CPSC-340 Machine Learning 2012

Homework # 5

NAME:____________________________________

Signature:__________________________________

STD. NUM: ________________________________

1 Regularized linear regression

1.1 Standardize the data.

Download the prostate cancer dataset from the course website. In this prostate cancer study 9 variables – including age, log weight, log cancer volume, etc. – were measured for 97 patients. We will now construct a model to predict the 9th variable a linear combination of the other 8. A description of this dataset appears in the textbook of Hastie et al, freely available on the course website:

"*The data for this example come from a study by Stamey et al. (1989) that examined the correlation between the level of prostate specific antigen (PSA) and a number of clinical measures, in 97 men who were about to receive a radical prostatectomy. The goal is to predict the log of PSA (lpsa) from a number of measurements including log cancer volume (lcavol), log prostate weight lweight, age, log of benign prostatic hyperplasia amount lbph, seminal vesicle invasion svi, log of capsular penetration lcp, Gleason score gleason, and percent of Gleason scores 4 or 5 pgg45.* "

1. First load the data and split it into a response vector (`y`) and a matrix of attributes (`X`),

   ```
   X = np.loadtxt('prostate.data', skiprows=1)
   y = X[:,-1]
   X = X[:,0:-1]
   ```

2. Choose the first 50 patients as the training data. The remaining patients will be the test data.

   ```
   ytrain, ytest = y[0:50], y[50:]
   Xtrain, Xtest = X[0:50], X[50:]
   ```

3. Set both variables to have zero mean and standardize the input variables to have unit variance.

   ```
   Xbar = np.mean(Xtrain, axis=0)
   Xstd = np.std(Xtrain, axis=0)
   ybar = np.mean(ytrain)
   ytrain = ytrain - ybar
   Xtrain = (Xtrain - Xbar) / Xstd
   ```

   Note: here we are using the "broadcasting" feature of numpy, which allows summation of a vector and a matrix. For example:

   ```
   >>> a
   array([[1, 2, 3],
          [4, 5, 6]])
   >>> c
   ```

```
array([7, 8, 9])
>>> a+c
array([[ 8, 10, 12],
       [11, 13, 15]])
```

You will have to be careful when you do this in numpy in order to make the shapes match up as they should. For more details on how to do this properly look up "numpy broadcasting".

Important detail: In this step we are learning the bias θ_0. We will need these exact terms (`Xbar, Xstd, ybar`) for later when we do predictions. Mathematically, what we are saying is that the bias term will be computed separately as follows:

$$\theta_0 = \bar{y} - \bar{\mathbf{x}}^T \widehat{\boldsymbol{\theta}}$$

where $\bar{y}$ is the mean of the elements of the **training data** vector $\mathbf{y}$ and $\bar{\mathbf{x}}^T$ is the vector of 8 means for the input attributes. Note that in this case the 8-dimensional parameter vector $\widehat{\boldsymbol{\theta}}$ includes all the parameters other than the bias term that have been learned with either ridge or lasso. That is, we first learn $\widehat{\boldsymbol{\theta}}$ using standardized data and then proceed to learn θ_0.

When we encounter a new input $\mathbf{x}^\star$ in the test set, we need to standardize it before making a prediction. The actual prediction should be:

$$\widehat{y} = \bar{y} + \sum_{j=1}^{8} \frac{x_j^\star - \bar{x}_j}{\sigma_j} \widehat{\theta}_j$$

where $\bar{x}_j$ and σ_j are the mean and standard deviation of the j-th attribute **obtained from the training data.**

One reason for standardizing the inputs is that we want them to be comparable. Had we had an input much bigger than the other, we would have wanted to apply a different regularizer to it. By standardizing the inputs first, we only need a single scalar regularization coefficient δ^2.

1.2 Ridge regression

We will now construct a model using ridge regression to predict the 9th variable as a linear combination of the other 8.

1. Write code for ridge regression starting from the following skeleton:

   ```
   def ridge(X, y, d2):
       ???
       return theta
   ```

   Compute the ridge regression solutions for a range of regularizers (δ^2). Plot the values of each $\boldsymbol{\theta}$ in the y-axis against δ^2 in the x-axis. This set of plotted values is known as a regularization path. Your plot should look like Figure 1. *Hand in your version of this plot, along with the code you used to generate it.*

2. For each computed value of $\boldsymbol{\theta}$, compute the train and test error. Remember, you will have to standardize your test data with the same means and standard deviations as above (`Xbar, Xstd, ybar`) before you can make a prediction and compute your test error. In other words, to make a prediction do:

   ```
   yhat = ybar + numpy.dot((Xtest - Xbar) / Xstd, theta)
   ```

   Choose a value of δ^2 using cross-validation. What is this value? Show all your intermediate cross-validation steps and the criterion you used to choose δ^2. Plot the train and test errors as a function of δ^2. Your plot should look like Figure 2. *Hand in your version of this plot, along with the code used to generate it.*

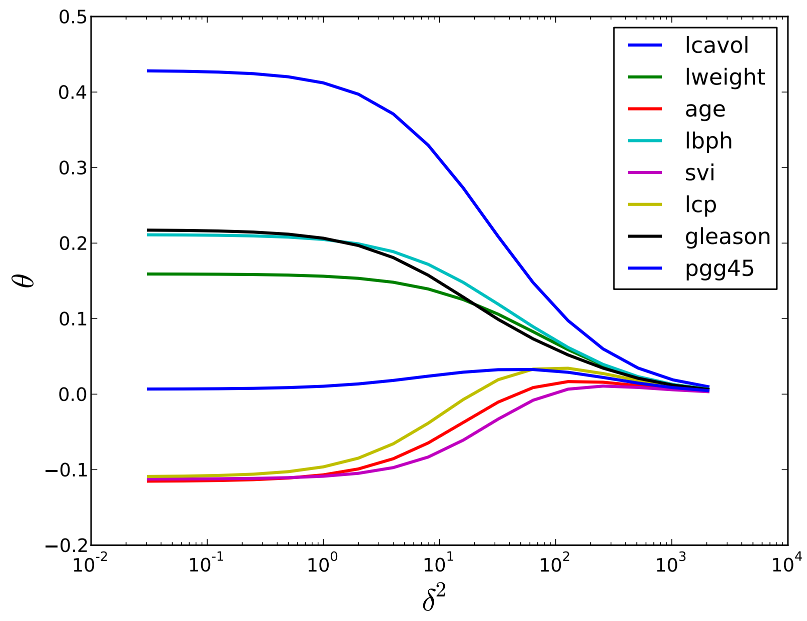

Figure 1: Regularization path for ridge regression.

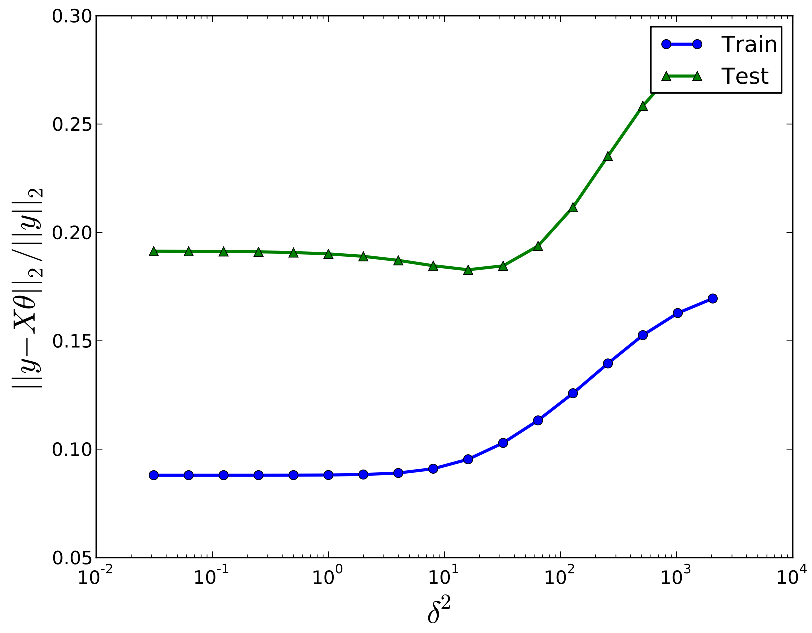

Figure 2: Relative error of the ridge estimator against regularization parameter δ^2.

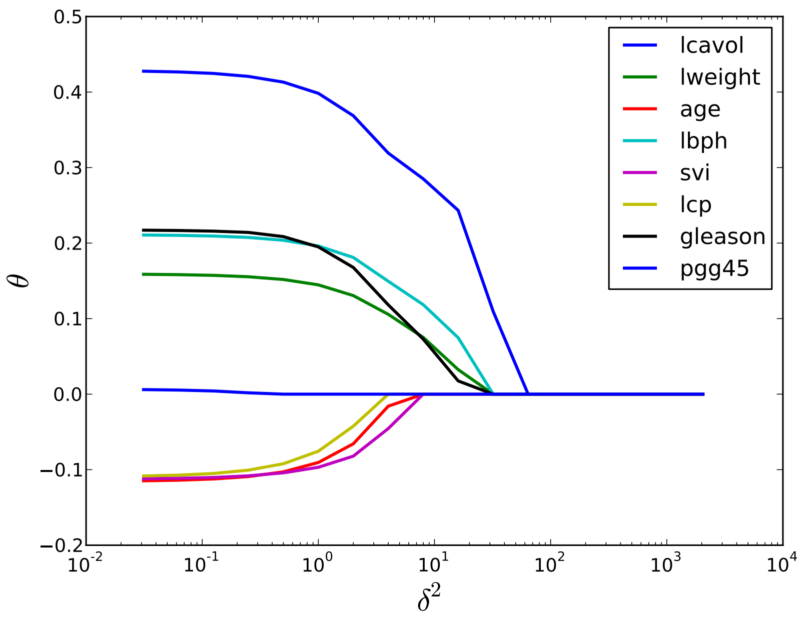

Figure 3: Regularization path for lasso.

1.3 Lasso

We will now implement the Lasso and try this code out on the prostate cancer data.

1. Implement the coordinate descent (aka "shooting" method) for solving Lasso. Pseudo-code for this solver is given in the slides. You should start with the following skeleton code:

```
def lasso(X, y, d2):
    theta_ = np.zeros(k)
    theta = ridge(X, y, d2)
    while np.sum(np.abs(theta-theta_)) > 1e-5:
        theta_ = theta.copy()
        ???
    return theta
```

2. Find the solutions and generate the two plots above, but now using this new Lasso solver. Your two plots should look like Figures 3 and 4. Once again choose δ^2 by cross-validation and write down all your steps. What is the optimal value of δ^2? Which parameters (and hence inputs) are active for this choice of regularization parameter? That is, which are the non-zero thetas?

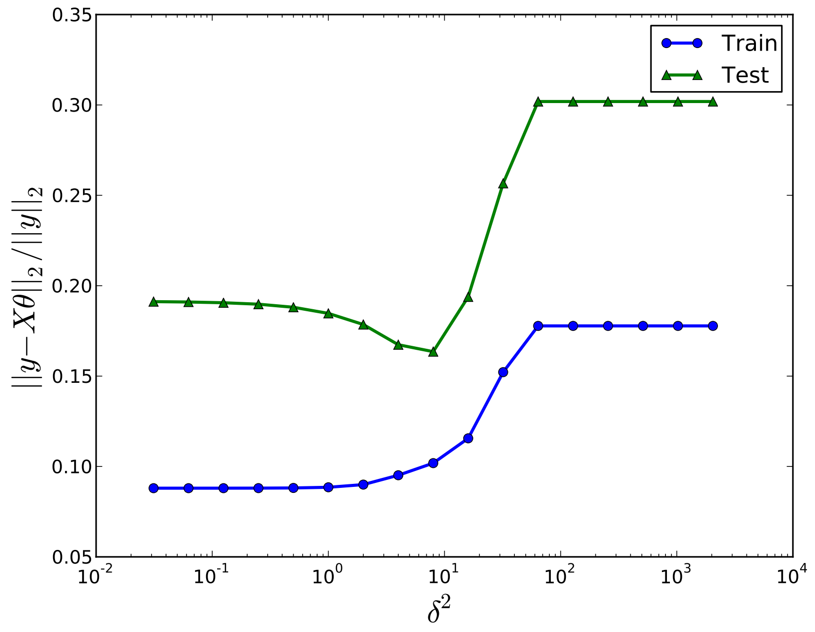

Figure 4: Relative error of the lasso estimator against regularization parameter δ^2.

2 Twitter sentiment classification

In this question, you will be given data consisting of a large number of tweets and be asked to build a classifier which will determine the sentiment of any new tweets we give you as either positive or negative.

In particular, the data we give you will contain one million tweets and approximately ten thousand binary features. The data will be a sparse matrix (i.e. most of its elements will be zero) created using methods from the package `scipy.sparse`. Once you have the datafile `tweets.mtx` (via a link at the end of this document) you can load it using the following commands:

```
import scipy.io
data = scipy.io.mmread('tweets.mtx').tocsr()
```

This just loads the data and converts it into Compressed Sparse Row (CSR) format. This format is just a more efficient way to access the data, but for more details see the `scipy.sparse` documentation (again, linked to at the end of this document).

The format of the data is also pretty basic: each row coincides with one tweet where the first column is the label of that tweet (i.e. positive sentiment or negative) and the remaining columns are the binary features. We can split this up into our standard input/label matrices via:

```
X = data[:,1:]
y = data[:,0]
```

You may also want to treat `y` as a standard binary vector, which we can do via:

```
y = np.array(y.todense()).flatten()
```

However, while `y` is small enough that this should be fine, **do not do this** to `X`! A fully dense `X` will probably take up more memory than your computer can handle.

At this point we should probably also mention what you can do with sparse matrices. These objects are treated as matrices, so if we wanted to take the dot-product between `y` and `X` we could write this as just y*X—assuming that we converted y into a standard array. This operation is quite fast (also, think about what this dot-product actually means) so you should try and use as many of these sparse matrix operations as you can.

Note: since all of the data is binary we can get, for example, the negation of y by computing 1-y. But we've already said that y should be small enough that this dense vector will fit into memory. You might be tempted to look at 1-X, i.e. the set of all features that were not "on". **Do not do this!** This matrix will be fully dense, and as we mentioned earlier *bad things will happen.*

Finally, while this is all the data we will give you, we are keeping some extra data to test your classifier on. It is up to you to decide how to test your solution, whether to perform cross-validation, etc.

The testing of your code will be fairly simple too. Once you turn in your code (see the next section!) we will run your code on our held-out dataset and determine the accuracy of your classifier on these tweets. If you get greater than 75% accuracy you will get full marks. Greater than 65% accuracy will net you half marks. Anything below this will receive zero marks. Note, however, that just randomly guessing should get about 50% accuracy.

2.1 What you need to implement

For this assignment we'll need you to electronically turn in your python code (details on that shortly!) so that we can test your classifier. In particular we will have you turn in a file classify.py which contains a class MyClassifier. The majority of the work done by this class should be contained in the methods fit(X, y) and predict(X) where the parameters are a sparse matrix and a dense vector as noted in the previous section.

We will also require your class to have a method load_params(fname) which will load parameters for your model from a file in .npz format. For more information on this format see numpy.savez. Below we have implemented a class which you can use as boiler-plate code (or inherit from) which implements loading/saving functionality:

```
class Classifier(object):
    def __init__(self):
        self.params = []

    def fit(self, X, y):
        raise NotImplementedError

    def predict(self, X):
        raise NotImplementedError

    def save_params(self, fname):
        params = dict([(p, getattr(self, p)) for p in self.params])
        np.savez(fname, **params)

    def load_params(self, fname):
        params = np.load(fname)
        for name in self.params:
            setattr(self, name, params[name])
```

With this class you can then populate the list self.params with the names of your parameters (as strings) inside your initialization method. Then after fitting the data you can call save_params to save the learned parameters.

We will then require you to turn in a params.npz file created in this way. This allows us to not restrict the amount of time your classifier takes to learn (so long as it's done before the deadline!) and you can make your classifier as complex as it needs to be.

An example classifier which conforms to this interface, but would obviously get no points, could be written as:

```
class RandomClassifier(Classifier):
    def fit(self, X, y):
        pass

    def predict(self, X):
        return np.random.randint(2, size=X.shape[0])
```

wheras an implementation of Naive Bayes might look something like:

```
class NBayes(Classifier):
    def __init__(self):
        self.params = ['logpi', 'logtheta']

    def fit(self, X, y):
        # implementation here...
        self.logpi = ...
        self.logtheta = ...

    def predict(self, X):
        # implementation here...
```

2.2 What you need to turn in

In summary, **you must turn in** the following:

1. a classifier implemented as the class `MyClassifier` included in the file `classify.py` and conforming to the previously described interface.
2. a set of parameters saved in the file `params.npz`.

Once you have turned this in, **our testing procedure** will perform the following code:

```
from classify import MyClassifier
classifier = MyClassifier()
classifier.load_params('params.npz')
yhat = classifier.predict(Xtest)
accuracy = np.sum(yhat == ytest) / ytest.size
```

where `Xtest` is our holdout data, in the same format as the matrix `X` from the previous section. So ensure that your code can do this much.

3 The contest (for bonus points)

We will also be running a contest, throughout the rest of the course, to see which student can get the highest accuracy on the twitter dataset using whichever classifier they come up with. First, second, and third places will be awarded 20, 10, and 5 bonus marks on the **overall grade** respectively. Ties will be broken randomly.

The first contest turn-in date on **November 23rd** will allow you to try out your code and see where you stand in the rankings. The second, final, turn-in date is **November 26th**.

4 Links and extra reading

1. `https://www.cs.ubc.ca/~bshakibi/tweets.mtx`: the data.
2. `http://www.scipy.org/SciPyPackages/Sparse`: documentation on scipy's sparse matrix facilities.
3. `http://docs.scipy.org/doc/numpy/reference/generated/numpy.savez.html`: documentation on the numpy function for saving `.npz` files.