, decreasing λ by a constant ratio until finished.
- The smallest value of λ for which the solution $\hat{\mathbf{w}}$ is entirely zero is given by

$$\lambda_{max} = 2 \left\| \mathbf{X}^T (y - \bar{y}) \right\|_\infty$$

  This is helpful for choosing the first λ in a regularization path.

Finally here are some pointers toward useful parts of Python:

- `numpy`, `scipy.sparse`, and `matplotlib` are useful computation packages.
- For storing sparse matrices, the `scipy.sparse.csc_matrix` (compressed sparse column) format is fast for column operations.
- Important note for numpy users, `scipy.sparse.csc_matrix` uses matrix semantics instead of `numpy.ndarray`. Please refer to `http://wiki.scipy.org/NumPy_for_Matlab_Users` for difference between numpy matrix and ndarray. Specifically, the * operation is matrix multiplication instead of the elementwise product.
- See the short note on `scipy.sparse.csc_matrix` in `guide_csc_matrix.py` to walk you through the necssary features you need.
- `scipy.io.mmread` reads sparse matrices in Matrix Market Format.
- `numpy.random.randn` is nice for generating random Gaussian arrays.
- `numpy.linalg.lstsq` works for solving unregularized least squares.
- If you're new to Python but experienced with Matlab, consider reading NumPy for Matlab Users at `http://wiki.scipy.org/NumPy_for_Matlab_Users`.

5.3 Try out your work on synthetic data [12 points]

We will now try out your solver with some synthetic data. A benefit of the Lasso is that if we believe many features are irrelevant for predicting $\mathbf{y}$, the Lasso can be used to enforce a sparse solution, effectively differentiating between the relevant and irrelevant features.

Let's see if it actually works. Suppose that $\mathbf{x} \in \mathbb{R}^d, y \in \mathbb{R}, k < d$, and pairs of data $(\mathbf{x}_i, y_i)$ are generated independently according to the model

$$y_i = w_0^* + w_1^* x_{i,1} + w_2^* x_{i,2} + \ldots w_k^* x_{i,k} + \epsilon_i$$

where $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ is some Gaussian noise. Note that since $k < d$, the features $k+1$ through d are unnecessary (and potentially even harmful) for predicting y.

With this model in mind, you are now tasked with the following:

1. *(6 points)* Let $N = 50, d = 75, k = 5$, and $\sigma = 1$. Generate some data by drawing each element of $\mathbf{X} \in \mathbb{R}^{N \times d}$ from a standard normal distribution $\mathcal{N}(0, 1)$. Let $w_0^* = 0$ and create a $\mathbf{w}^*$ by setting the first k elements to ± 10 (choose any sign pattern) and the remaining elements to 0. Finally, generate a Gaussian noise vector ϵ with variance σ^2 and form $\mathbf{y} = \mathbf{X}\mathbf{w}^* + w_0^* + \epsilon$.

   With your synthetic data, solve multiple Lasso problems on a regularization path, starting at λ_{max} and decreasing λ by a constant ratio until few features are chosen correctly. Compare the sparsity pattern of your Lasso solution $\hat{\mathbf{w}}$ to that of the true model parameters $\mathbf{w}^*$. Record values for precision (number of correct nonzeros in $\hat{\mathbf{w}}$/total number of nonzeros in $\hat{\mathbf{w}}$) and recall (number of correct nonzeros in $\hat{\mathbf{w}}$/k).

   How well are you able to discover the true nonzeros? Comment on how λ affects these results and include plots of precision and recall vs. λ.

2. *(6 points)* Change σ to 10, regenerate the data, and solve the Lasso problem using a value of λ that worked well when $\sigma = 1$. How are precision and recall affected? How might you change λ in order to achieve better precision or recall?

5.4 Become a data scientist at Yelp [24 points]

We'll now put the Lasso to work on some real data. Recently Yelp held a recruiting competition on the analytics website Kaggle. Check it out at `http://www.kaggle.com/c/yelp-recruiting`. (As a side note, browsing other competitions on the site may also give you some ideas for class projects.)

For this competition, the task is to predict the number of useful upvotes a particular review will receive. For extra fun, we will add the additional task of predicting the review's number of stars based on the review's text alone.

For many Kaggle competitions (and machine learning methods in general), one of the most important requirements for doing well is the ability to discover great features. We can use our Lasso solver for this as follows. First, generate a large amount of features from the data, even if many of them are likely unnecessary. Afterward, use the Lasso to reduce the number of features to a more reasonable amount.

Yelp provides a variety of data, such as the review's text, date, and restaurant, as well as data pertaining to each business, user, and check-ins. This information has already been preprocessed for you into the following files:

`upvote_data.csv`	Data matrix for predicting number of useful votes
`upvote_labels.txt`	List of useful vote counts for each review
`upvote_features.txt`	Names of each feature for interpreting results
`star_data.mtx`	Data matrix for predicting number of stars
`star_labels.txt`	List of number of stars given by each review
`star_features.txt`	Names of each feature

For each task, data files contain data matrices, while labels are stored in separate text files. The first data matrix is stored in CSV format, each row corresponding to one review. The second data matrix is stored in Matrix Market Format, a format for sparse matrices. Meta information for each feature is provided in the final text files, one feature per line. For the upvote task, these are functions of various data attributes. For the stars task, these are strings of one, two, or three words (n-grams). The feature values correspond roughly to how often each word appears in the review. All columns have also been normalized.

To get you started, the Python following code should load the data:

```
import numpy as np
import scipy.io as io
import scipy.sparse as sparse

# Load a text file of integers:
y = np.loadtxt("upvote_labels.txt", dtype=np.int)

# Load a text file of strings:
featureNames = open("upvote_features.txt").read().splitlines()

# Load a csv of floats:
A = np.genfromtxt("upvote_data.csv", delimiter=",").tocsc()

# Load a matrix market matrix, convert it to csc format:
B = io.mmread("star_data.mtx").tocsc()
```

For this part of the problem, you have the following tasks:

1. *(6 points)* Solve lasso to predict the number of useful votes a Yelp review will receive. Use the first 4000 samples for training, the next 1000 samples for validation, and the remaining samples for testing.

Starting at λ_{max}, run Lasso on the training set, decreasing λ using previous solutions as initial conditions to each problem. Stop when you have considered enough λ's that, based on validation error, you can choose a good solution with confidence (for instance, when validation error begins increasing or stops decreasing significant). At each solution, record the root-mean-squared-error (RMSE) on training and validation data. In addition, record the number of nonzeros in each solution.

Plot the RMSE values together on a plot against λ. Separately plot the number of nonzeros as well.

2. *(3 points)* Find the λ that achieves best validation performance, and test your model on the remaining set of test data. What RMSE value do you obtain?

3. *(3 points)* Inspect your solution and take a look at the 10 features with weights largest in magnitude. List the names of these features and their weights, and comment on if the weights generally make sense intuitively.

4. *(12 points)* Repeat part 1, 2, 3 using the data matrix and labels for predicting the score of a review. Use the first 30,000 examples for training and divide the remaining samples between validation and testing as before.