Project 1: Classification

Re-submit Assignment

Due Mar 3 by 11:55pm **Points** 100 **Submitting** a file upload
File Types py, txt, and pdf **Available** after Feb 15 at 10:30am

Project 1: Classification

The goal of this project is to implement from scratch the main Machine Learning techniques we have learned so far.

Introduction

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download each of these individually, or you can head to the **Files page**, select the p1 folder and click the Download as Zip button.

Files You'll Edit

dumbClassifiers.py: This contains a handful of "warm up" classifiers to get you used to our classification framework.

dt.py: Will be your simple implementation of a decision tree classifier.

knn.py: This is where your nearest-neighbor classifier modifications will go.

perceptron.py: Take a guess :).

Files you might want to look at

binary.py: Our generic interface for binary classifiers (actually works for regression and other types of classification, too).

datasets.py: Where a handful of test data sets are stored.

util.py: A handful of useful utility functions: **these will undoubtedly be helpful to you, so take a look!**

runClassifier.py: A few wrappers for doing useful things with classifiers, like training them, generating learning curves, etc.

mlGraphics.py: A few useful plotting commands

`data/*`: all of the datasets we'll use.

What to Submit

You will hand in all of the python files listed above under "Files you'll edit" as well as a partners.txt file that lists the **names** and **last four digits of the UID** of all members in your team. Finally, you'll hand in a **writeup.pdf** file that answers all the written questions in this assignment (denoted by **WU#** in this file).

Autograding

Your code will be autograded for technical correctness. Please **do not** change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. However, the correctness of your implementation -- not the autograder's output -- will be the final judge of your score. If necessary, we will review and grade assignments individually to ensure that you receive due credit for your work.

Warming up to Classifiers (10%)

Let's begin our foray into classification by looking at some very simple classifiers. There are three classifiers in `dumbClassifiers.py`, one is implemented for you, the other two you will need to fill in appropriately.

The already implemented one is `AlwaysPredictOne`, a classifier that (as its name suggest) always predicts the positive class. We're going to use the `TennisData` dataset from `datasets.py` as a running example. So let's start up python and see how well this classifier does on this data. You should begin by importing `util`, `datasets`, `binary` and `dumbClassifiers`. Also, be sure you always have `from numpy import *` and `from pylab import *`. You can achieve this with `from imports import *` to make life easier.

```
>>> h = dumbClassifiers.AlwaysPredictOne({})
>>> h
AlwaysPredictOne
>>> h.train(datasets.TennisData.X, datasets.TennisData.Y)
>>> h.predictAll(datasets.TennisData.X)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

Indeed, it looks like it's always predicting one!

Now, let's compare these predictions to the truth. Here's a very clever way to compute accuracies (**WU1:** why is this computation equivalent to computing classification accuracy?):

```
>>> mean((datasets.TennisData.Y > 0) == (h.predictAll(datasets.TennisData.X) > 0))
0.6428571428571429
```

That's training accuracy; let's check test accuracy:

```
>>> mean((datasets.TennisData.Yte > 0) == (h.predictAll(datasets.TennisData.Xte) > 0))
0.5
```

Okay, so it does pretty badly. That's not surprising, it's really not learning anything!!!

Now, let's use some of the built-in functionality to help do some of the grunt work for us. You'll need to import `runClassifier`.

```
>>> runClassifier.trainTestSet(h, datasets.TennisData)
Training accuracy 0.642857, test accuracy 0.5
```

Very convenient!

Now, your first implementation task will be to implement the missing functionality in `AlwaysPredictMostFrequent`. This actually will "learn" something simple. Upon receiving training data, it will simply remember whether +1 is more common or -1 is more common. It will then always predict this label for future data. Once you've implemented this, you can test it:

```
>>> h = dumbClassifiers.AlwaysPredictMostFrequent({})
>>> runClassifier.trainTestSet(h, datasets.TennisData)
Training accuracy 0.642857, test accuracy 0.5
>>> h
AlwaysPredictMostFrequent(1)
```

Okay, so it does the same as `AlwaysPredictOne`, but that's because +1 is more common in that training data. We can see a difference if we change to a different dataset: `SentimentData` is the data you've seen before, now Python-ified.

```
>>> runClassifier.trainTestSet(dumbClassifiers.AlwaysPredictOne({}), datasets.SentimentData)
Training accuracy 0.504167, test accuracy 0.5025
>>> runClassifier.trainTestSet(dumbClassifiers.AlwaysPredictMostFrequent({}), datasets.Senti
mentData)
Training accuracy 0.504167, test accuracy 0.5025
```

Since the majority class is "1", these do the same here.

The last dumb classifier we'll implement is `FirstFeatureClassifier`. This actually does something slightly non-trivial. It looks at the first feature (i.e., `X[0]`) and uses this to make a prediction. Based on the training data, it figures out what is the most common class for the case when `X[0] > 0` and the most common class for the case when `X[0] <= 0`. Upon receiving a test point, it checks the value of `X[0]` and returns the corresponding class. Once you've implemented this, you can check it's performance:

```
>>> runClassifier.trainTestSet(dumbClassifiers.FirstFeatureClassifier({}), datasets.TennisDa
ta)
Training accuracy 0.714286, test accuracy 0.666667
>>> runClassifier.trainTestSet(dumbClassifiers.FirstFeatureClassifier({}), datasets.Sentimen
tData)
Training accuracy 0.504167, test accuracy 0.5025
```

Decision Trees (30%)

Our next task is to implement a decision tree classifier. There is stub code in `dt.py` that you should edit. Decision trees are stored as simple data structures. Each node in the tree has a `.isLeaf` boolean that tells us if this node is a leaf (as opposed to an internal node). Leaf nodes have a `.label` field that says what class to return at this leaf. Internal nodes have: a `.feature` value that tells us what feature to split on; a `.left` *tree* that tells us what to do when the feature value is *less than 0.5*; and a `.right` *tree* that tells us what to do when the feature value is *at least 0.5*. To get a sense of how the data structure works, look at the `displayTree` function that prints out a tree.

Your first task is to implement the training procedure for decision trees. We've provided a fair amount of the code, which should help you guard against corner cases. (Hint: take a look at `util.py` for some useful functions for implementing training. Once you've implemented the training function, we can test it on simple data:

```
>>> h = dt.DT({'maxDepth': 1})
>>> h
Leaf 1

>>> h.train(datasets.TennisData.X, datasets.TennisData.Y)
>>> h
Branch 6
```

```
  Leaf 1.0
  Leaf -1.0
```

This is for a simple depth-one decision tree (aka a decision stump). If we let it get deeper, we get things like:

```
>>> h = dt.DT({'maxDepth': 2})
>>> h.train(datasets.TennisData.X, datasets.TennisData.Y)
>>> h
Branch 6
  Branch 7
    Leaf 1.0
    Leaf 1.0
  Branch 1
    Leaf -1.0
    Leaf 1.0

>>> h = dt.DT({'maxDepth': 5})
>>> h.train(datasets.TennisData.X, datasets.TennisData.Y)
>>> h
Branch 6
  Branch 7
    Leaf 1.0
    Branch 2
      Leaf 1.0
      Leaf -1.0
  Branch 1
    Branch 5
      Branch 4
        Leaf -1.0
        Branch 3
          Leaf -1.0
          Leaf 1.0
      Leaf 1.0
    Leaf 1.0

or:
>>> h
Branch 6
  Branch 7
    Leaf 1.0
    Branch 2
      Leaf 1.0
      Leaf -1.0
  Branch 1
    Branch 7
      Branch 2
        Leaf -1.0
        Leaf 1.0
      Leaf -1.0
    Leaf 1.0
```

We can do something similar on the sentiment data (this will take a bit longer):

```
>>> h = dt.DT({'maxDepth': 2})
>>> h.train(datasets.SentimentData.X, datasets.SentimentData.Y)
>>> h
Branch 2428
  Branch 3843
    Leaf 1.0
    Leaf -1.0
  Branch 3893
    Leaf -1.0
    Leaf 1.0
```

The problem here is that words have been converted into numeric ids for features. We can look them up (your results here might be different due to hashing):

```
>>> datasets.SentimentData.words[2428]
'bad'
>>> datasets.SentimentData.words[3843]
'worst'
>>> datasets.SentimentData.words[3893]
'sequence'
```

Based on this, we can rewrite the tree (by hand) as:

```
Branch 'bad'
  Branch 'worst'
    Leaf -1.0
    Leaf 1.0
  Branch 'sequence'
    Leaf -1.0
    Leaf 1.0
```

Now, you should go implement prediction. This should be easier than training! We can test by (this takes about a minute for me):

```
>>> runClassifier.trainTestSet(dt.DT({'maxDepth': 1}), datasets.SentimentData)
Training accuracy 0.630833, test accuracy 0.595
>>> runClassifier.trainTestSet(dt.DT({'maxDepth': 3}), datasets.SentimentData)
Training accuracy 0.701667, test accuracy 0.6175
>>> runClassifier.trainTestSet(dt.DT({'maxDepth': 5}), datasets.SentimentData)
Training accuracy 0.765833, test accuracy 0.625
```

Looks like it does better than the dumb classifiers on training data, as well as on test data! Hopefully we can do even better in the future!

We can use more `runClassifier` functions to generate learning curves and hyperparameter curves:

```
>>> curve = runClassifier.learningCurveSet(dt.DT({'maxDepth': 9}), datasets.SentimentData)
[snip]
>>> runClassifier.plotCurve('DT on Sentiment Data', curve)
```

This plots training and test accuracy as a function of the number of data points (x-axis) used for training and y-axis is accuracy.

WU2: We should see training accuracy (roughly) going down and test accuracy (roughly) going up. Why does training accuracy tend to go *down?* Why is test accuracy not monotonically increasing? You should also see jaggedness in the test curve toward the left. Why?

We can also generate similar curves by changing the maximum depth hyperparameter:

```
>>> curve = runClassifier.hyperparamCurveSet(dt.DT({}), 'maxDepth', [1,2,4,6,8,12,16], datas
ets.SentimentData)
[snip]
>>> runClassifier.plotCurve('DT on Sentiment Data (hyperparameter)', curve)
```

Now, the x-axis is the value of the maximum depth.

WU3: You should see training accuracy monotonically increasing and test accuracy making something like a hill. Which of these is *guaranteed* to happen and which is just something we might expect to happen? Why?

Nearest Neighbors (30%)

To get started with geometry-based classification, we will implement a nearest neighbor classifier that supports both KNN classification and epsilon-ball classification. This should go in `knn.py`. The only function here that you have to do anything about is the `predict` function, which does all the work.

In order to test your implementation, here are some outputs (suggestion: implementing epsilon-balls first, since they're slightly easier):

```
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 0.5}), datasets.TennisData)
Training accuracy 1, test accuracy 1
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 1.0}), datasets.TennisData)
Training accuracy 0.857143, test accuracy 0.833333
```

```
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 2.0}), datasets.TennisData)
Training accuracy 0.642857, test accuracy 0.5

>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 1}), datasets.TennisData)
Training accuracy 1, test accuracy 1
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 3}), datasets.TennisData)
Training accuracy 0.785714, test accuracy 0.833333
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 5}), datasets.TennisData)
Training accuracy 0.857143, test accuracy 0.833333
```

You can also try it on a different task which consists of classifying digits. Given an image of a hand-drawn digit (28x28 pixels, greyscale), your task it decide whether it's a ONE or a TWO.

```
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 6.0}), datasets.DigitData)
Training accuracy 0.96, test accuracy 0.64
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 8.0}), datasets.DigitData)
Training accuracy 0.88, test accuracy 0.81
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': False, 'eps': 10.0}), datasets.DigitData)
Training accuracy 0.74, test accuracy 0.74

>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 1}), datasets.DigitData)
Training accuracy 1, test accuracy 0.94
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 3}), datasets.DigitData)
Training accuracy 0.94, test accuracy 0.93
>>> runClassifier.trainTestSet(knn.KNN({'isKNN': True, 'K': 5}), datasets.DigitData)
Training accuracy 0.92, test accuracy 0.92
```

WU4: For the digits data, generate train/test curves for varying values of K and epsilon (you figure out what are good ranges, this time). Include those curves: do you see evidence of overfitting and underfitting? Next, using K=5, generate learning curves for this data.

For the remaining part of this section, our goal is to look at whether what we found for uniformly random data points (in HW03) holds for naturally occurring data (like the digits data) too! We must hope that it doesn't, otherwise KNN has no hope of working! but let's verify...

The problem is: the digits data is 784 dimensional, period, so it's not obvious how to try "different dimensionalities." For now, we will do the simplest thing possible: if we want to have 128 dimensions, we will just select 128 features randomly. You can re-use and modify code from the HighD.py script from HW03 for this purpose.

WU5: A. First, get a histogram of the raw digits data in 784 dimensions. You'll probably want to use the `computeDistances` function together with the plotting in HighD.
B. Rewrite `computeDistances` so that it can subsample features down to some fixed dimensionality. For example, you might write `computeDistancesSubdims(data, d)`, where `d` is the

target dimensionality. In this function, you should pick `d` dimensions at random (I would suggest generating a permutation of the number [1..784] and then taking the first d of them), and then compute the distance but *only* looking at those dimensions. **C.** Generate an equivalent plot to HighD with d in [2, 8, 32, 128, 512] but for the digits data rather than the random data. Include a copy of both plots and describe the differences.

Perceptron (30%)

This final section is all about using perceptrons to make predictions. I've given you a partial perceptron implementation in `perceptron.py`.

The last implementation you have is for the perceptron; see `perceptron.py` where you will have to implement part of the `nextExample` function to make a perceptron-style update.

Once you've implemented this, the magic in the `Binary` class will handle training on datasets for you, as long as you specify the number of epochs (passes over the training data) to run:

```
>>> runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 1}), datasets.TennisData)
Training accuracy 0.642857, test accuracy 0.666667
>>> runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 2}), datasets.TennisData)
Training accuracy 0.857143, test accuracy 1
```

You can view its predictions on the two dimensional data sets:

```
>>> runClassifier.plotData(datasets.TwoDDiagonal.X, datasets.TwoDDiagonal.Y)
>>> h = perceptron.Perceptron({'numEpoch': 200})
>>> h.train(datasets.TwoDDiagonal.X, datasets.TwoDDiagonal.Y)
>>> h
w=array([  7.3,   18.9]), b=0.0
>>> runClassifier.plotClassifier(array([ 7.3, 18.9]), 0.0)
```

You should see a linear separator that does a pretty good (but not perfect!) job classifying this data. Note that you should not close the popup window from the plotData call, since this line will be drawn on that plot. You might need to resize the window to see the line.

Finally, we can try it on the sentiment data:

```
>>> runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 1}), datasets.SentimentDat
a)
Training accuracy 0.835833, test accuracy 0.755
>>> runClassifier.trainTestSet(perceptron.Perceptron({'numEpoch': 2}), datasets.SentimentDat
a)
Training accuracy 0.955, test accuracy 0.7975
```

WU6: Using the tools provided, generate (a) a learning curve (x-axis=number of training examples) for the perceptron (5 epochs) on the sentiment data and (b) a plot of number of epochs versus train/test accuracy on the entire dataset.