Brennan Hall
861198641
CS171 - 18S
Late Days Used: 0
Total Late Days: 2

Assignment 2

Question 1: k-Nearest Neighbor Classifier *[50%]*

The second classifier we will see in this assignment is the k-Nearest Neighbor (or k-NN) classifier. k-NN is a lazy classification algorithm, which means that it skips the training step and makes a decision during testing time. For the distance calculations, you will use the Lp norm function you implemented in the previous assignment. You should implement the following function:

y_pred = knn_classifier(x_test, x_train, y_train, k, p)

Where the inputs are:

1. x_test = a 'test data point' x 'feature' matrix containing all the test data points that need to be classified
2. x_train = a 'training data point' x 'feature' matrix containing all the training data points that you will compare each test data point against.
3. y_train = a 'train data point' x 1 vector containing the labels for each training data point in x_train (there is obviously a 1-1 correspondence of rows between x_train and y_train).
4. k = the number of nearest neighbors you select
5. p = the parameter of Lp distance as we saw it in Assignment 1.

The output of the function is:

1. y_pred = a 'test data point' x 1 vector containing the predicted labels for each test data point in x_test (again, there is correspondence between the rows).

To properly implement this function, I needed to understand the central point of the k-NN classifier as well as the data set given to us. In a prior class, we had implemented an optimized recursive form of the nearest neighbor algorithm, and I had considered optimizing this algorithm in the same way, but I realized that it only worked for 2D data points.

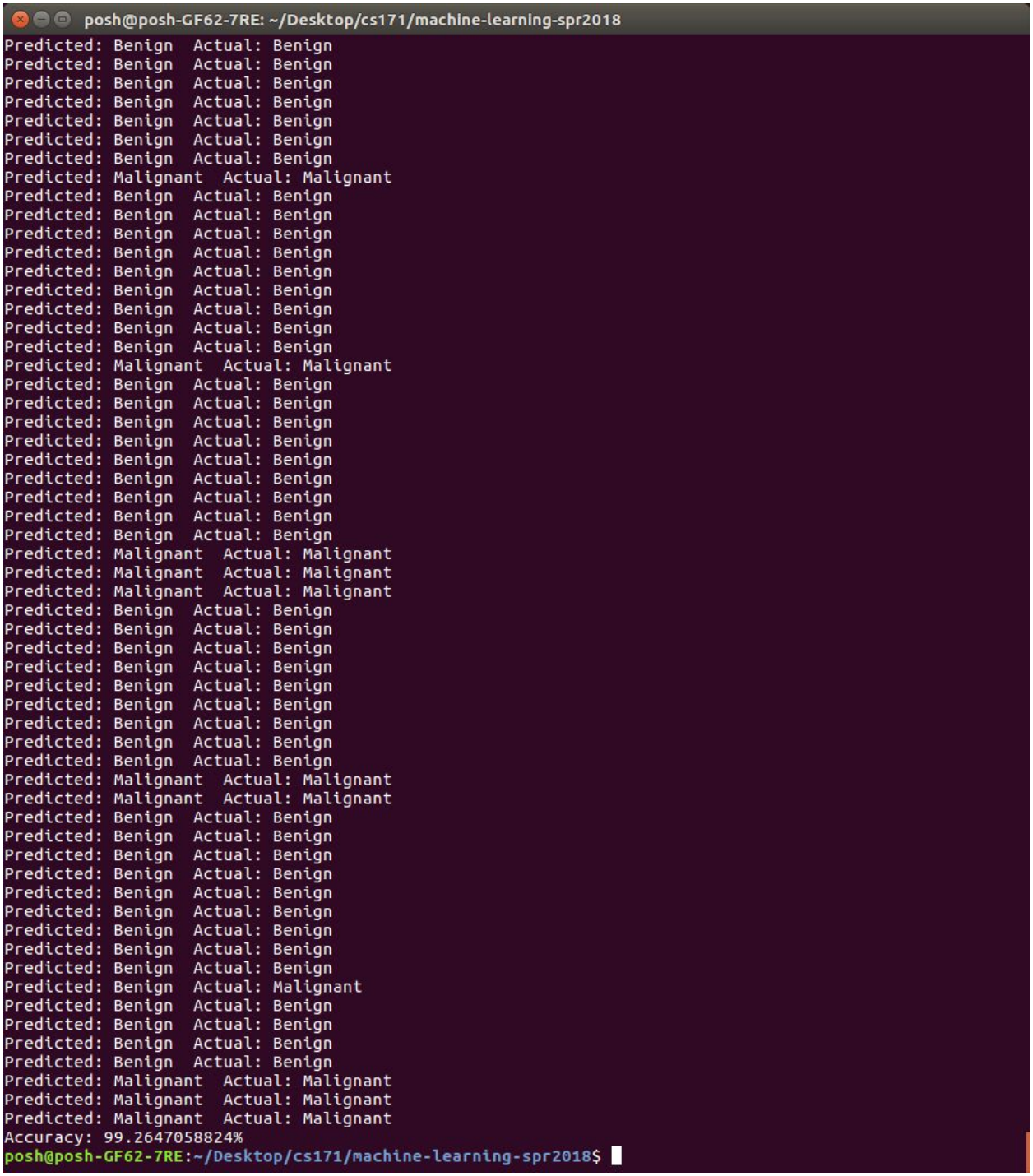
With this being said, my algorithm is a brute force approach that calculates the distances between the 20% test set to the 80% training set. In this sense, there is little extra cost to using higher values of k, so if higher values of k prove to make a better performing classifier, brute force handles takes care of this just fine.

When all the distances are calculated and put into a list, the list is sorted in ascending order and the top k values are selected to help classify the current test point. If more neighbors are benign than malignant, the test point will be labeled as benign, and vice versa. If the number of benign neighbors is equal to the number of malignant neighbors, the algorithm aires on the side of caution and classifies it as malignant.

The following classifications are performed on the unshuffled dataset, using the top 80% as training data and the bottom 20% as test data:

```
posh@posh-GF62-7RE: ~/Desktop/cs171/machine-learning-spr2018
posh@posh-GF62-7RE:~/Desktop/cs171/machine-learning-spr2018$ python assn2.py
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
```

```
posh@posh-GF62-7RE: ~/Desktop/cs171/machine-learning-spr2018
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Benign  Actual: Benign
Predicted: Malignant  Actual: Malignant
Predicted: Malignant  Actual: Malignant
```



The output of the function is y_pred, which is simply a list of the points and their labels assigned by the classifier. Y_pred is what is used when calculating the accuracy, since it has data on both the predicted value as well as the actual value.

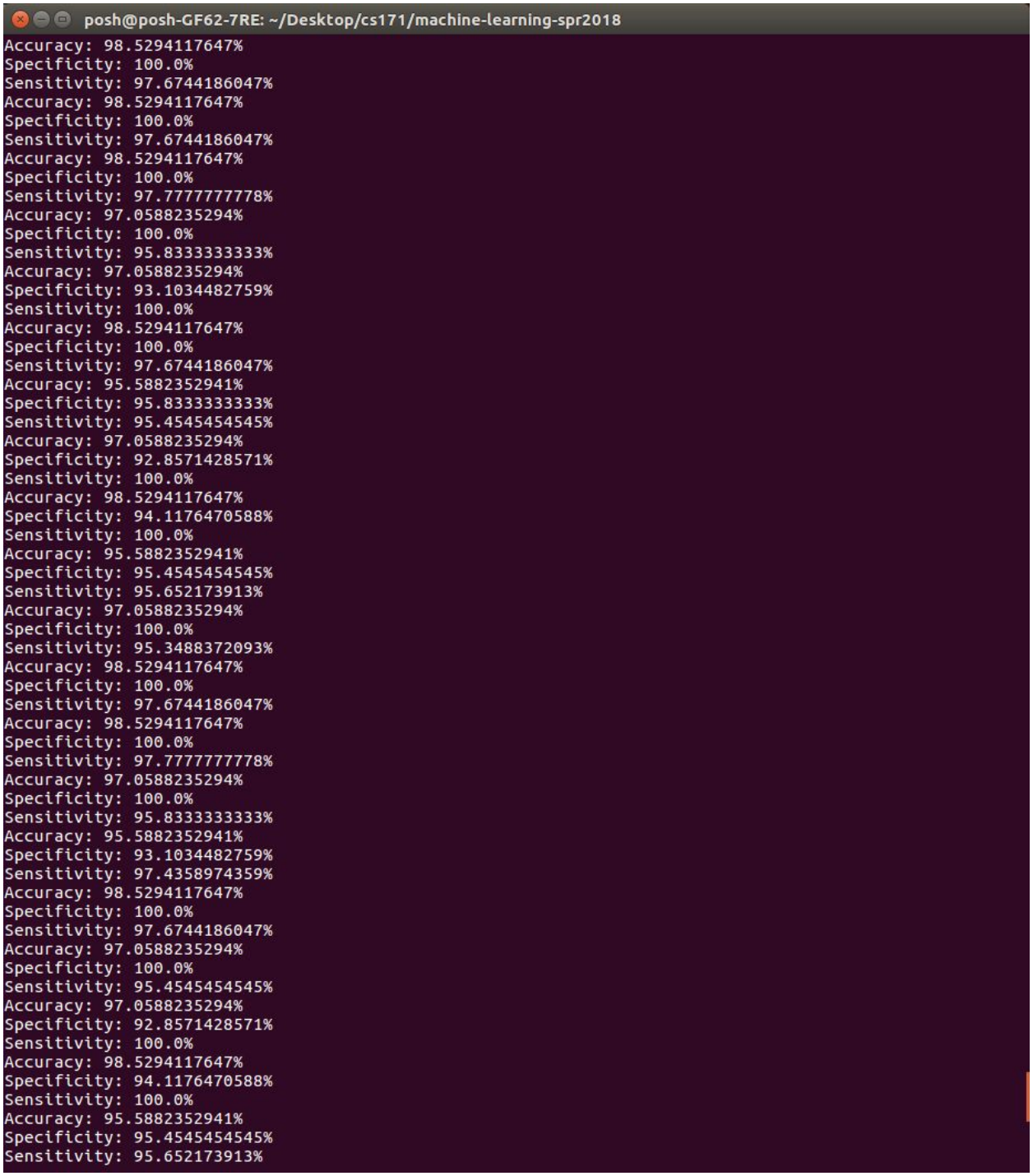
1. **Cross-validation**: Implement 10-fold cross-validation. As described in the lectures, shuffle the entire dataset randomly, and split it in 10 equal-sized portions. Then, select one of the portions and make it the "test" set, and the rest will be the "training set". Train and test your classifier using these two sets and record the following **performance** measures: 1) accuracy, 2) sensitivity, and 3) specificity; for all those metrics, compute the mean and standard deviation (you may use existing functions for mean and stdev). We are going to need the standard deviation so that we report error-bars around the mean value. Repeat this procedure by selecting the next fold as the test set (and the remaining 9 as training), until you iterate over all folds. You can make a single script/source file for the cross-validation, which will wrap around the appropriate calls to the different training/testing procedures with the appropriate inputs, as per the questions below.

When implementing cross-validation, my initial goal was to generalize the function so that any classifier could be used. The remnants of this preparation are still present in the code. I first shuffle the dataset, then partition it into 10 folds. Then the main for loop begins in which the classifier is called for each fold and training set. Each iteration of the classifier returns its own y_pred, on which the accuracy, specificity, and sensitivity are stored into arrays. Once all the calls are complete, these arrays of each performance value are reduced by taking the mean and the standard deviation.

```
# MAIN LOOP - Performs hundreds of k-NN iterations
for p in {1, 2}:
    for k in {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}:
        accuracyList = []
        specificityList = []
        sensitivityList = []
        for i in dataFolds:
            x_test = i
            x_train = []
            for j in dataFolds:
                if i != j:
                    x_train += j
            y_pred = []

            if classifierName == 'kNN':
                y_pred = knn_classifier(x_test, x_train, x_test, k, p)
```

This code snippet is the beginning part of the loop, which is more tailored towards the second part of this question. As stated, the current fold is used at the test set while the other folds are aggregated into one large training set (90/10 ratio).


posh@posh-GF62-7RE: ~/Desktop/cs171/machine-learning-spr2018
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.6744186047%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.6744186047%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.7777777778%
Accuracy: 97.0588235294%
Specificity: 100.0%
Sensitivity: 95.8333333333%
Accuracy: 97.0588235294%
Specificity: 93.1034482759%
Sensitivity: 100.0%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.6744186047%
Accuracy: 95.5882352941%
Specificity: 95.8333333333%
Sensitivity: 95.4545454545%
Accuracy: 97.0588235294%
Specificity: 92.8571428571%
Sensitivity: 100.0%
Accuracy: 98.5294117647%
Specificity: 94.1176470588%
Sensitivity: 100.0%
Accuracy: 95.5882352941%
Specificity: 95.4545454545%
Sensitivity: 95.652173913%
Accuracy: 97.0588235294%
Specificity: 100.0%
Sensitivity: 95.3488372093%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.6744186047%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.7777777778%
Accuracy: 97.0588235294%
Specificity: 100.0%
Sensitivity: 95.8333333333%
Accuracy: 95.5882352941%
Specificity: 93.1034482759%
Sensitivity: 97.4358974359%
Accuracy: 98.5294117647%
Specificity: 100.0%
Sensitivity: 97.6744186047%
Accuracy: 97.0588235294%
Specificity: 100.0%
Sensitivity: 95.4545454545%
Accuracy: 97.0588235294%
Specificity: 92.8571428571%
Sensitivity: 100.0%
Accuracy: 98.5294117647%
Specificity: 94.1176470588%
Sensitivity: 100.0%
Accuracy: 95.5882352941%
Specificity: 95.4545454545%
Sensitivity: 95.652173913%

2. **Evaluating k-NN**: We are going to evaluate the performance of k-NN for various parameter choices. In particular, within each fold of the cross-validation, record mean and standard deviation of the performance for k = 1:10, for p = 1 and p =2. Organize those results as plots where the y-axis shows the performance and the x-axis shows the number of nearest neighbors (k). You may combine the lines for p = 1 and p = 2 in the same plot and add a legend that shows which lines refers to which parameter. As above, report error-bars. In total, this will be 3 plots (if you combine p = 1 and p = 2 in the same plot) or 2x3 plots if you don't. Which k and which p yield the best performance?

For this second segment of the question, I decided to overhaul my cross-validation function to also plot the graph of the necessary values. In essence, cross-validation is run on each k 1:10 and each p 1,2 and all the performance values are evaluated and placed into the proper order for their array. Many iterations of k-NN are run, so the program takes a minute or so to run completely. Once all of the values have been evaluated, the cross-validation takes the computed arrays and draws each graph using the means of the performance values as the plotted data, and the standard deviations as the error bars. The graphs are shown here:

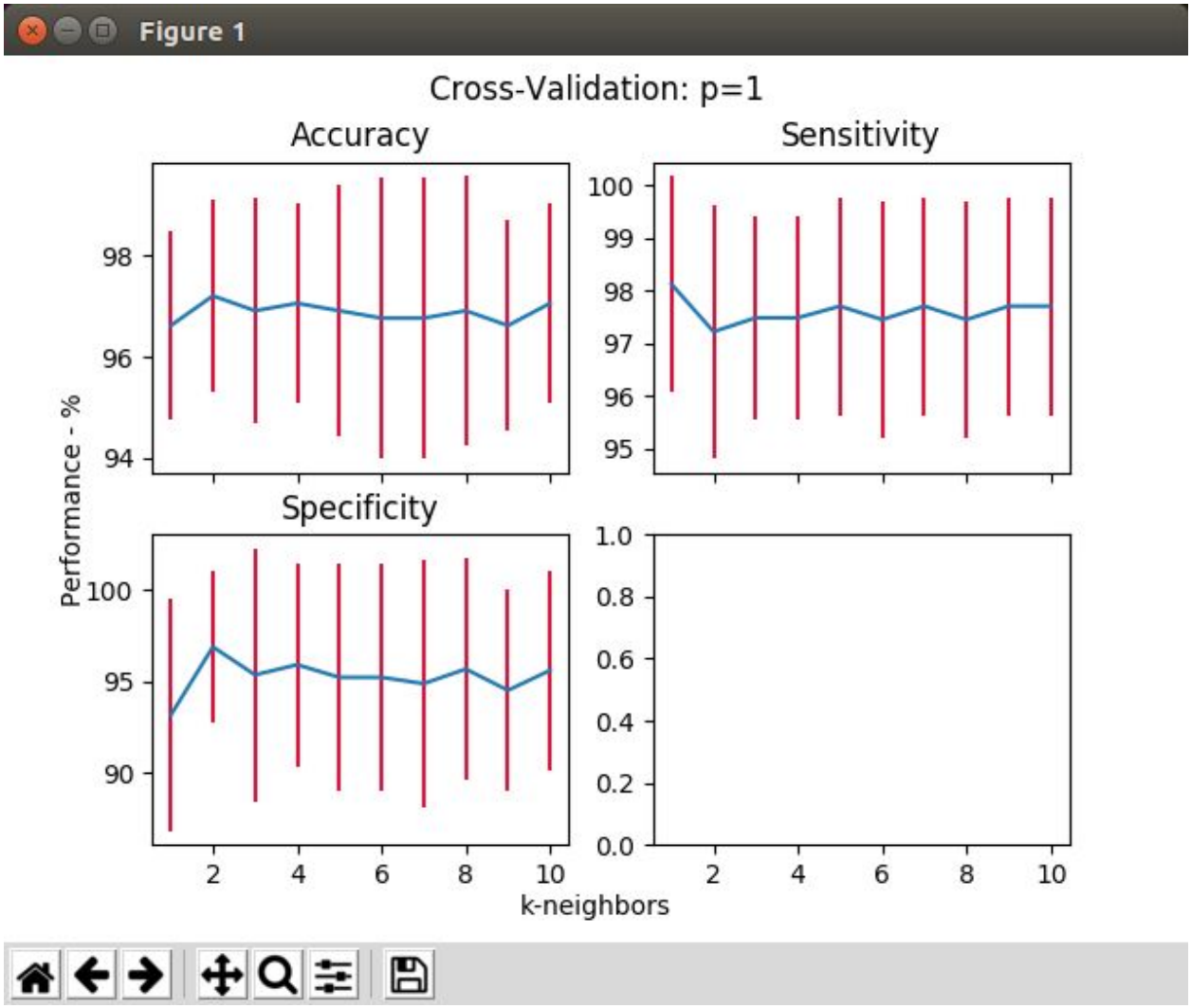
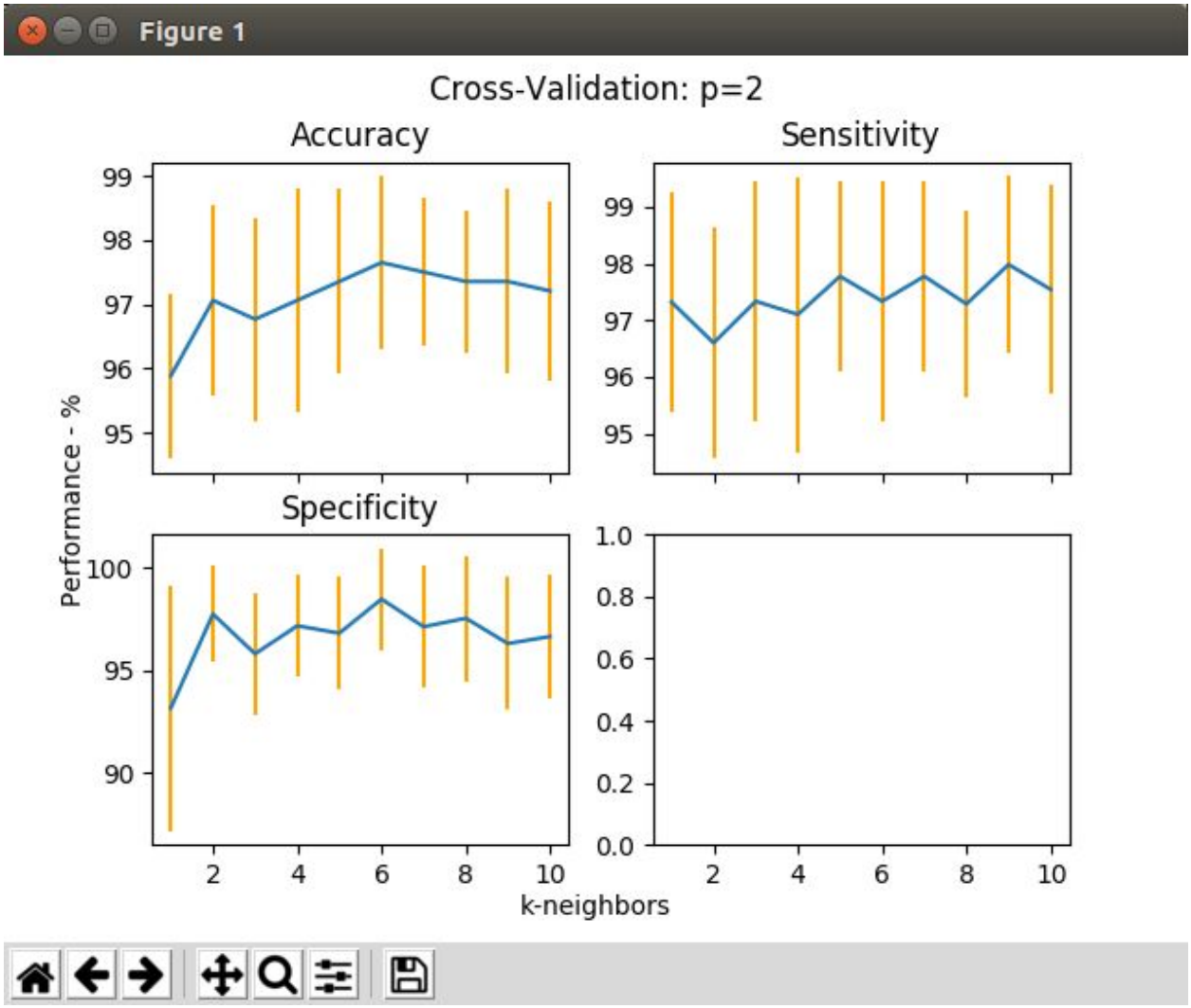


Figure 1
Cross-Validation: p=1
Accuracy
Sensitivity
Specificity
Performance - %
k-neighbors
98
96
94
100
99
98
97
96
95
100
95
90
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10



As shown above, there is a large amount of deviation for the majority of the data. Most of the values sit comfortably around the 97% range, but still have high amount of deviation. This is due to the different nature of each cross-validation iteration, since each fold can have harder to predict labels than others. It can be said that the graphs for p=2 are similar, but have higher variance, which might be more ideal in certain testing cases. On both sets of graphs, accuracy is highest on average around the k=6 mark, but most values for k that are above 1 give good enough values to begin with.

Sensitivity is observed to be slightly higher if k is an odd number. This is most likely due to the definitive nature of labeling data points when there is never an equal number of neighbors from different classes. If k is an odd number, there is always a definitive answer to which of each points neighbors there are more of, which means it is easier to garner true positive values. Specificity is the lowest for k=1, which makes sense because only one neighbor can make it hard to find negatives.

All in all, there are a variety of ways that these graphs could turn out, given that the data is shuffled. There is no objectively best pair of k,p given that there are many different outcomes, but in this scenario, the overall highest accuracy test case would be where p=2 and k=6. In this scenario, accuracy and sensitivity are on average around 98%, although sensitivity is slightly lower than average.