CPSC-340 Machine Learning and Data Mining 2012

Homework # 6

1 Twitter sentiment classification

The specification for your final homework is fairly simple: 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 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...
```

3 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.

The assignment will be due on the **Monday, April 2nd**.

4 The contest (for bonus points)

We will also be running a contest to see which student can get the highest accuracy on this dataset. First, second, and third places will be awarded 20, 10, and 5 bonus marks on the **overall grade** respectively. Ties will be broken randomly.

We will also have **intermediate turnin dates** on March 26th and March 30th to allow for you to try out your code and see where you stand in the rankings. However, you don't need to turn anything until the 2nd, and only your last handin will count.

5 Links and extra reading

1. `http://www.cs.ubc.ca/~hoffmanm/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.