README:

To follow along in the analytical report, there is a procedure called **GrandPoohBah.** That will be the only thing necessary to run, all the other functions are utility functions implemented in the GPB procedure. The functions are as follows

cleanData() makes data numeric

splitData() creates test and training set of initial data

PerceptronTrain() creates a perceptron

perceptronTest() compute activation for a row given to perceptron

LeastErrorPerceptron() create many perceptrons, cross validate and return the one with the least error

PredictedOutput() classification for each row

findError() error of model in test set

findBestKNN() finds best k for KNN based on performance in dev set

Goal: To create 3 models: perceptron, knn, and decision tree to classify and compare them on the currency training set. Optimize the perceptron algorithm, select best hyperparameters for knn, and see if decision tree can help with feature engineering. So that finally we will have a dataset of best features and a model to go with it.

Feature Engineering.

The currency training set consisted of different dimensions of a dollar bill and whether or not it was fake. Using lasso and ridge regression it was found that many of the dimensions such as length and width were highly correlated with other dimensions so they were removed. Thus the data set used for training consisted of only 3 features. **Top and Bottom Margin** and **Diagonal length**. These had enough variance between them to warrant their continued use as features. Feature engineering was also informally done during the creation of the classification tree. I created 2 other sets I ran each test for comparisons. One is **SINGLECOLUMNSET** which only uses diagonal length as a feature and then **NORMALSET** which normalizes the features. My rationale behind normalization was that with the weights may not have enough time to properly update so I also wanted an opportunity to make the assumption all features were created equal.

I also split the data **.8/.2** training/test respectively. While randomizing the training set before use.

Perceptron and Optimization Process

For the perceptron I tried several different methods on the respective sets. **SINGLECOLUMNSET, NORMALSET,** and the normal **TRAINING** set.

Firstly, I tried to just create a single perceptron. Then through Cross Validation, I attempted to use **AVERAGE** perceptron and **LEASTERROR** perceptron. Across all datasets Average perceptron had an error consistently close enough to .5 that it was considered an ineffective method and dropped from further consideration for perceptron selection. Using the least error perceptron created through cross validation varied wildly. The range of error for that perceptron was **[.267,.73]** , but when fit to a histogram in **LeastErrorPerceptronTestScript** it shows that most of the errors are centered around .43 and .57. Meaning that this is also not a very good classifier.

In another attempt at making a good perceptron classifier for the currency training set, I noticed that a **positive bias** or a bias toward real money almost always resulted in a classification error of above .5. I was unable to fully determine why, but because of it I restricted the perceptron to have a bias < 0. Thus assuring at least some negative bias, again trying to minimize error.

Overall there was no real way to manipulate the perceptron or the dataset before training to produce a perceptron that could classify well on a TEST set.

The largest issue with the perceptron was in its **PredictedOutputs**. There was very likely to classify the entire test set in one class or another, as a legitimate way of reducing the error in many cases. This was not solved through cross validation through any number of splits, thus this could be the result of a single perceptron's (as written by me) inability to accurately find a vector of weights to properly classify with a low number of inputs.

In my final attempt to create a perceptron I tried feeding it a larger training set, by taking what we had and copying the rows 10x before mixing them up thus giving the perceptron 10x more training material. In my specific example it did not change much, they weights were overall much larger, but classified data similarly.

KNN Optimization

Using the same analytical methods on the individual datasets as done on the perceptron, The Knn classifier consistently scored a nearly perfect score. By running the **FindBestKNN** for several k's and distributing the success rate of each k. K=1 turned out to be the best, with k=2 closely following. After k=2 the error was 0 thus any more neighbors are redundant in voting. Fitting the different datasets to the selected KNN method all resulted in perfect scores. So KNN classification was extremely successful in this dataset.

Decision Tree

The decision tree when trained on the normal TRAINING SET with no feature engineering as opposed to testing it onto the other sets as well. This was a conscious decision because of the types of decision trees made by MATLAB. They were of the following form, with left being yes and right being no:

```
                              Diagonal > 140.5

             Top margin > 9.6                         Class = -1

Class = -1                  class = 1
```

This tree doesn't even use Any of the features besides Top Margin and Diagonal. It even excluded bottom margin, something I was not comfortable doing from the matrices provided by lasso regression. Due to the nature of the tree there was no need to implement purity measures or stop criteria because they tree's dept was very small so any pruning or manipulation would likely decrease its classification accuracy

Conclusion

KNN was the best method for classification of this dataset. Though Decision trees and knn both had nearly 0 error (actual range from 0 to .07, but overwhelmingly 0), decision trees had hiccups that I didn't see in my analysis of KNN, due to decision trees using limited features in their classification.

We reduced the number of dimensions using the decision tree, lasso, and ridge regression, thus increasing efficiency without sacrificing accuracy in the KNN case. KNN wouldn't often have errors, but those that did show up were prior to any dimensionality reduction, thus not relevant in this case. The perceptron was not a strong contender against KNN and Decision trees. Possibly because they are much better on smaller datasets not offering the opportunity to find accurate weights for the perceptron.

Additions/Notes:

Error didn't seem like the best way of comparing models on development set.

What if given a larger dataset, do classification methods for trees or weights change?

How to get a perceptron to accurately classify this data besides feature engineering?