CIS 521: HW 7 - Machine Learning Supplemental

1 Hints, Overview of Programming

1.1 Dataset

First and foremost, we recommend taking a small sample of the datasets (say the first 100 lines or something similar) to make sure your algorithms work the way you expect before continuing to run them on the larger datasets.

The dataset consists of many examples of postings from a web forum. The `Dataset` class converts these postings into a numeric form that we can run our methods on. Each *example* is a single posting, which can be anywhere from 50-500 words or so. Each word is converted to a unique number, which you can see from the `dataset.getWordlist()` method. (That returns a list where the j'th element of the list is the word that's numbered j.) Finally, each example is converted to a row $\mathbf{x}_i$ of a data matrix $\mathbf{X}$, so that the j'th column is a 1 if the word occured in the example or 0 otherwise. For example, the posting "my pc sucks," if "my" was word 5, "pc" was word 152, and "sucks" was word 784, would be a vector $\mathbf{x}_i$ with zeros in all columns except for 5, 152, and 784. To indicate which newsgroup each post came from, `dataset` creates another vector $\mathbf{y}$ where $y_i = 1$ if the example came from the first file and $y_i = -1$ if the example came from the second file.

To evaluate each algorithm, you need to give it some material to study and then a *separate* set of material to test it on. (Otherwise, it would be able to just memorize the answers to the test.) This is the purpose of the `getTrainAndTestSets` method; it randomly splits the entire dataset into some fraction training and testing (specified as a parameter). However, just as if you were only given irrelevant parts of a textbook to study for an exam, we might accidentally split the data into a training set that contains very little information that's useful for the test set. The `seed` parameter is used to seed the random number generator, so that any two calls to `getTrainAndTestSet` with the same seed will result in the same random split; if it's a hard split, it will be hard for all algorithms so long as the seed is the same. To average out the inherent difficulty of the random split, you need to run everything with 5 different seeds and average the resulting numbers.

1.2 NumPy & Naive Bayes

NumPy has a lot of advantages over regular Python that dramatically reduce the lines of code needed for simple operations. For naive Bayes, we need to compute something like:

$$\hat{p}(X_j|Y == 1) = \frac{\#[X_j == 1, Y == 1]}{\#[Y == 1]}$$

With NumPy, we can do this for all columns simultaneously in a few lines; I won't bother with that, but at the very least, know that writing the code `Ytrain==1` will return an array of the same length as `Ytrain` thats `True` where `Ytrain` is 1 and `False` otherwise. We can then use this to *index* into any dimension of any variable with the same length in that dimension, selecting elements where `True` and rejecting them where `False`. So, we can do compute the above with the simple code: `np.sum( X[Y==1,j] ) / np.sum(Y==1)`. *Bam.*

1.3 Regression etc.

Linear regression is defined as the following. For a given input $\mathbf{x}_i$, our guess for the output is a linear combination of the features, $\hat{y}_i = \mathbf{x}_i \cdot \mathbf{w}$, where w_j is the weight of word j. Hey, you ask, but we know from the above that y_i is only going to be -1 or +1, while $\mathbf{x}_i \cdot \mathbf{w}$ could be any real number. That's true, and in fact linear regression is not the ideal method to solve this problem, but it's one of the easiest algorithms to implement once you know what to do. The necessary "fix" is to say, let's predict the *sign* of $\mathbf{x}_i \cdot \mathbf{w}$, since that will always be -1 or +1, and that's what you should do for this problem. (Note that `np.sign` may also return zero so you shouldn't use that function for this.)

The next question is, how do we find $\mathbf{w}$? One way is to try to minimize the number of mistakes we make on the training set when we use $\hat{y}_i = \text{sign}(\mathbf{x}_i \cdot \mathbf{w})$. The problem is there's no simple equation to solve for $\mathbf{w}$ when we do that; the closing thing we've discussed is an iterative approach called the perceptron algorithm (which you also need to implement for this assignment). Instead, the easy problem to solve is to minimize the squared residual:

$$\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2,$$

where $\mathbf{X}\mathbf{w} - \mathbf{y}$ is a vector where the i'th element is the difference $\mathbf{x}_i \cdot \mathbf{w} - y_i$, called the *residual*. By taking the derivative with respect to $\mathbf{w}$ and setting to zero (as in standard calculus), we get the following solution for $\mathbf{w}$:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

Note that if y_i was just a real number and not -1 or +1, then minimizing the squared residuals makes perfect sense; here, it is suboptimal because of the "take the sign" trick we're using to actually compute predictions.

However, sometimes that equation is unsolvable, and sometimes we want to prevent overfitting. To do this, we minimize a *penalized residual:*

$$\min_{\mathbf{w}} ||\mathbf{X}\mathbf{w} - \mathbf{y}||_2^2 + \lambda ||\mathbf{w}||_2^2,$$

where λ is a pre-determined constant. This says, "minimize the residuals but ALSO try to make $\mathbf{w}$ as close to zero as possible." As λ increases, $\mathbf{w}$ gets closer to zero, "hedging our bet" and maybe preventing some overfitting. Eventually if λ gets too large, $\mathbf{w}$ will get so close to zero it will stop predicting anything accurately, so choosing λ is finding the right trade-off between fitting the training set and "hedging our bets." For this assignment, you can simply play around with a few λ values and choose one to use for *all* regression methods, just say which values you tried and which you chose in your report. Now, if we take the derivative again and solve for $\mathbf{w}$, we get the following equation:

$$\mathbf{w} = (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y}$$

This is the only linear regression equation you need to implement to train your method, and you can do it in one line using NumPy. In NumPy, if you convert `Xtrain` and `Ytrain` to matrices using

`np.asmatrix`, then you can simply write `Xtrain.T` for transpose, use the standard multiplication operation, and use `np.linalg.inv` to take the inverse. Then, to generate predictions for ANY input `X`, you can simply run `X*w`, so long as you've saved `w` from training somewhere.

1.4 Step/streamwise regression

The important bit here is to realize that all you're doing is doing a search to choose which columns of $\mathbf{X}$ are the most useful. It's key to take advantage of NumPy. Given a list of columns `cols`, you can write `Xtrain[:, cols]` to get a slice of $\mathbf{X}$ with only those columns listed in `cols` and in order that they are in `cols`. E.g., if `cols = [121, 3, 67]`, then `Xtrain[:,cols]` woudl give you a array object with three columns corresponding to columns 121, 3, and 67 respectively. **So, for both methods, simply maintain a list of columns you've chosen so far, and use that list at test time when you need to make predictions.**

A **rough** outline for streamwise regression is simple:

1. Initialize `cols` to be empty and a current error to be 100%.
2. For $j = 0 \ldots f$, where f is the number of columns
   (a) Append j to cols
   (b) Learn `w = trainRidge(Xtrain[:,cols], Ytrain, lambda)`
   (c) Compute *training error* of `w`
   (d) If the training error is not an improvement by some threshold, pop j off of cols so we don't include it in the regression.

After the end of this loop, you will have chosen some subset of the columns to use in your regression, so train a final `w` and use that for assessing the training and test error.

A **rough** outline for stepwise regression is only slightly different:

1. Initialize `cols` to be empty and a current error to be 100%.
2. Repeat until a certain number of features are chosen or no features are an improvement:
   (a) For $j = 0 \ldots f$, where f is the number of columns
      i. Append j to cols
      ii. Learn `w = trainRidge(Xtrain[:,cols], Ytrain, lambda)`
      iii. Compute *training error* of `w` and keep track of whether or not j is the *best* improvement over all columns so far.
      iv. Pop j from cols
   (b) Append the *best* j to cols, or stop if no j is an improvement.

So, stepwise has to run f regressions each time it adds *one* column to the currently selected list (you can be smarter and not check the same column twice, but whatever). It will therefore be a lot slower, and you will probably want to only run it to somewhere between 40-100 features unless you've got a very fast computer.