Project 01 Write Up

October 9, 2022

```python
import util
import datasets
import binary
import statistics as stats
import numpy as np

import dumbClassifiers
import runClassifier
import dt
import knn
import perceptron
```

WU1: why is this computation equivalent to computing classification accuracy?

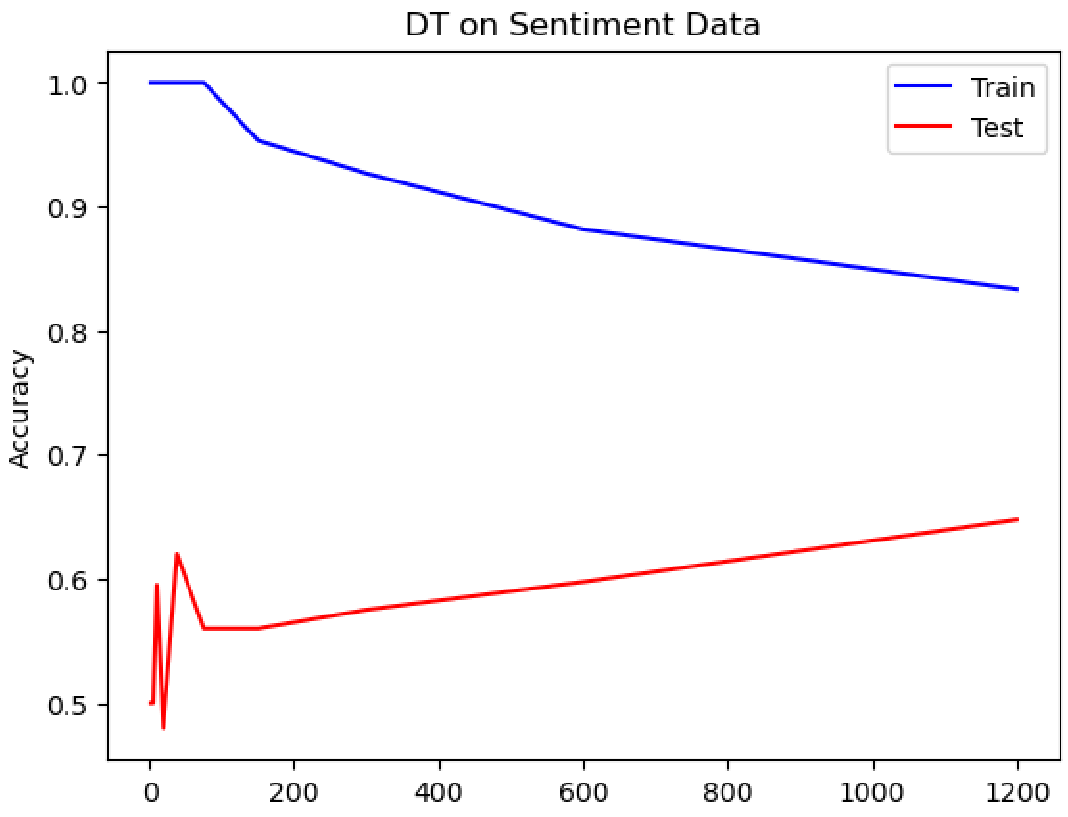
It's equivalent because predictAll will always return 1. "mean" treats a match as 1 and a mismatch as 0, so it can be used as classification accuracy.

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?

When there are few data points in the training set, a tree of depth 9 can perfectly fit the data. With more data points, there will be collisions(similar features map to different labels) or important complexity in the data that can't be accounted for with a tree of depth 9, which reduces training set accuracy.

There is a jagged region on the left of the test curve because it's possible for the tree to perfectly fit the training data as there are 2ˆ9(512) possible leaves, if there are no collisions and the training examples are separable from the data. The tree can model spurious correlations that exist in both the training set and test set and perform better, but once new data is introduced to the training set, those correlations are removed, which drops accuracy. Once enough data is present in the training set, the spurious correlations are all averaged out, so the test accuracy curve smooths out.

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

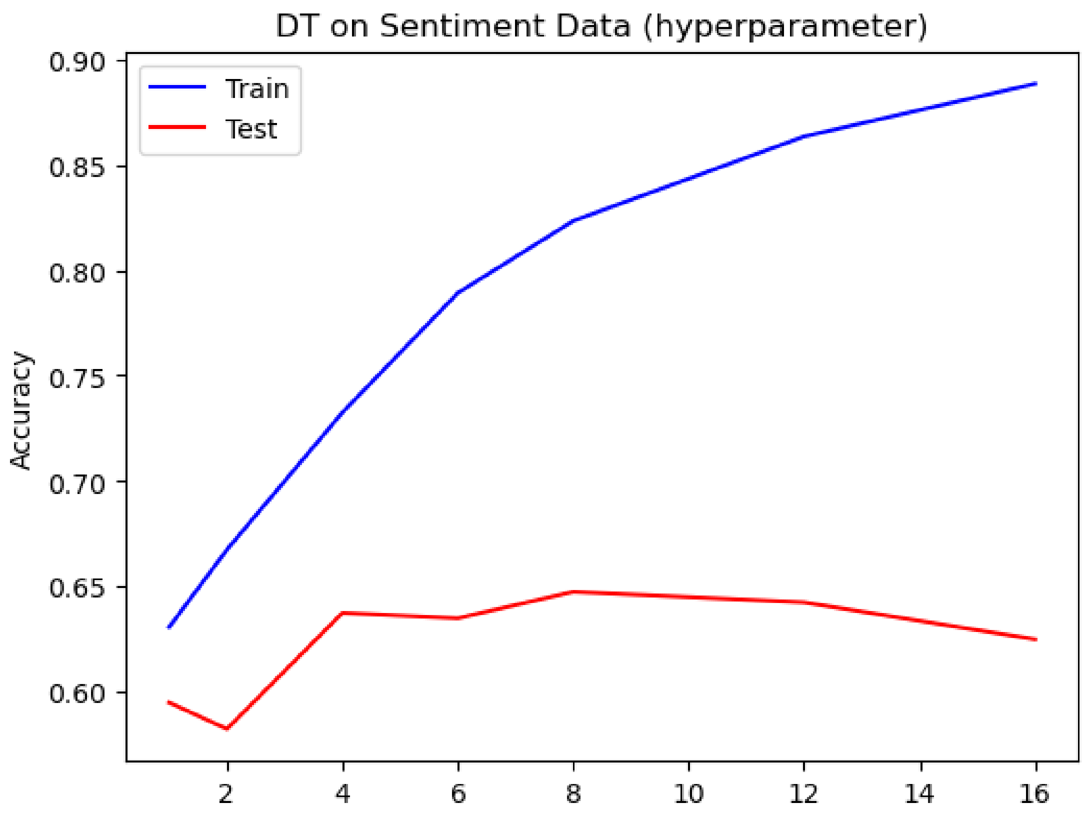


WU3: You should see training accuracy monotonically increasing and test accuracy making something like a hill. Which of these is guaranteed to happen and which is just something we might expect to happen? Why?

Training accuracy monotonically increasing is guaranteed. This is because decision trees will deterministically improve accuracy with depth because it separates the data into more nuanced groups the more depth the tree has.

The test accuracy hump is not guaranteed because if the test data is less similar to the training data, then we'll see a hump as the model fits the training data better than the test data. However, if the test data is more similar to the training data, it will take more depth before we see test accuracy start to decrease. Eventually, you should see overfitting of the training data.

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



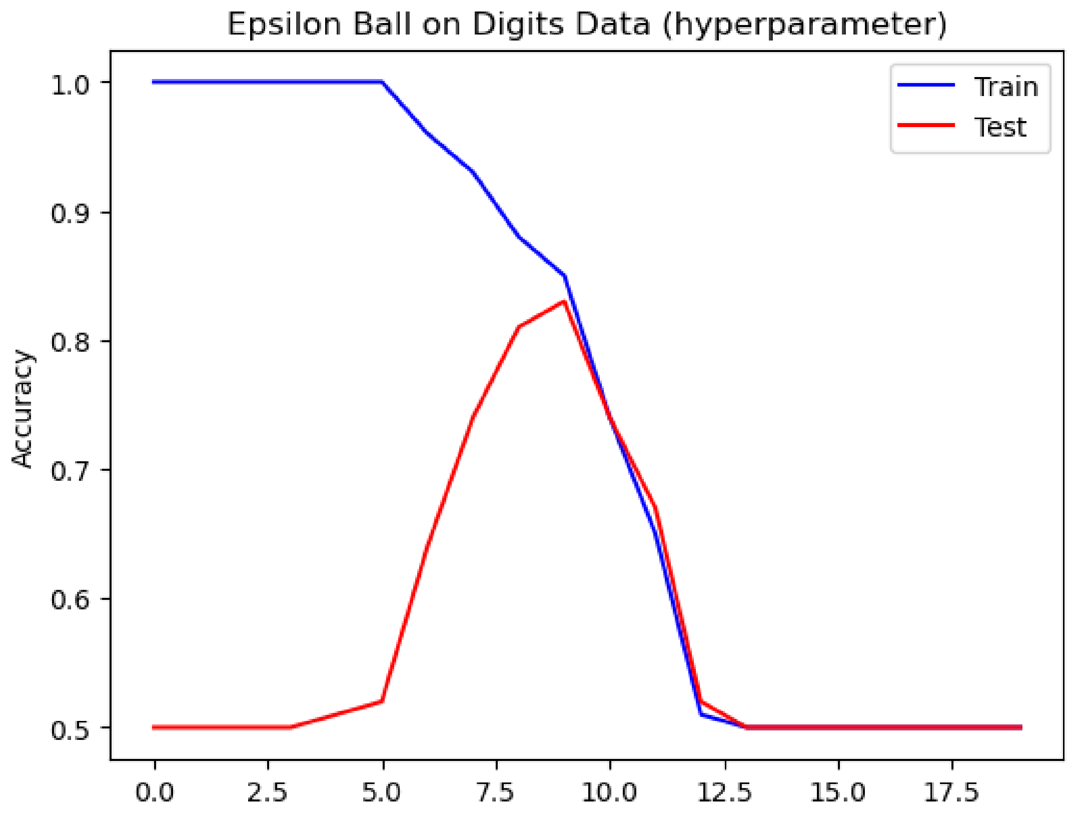
WU4: For the digits data, generate train/test curves for varying values of K and epsilon (you figure out what are good ranges, this time). Include those curves: do you see evidence of overfitting and underfitting? Next, using K=5, generate learning curves for this data.

For epislon-ball, we initially see overfitting of the training data for epsilon less than 9 and then underfitting as the model becomes too simple to describe the data for epsilon greater than 10, until it drops to a random guess baseline of 50%. The baseline for this model is 50/50 because we're trying to distinguish a written 1 vs a written 2.

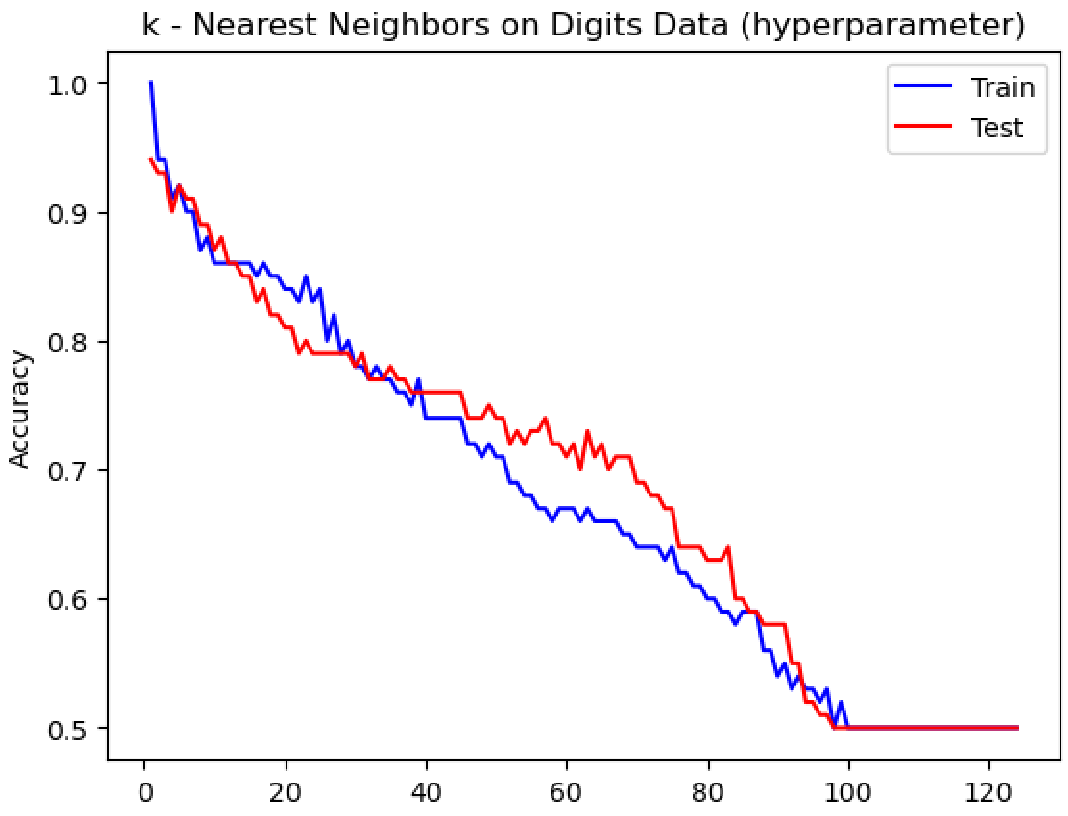
For the KNN, there appears to be overfitting before K = 5. Afterwards, there appears to be an oscillation between underfitting and overfitting after K = 5. At round K=100, we hit the baseline random guessing rate of 50%.

The K=5 kNN appears to oscillate between overfiting and underfitting, but converges around 100 training points with more accuracy potential with more data points.

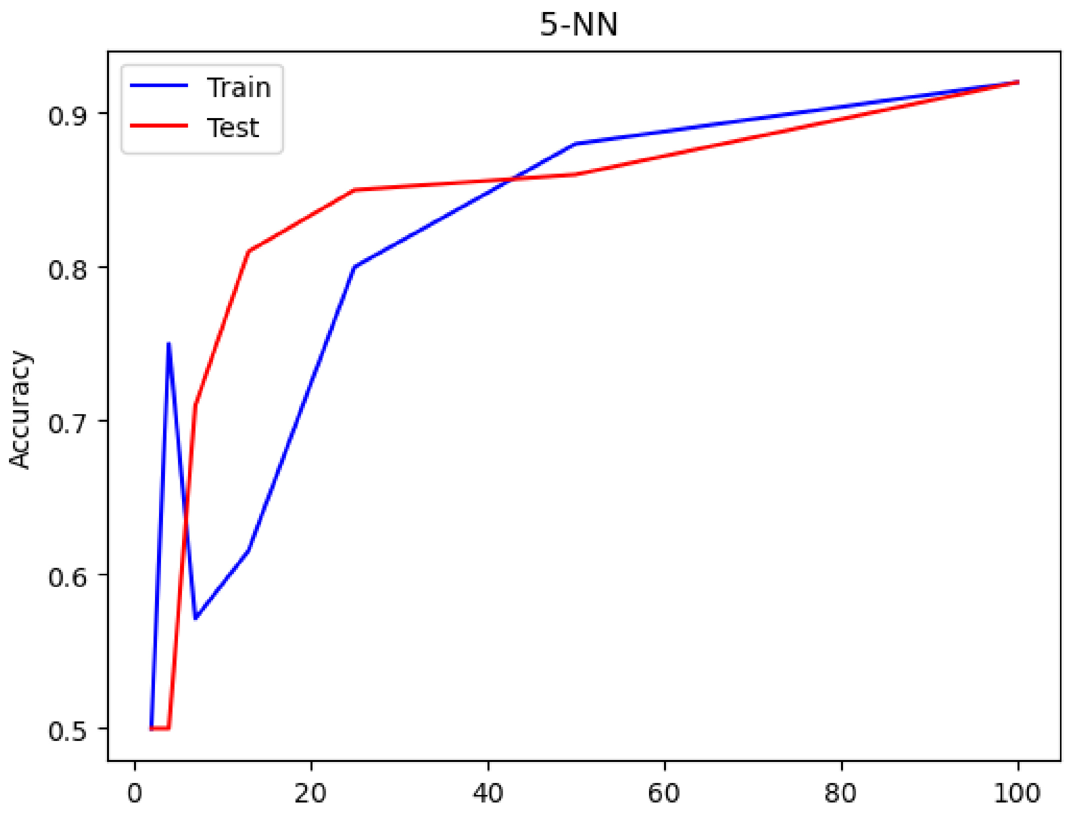
```
curveEpsilon = runClassifier.hyperparamCurveSet(knn.KNN({'isKNN': False}),
                                                'eps', range(0,20),
                                                datasets.DigitData)
runClassifier.plotCurve('Epsilon Ball on Digits Data (hyperparameter)',␣
 ↪curveEpsilon)
```



```
curveKNN = runClassifier.hyperparamCurveSet(knn.KNN({'isKNN': True}),
                                            'K', range(1,125),
                                            datasets.DigitData)
runClassifier.plotCurve('k - Nearest Neighbors on Digits Data (hyperparameter)',␣
 ↪curveKNN)
```

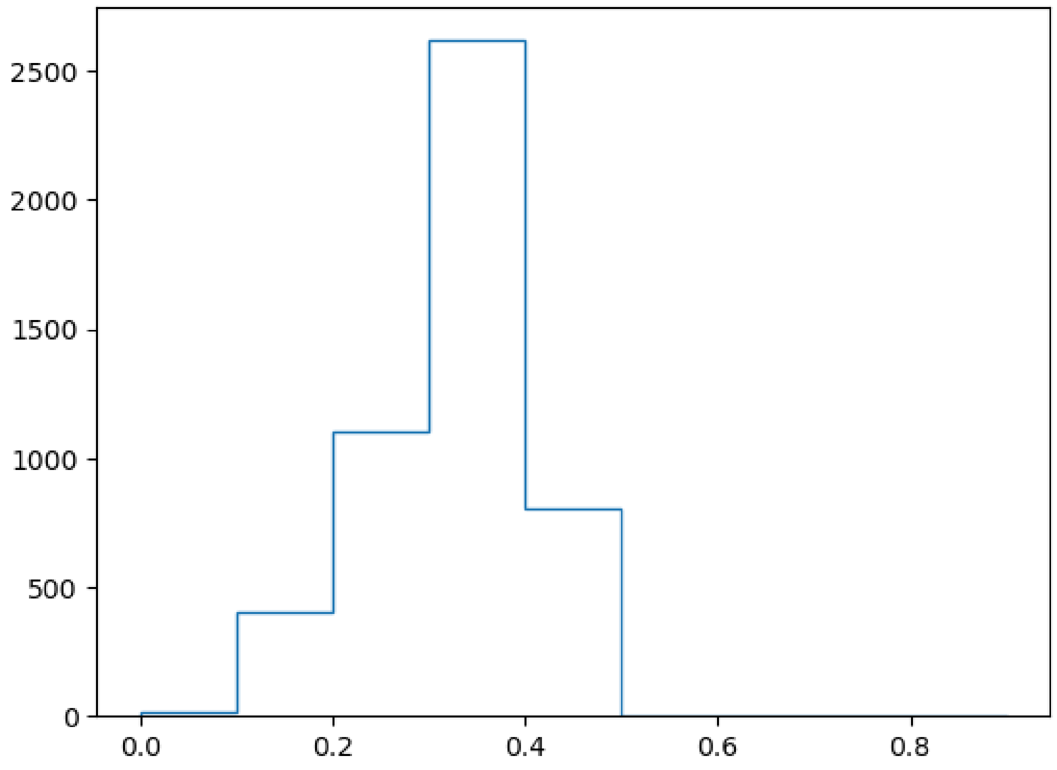


```
curve = runClassifier.learningCurveSet(knn.KNN({'isKNN': True, "K": 5}),
                                        datasets.DigitData)
runClassifier.plotCurve('5-NN', curve)
```



WU5A: 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.

```
from matplotlib import pyplot as plt

pdata = hd.computeDistances(datasets.DigitData.X)
plt.hist(pdata, [x/10 for x in range(10)], histtype='step')
plt.show()
```



WU5B: 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.

```
import HighD as hd
import random as leo_random

def computeDistancesSubdims(data, d):
    N = len(data)
    D = len(data[0])
    dist = []

    randomlist = leo_random.sample(range(D), d)

    for n in range(N):
        ndata = [datasets.DigitData.X[n][i] for i in randomlist]
        for m in range(n):
            mdata = [datasets.DigitData.X[m][i] for i in randomlist]
            dist.append(hd.computeExampleDistance(ndata , mdata)  / np.sqrt(d))
    return dist
```

WU5C: 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.

As dimensionality increases, the mean for both move toward 0.4.

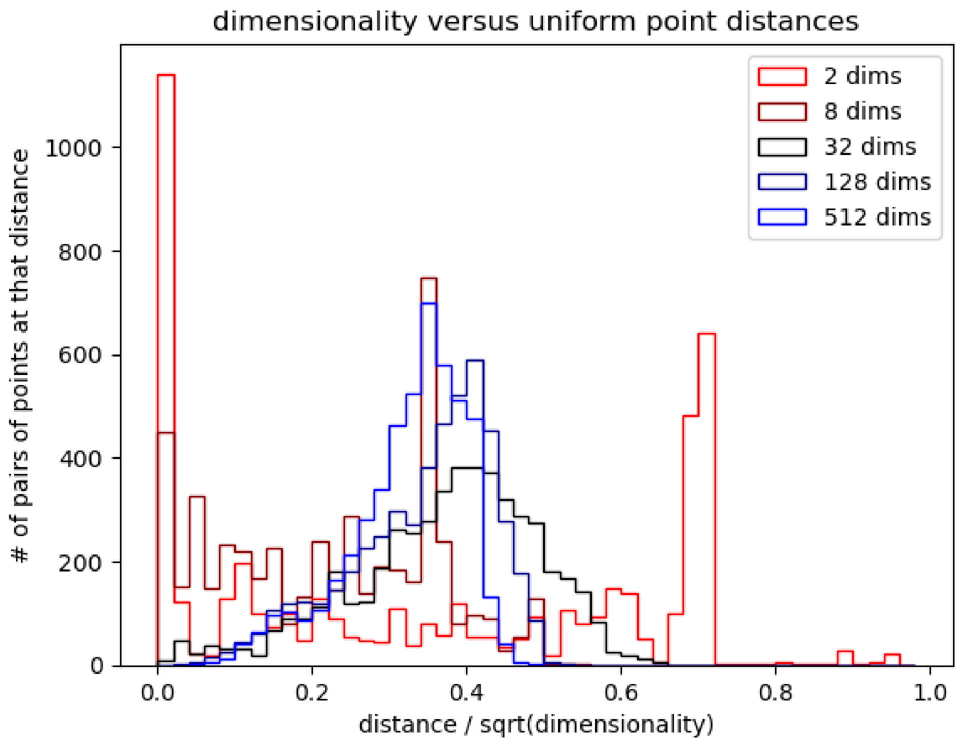
However, the differences are that the digits data plot has a much more pronounced skew to the left for dimensions over 8, but the random data also has a skew to the right for dimensionality of 2 but it is much more subtle than the digits data.

The digits data also seems to have bi-modality for 2 and 8 dimensions, which is very different. However this might be due to fact that we have two classifications and the two modalities represent the the distance between each digit with itself (zero), and each digit to the other digit (zero to one).

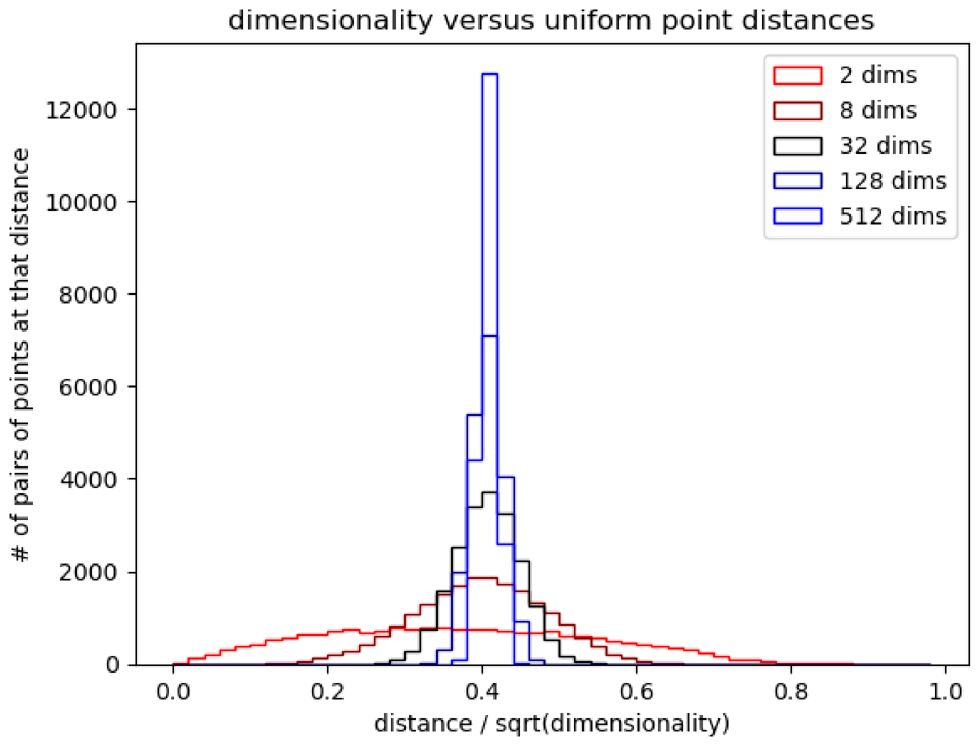
```
import HighD as hd
```

```
N    = 200                          # number of examples
Dims = [2, 8, 32, 128, 512]    # dimensionalities to try
Cols = ['#FF0000', '#880000', '#000000', '#000088', '#0000FF']
Bins = np.arange(0, 1, 0.02)

plt.xlabel('distance / sqrt(dimensionality)')
plt.ylabel('# of pairs of points at that distance')
plt.title('dimensionality versus uniform point distances')

for i,d in enumerate(Dims):
    distances = computeDistancesSubdims(datasets.DigitData.X ,d)
    print ("D=%d, average distance=%g" % (d, np.mean(distances) * np.sqrt(d)))
    plt.hist(distances, Bins, histtype='step', color=Cols[i])

#    if waitForEnter:
#        plt.legend(['%d dims' % d for d in Dims])
#        plt.show()
#        x = raw_input('Press enter to continue...')

plt.legend(['%d dims' % d for d in Dims])
plt.savefig('fig.pdf')
plt.show()
```

```
Digit Data Plot
```

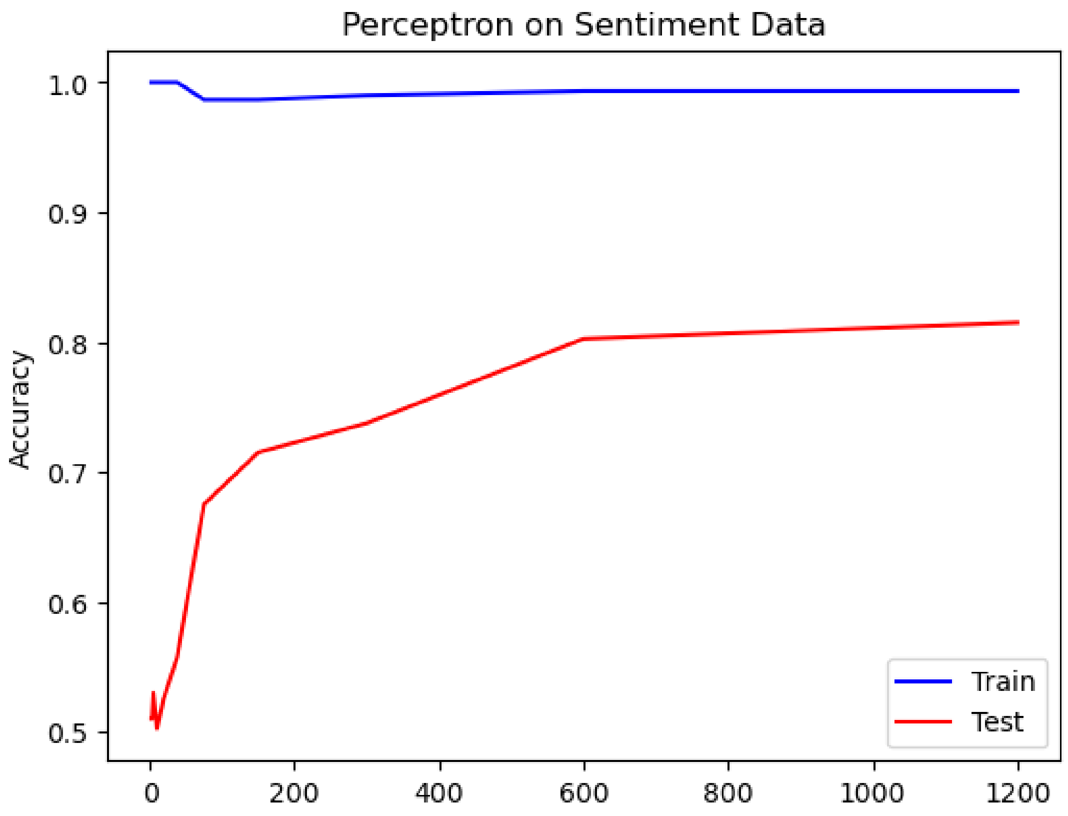


```
Random Data Plot
```



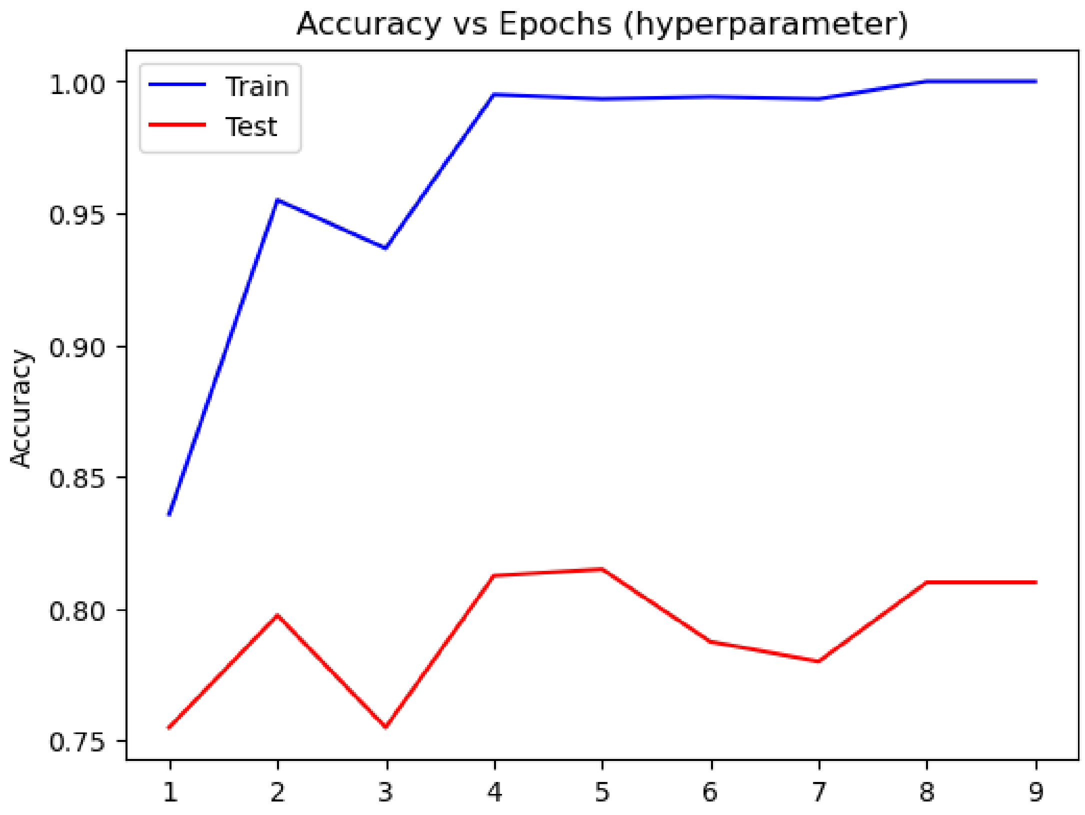
WU6: Using the tools provided, generate (a) a learning curve (x-axis=number of training examples) for the perceptron (5 epochs) on the sentiment data.

```
curve = runClassifier.learningCurveSet(perceptron.Perceptron({'numEpoch': 5}),
                                       datasets.SentimentData)
runClassifier.plotCurve('Perceptron on Sentiment Data', curve)
```



WU6: Using the tools provided, generate (b) a plot of number of epochs versus train/test accuracy on the entire dataset.

```
curve = runClassifier.hyperparamCurveSet(perceptron.Perceptron({}),
                                          'numEpoch', range(1,10),
                                          datasets.SentimentData)
runClassifier.plotCurve('Accuracy vs Epochs (hyperparameter)', curve)
```


Accuracy vs Epochs (hyperparameter)
Train
Test
Accuracy