Alena McCain
U00833503
brand.18@wright.edu

Implementation

The first section of the code reads the training data and test data for the classifiers. ReadInputData reads the feature data for each document from an input file. The output matrix is transposed so the documents are the rows and the features of columns because that is the format that Sci-kit Learn classifiers expect. In ReadClassData, reads the class label for each document. Sci-kit Learn classifiers only expect a single array of labels (where each index corresponds to a document), so that is why only the second column of the class input files is preserved.

```
# Read in the feature data
def ReadInputData(fileName):
    # Transpose so the rows are the documents and the features are the columns
    return np.loadtxt(fileName).transpose()


# Read in the class data
def ReadClassData(fileName):
    # Only get the column with the classifications
    return np.loadtxt(fileName)[:, 1]
```

The next section of the code calculates the K nearest neighbors for each document in the training set. It uses cosine similarity to calculate document similarity, as it is usually the most robust similarity metric. A calculates the cosine similarity of each of the 200 test documents with all 800 training documents. Then for each test document, it determines the classes of the k closest documents and returns them in the matrix of size 200xk.

```
# Gets top k Cosine Similarity values (nearest neighbor training document
classes) for each test document
def GetNearestNeighborClasses(xTrain, xTest, yTrain, k):
    # Calculate Euclidian distance
    cosineSimilarityMatrix = pairwise.cosine_similarity(X=xTest, Y=xTrain)
    maxValues = []
    # For each test document
    for testDocument in cosineSimilarityMatrix:
        # Get the training documents (indicies) of the k nearest neighbors
        indicies = np.argsort(testDocument)[-k:]
```

Alena McCain
U00833503
brand.18@wright.edu

```
        # Find the corresponding classes
        classes = [yTrain[i] for i in indicies]
        maxValues.append(classes)
    return np.array(maxValues)
```

After that, RunKNearestNeighbors takes the 200xk matrix and determines for each document what the most frequent class is out of the k class labels. It then makes a label prediction array that stores the most frequent label in the test document index. Hence the predicted label array is a size of 200. The actual labels versus the predicted labels of the test documents are then compared by creating a confusion matrix, calculating accuracy, precision, recall, and F1 score.

```
# Does all calculations for K Nearest Neighbors classifier
def RunKNearestNeighbors(xTrain, yTrain, xTest, yTest, k):
    yPredicted = []
    # Find the k nearest neighbor training document classes of every test
document
    knn = GetNearestNeighborClasses(xTrain, xTest, yTrain, k)
    # For each list of test document nearest neighbors
    for testDocumentNN in knn:
        # Find the most frequent class of the nearest neighbors and make that the
prediction
        values, counts = np.unique(testDocumentNN, return_counts=True)
        yPredicted.append(values[counts.argmax()])
    # Calculate confusion matrix and accuracy
    cm = confusion_matrix(yTest, yPredicted)
    accuracy = (cm[0][0] + cm[1][1]) / 200
    p = cm[1][1] / (cm[1][1] + cm[1][0])
    r = cm[1][1] / (cm[1][1] + cm[0][1])
    f1 = 2 * (p * r) / (p + r)
    return accuracy, cm, p, r, f1
```

The next two sections of code create and run the naïve Bayes in the linear SVM classifiers, respectively. In both cases, the classifier is created, fit to the training data, and then makes predictions

Alena McCain
U00833503
brand.18@wright.edu

using the test documents. The prediction method for the classifiers returns an array of size 200, much like the hand-implemented K nearest neighbors classifier. The actual labels versus the predicted labels of the test documents are then compared by creating a confusion matrix, calculating accuracy, precision, recall, and F1 score.

```
# Does all calculations for Multinomial Naive Bayes classifier
def RunSkLearnMultinomialNB(xTrain, yTrain, xTest, yTest):
    # Create classifier
    nb = MultinomialNB()
    # Train classifier
    nb.fit(xTrain,yTrain)
    # Test classifier
    yPredicted = nb.predict(xTest)
    # Calculate confusion matrix and accuracy
    cm = confusion_matrix(yTest, yPredicted)
    accuracy = (cm[0][0] + cm[1][1]) / 200
    p = cm[1][1] / (cm[1][1] + cm[1][0])
    r = cm[1][1] / (cm[1][1] + cm[0][1])
    f1 = 2 * (p * r) / (p + r)
    return accuracy, cm, p, r, f1


# Does all calculations for SVM classifier
def RunSkLearnSVM(xTrain, yTrain, xTest, yTest):
    # Create classifier
    svm = LinearSVC()
    # Train classifier
    svm.fit(xTrain,yTrain)
    # Test classifier
    yPredicted = svm.predict(xTest)
    # Calculate confusion matrix and accuracy
```

Alena McCain
U00833503
brand.18@wright.edu

```
    cm = confusion_matrix(yTest, yPredicted)
    accuracy = (cm[0][0] + cm[1][1]) / 200
    p = cm[1][1] / (cm[1][1] + cm[1][0])
    r = cm[1][1] / (cm[1][1] + cm[0][1])
    f1 = 2 * (p * r) / (p + r)
    return accuracy, cm, p, r, f1
```

The next section of code writes a confusion matrix, accuracy, precision, recall, and F1 score to the file specified.

```
# Write confusion matrix, accuracy, precision, recall, and f1 data to file
def WriteStatsToFile(fileName, accuacy, confusionMatrix, p, r, f1):
    fileObject = open(fileName, "w")
    fileObject.write("Accuracy:" + str(accuacy) + "\n")
    fileObject.write("Precision:" + str(p) + "\n")
    fileObject.write("Recall:" + str(r) + "\n")
    fileObject.write("F1:" + str(f1) + "\n")
    fileObject.write("Confusion Matrix:\n")
    for row in confusionMatrix:
        fileObject.write(" ".join([str(a) for a in row]) + "\n")
    fileObject.close()
```

The next section of code creates a bar graph comparison between the number of class labels of the documents for the training set and the test set. This is used to determine if one of the classes has an imbalance, that could impact classification results.

```
# Graph class distribution in the training and test sets
def GraphClassData(yTrain, yTest):
    trainCounter = Counter(yTrain)
    testCounter = Counter(yTest)
    sets = ("Train", "Test")
    counts = {
        'Microsoft Windows': (trainCounter[0.0], testCounter[0.0]),
```

Alena McCain
U00833503
brand.18@wright.edu

```
        'Hockey': (trainCounter[1.0], testCounter[1.0])
    }

    x = np.arange(len(sets))  # the label locations
    width = 0.25  # the width of the bars
    multiplier = 0

    fig, ax = plt.subplots(layout='constrained')

    for attribute, measurement in counts.items():
        offset = width * multiplier
        rects = ax.bar(x + offset, measurement, width, label=attribute)
        ax.bar_label(rects, padding=3)
        multiplier += 1

    # Add some text for labels, title and custom x-axis tick labels, etc.
    ax.set_ylabel('Count')
    ax.set_title('Class Distribution in the Training and Test Sets')
    ax.set_xticks(x + width, sets)
    ax.legend(loc='upper left', ncols=3)
    ax.set_ylim(0, 500)

    plt.show()
```

The final section of the code is the main program. It reads the input data, graphs the class distribution, creates and runs the classifiers, and lastly outputs the performance data for each classifier to their respective files. The performance data used are accuracy, confusion matrices, precision, recall, and F1 score, as mentioned above.

```
################### MAIN CODE ############################

# Read in files
```

Alena McCain
U00833503
brand.18@wright.edu

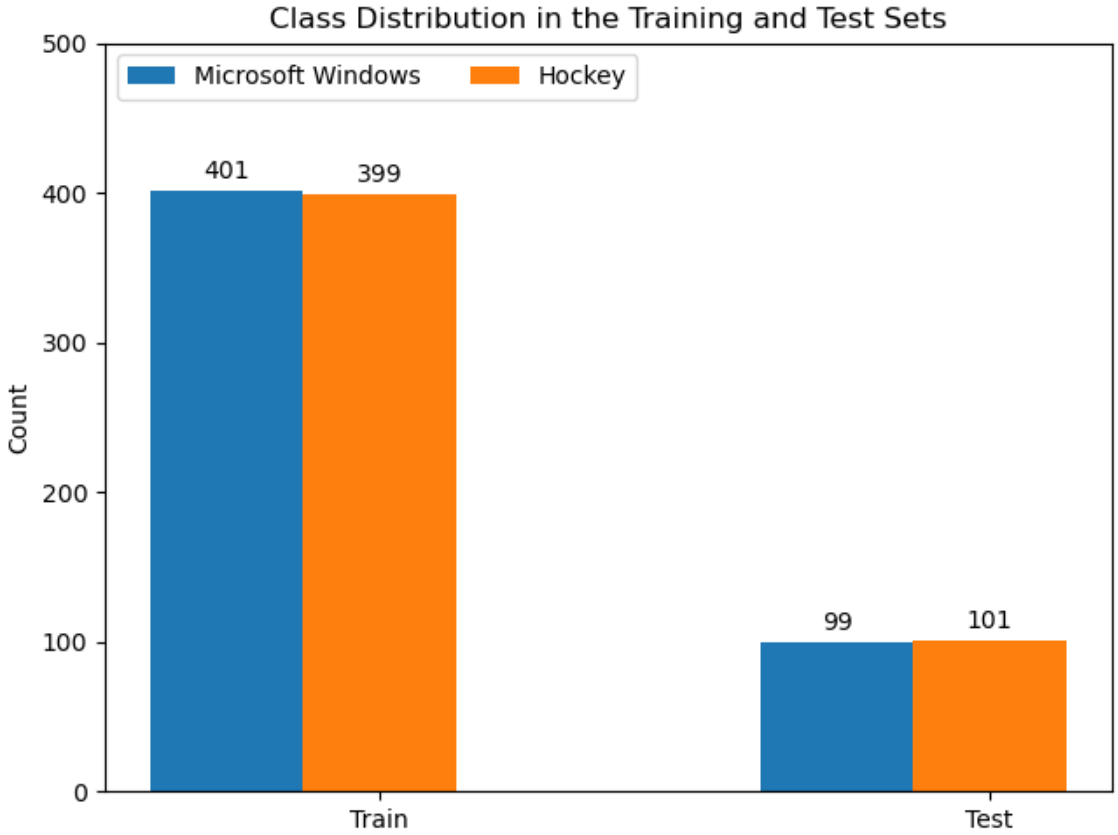
```
xTrain = ReadInputData("./2Newsgroups/trainMatrixModified.txt")

yTrain = ReadClassData("./2Newsgroups/trainClasses.txt")

xTest = ReadInputData("./2Newsgroups/testMatrixModified.txt")

yTest = ReadClassData("./2Newsgroups/testClasses.txt")


# Graph class distribution

GraphClassData(yTrain, yTest)


# Run classifiers

nbAc, nbMat, nbP, nbR, nbF1 = RunSkLearnMultinomialNB(xTrain, yTrain, xTest,
yTest)

svmAc, svmMat, svmP, svmR, svmF1 = RunSkLearnSVM(xTrain, yTrain, xTest, yTest)

knnAc, knnMat, knnP, knnR, knnF1 = RunKNearestNeighbors(xTrain, yTrain, xTest,
yTest, k=5)


# Write classifier stats to files

WriteStatsToFile("NB.txt", nbAc, nbMat, nbP, nbR, nbF1)

WriteStatsToFile("SVM.txt", svmAc, svmMat, svmP, svmR, svmF1)

WriteStatsToFile("KNN.txt", knnAc, knnMat, knnP, knnR, knnF1)
```

Results

After looking at the class distribution data of the training set in the test set (shown below), it appears they are both equally represented in this data set. Therefore, using accuracy alone should

Alena McCain
U00833503
brand.18@wright.edu

provide a precise comparison of the classifiers.



When comparing the three classifiers, naïve Bayes performed the best with 99% accuracy. The linear SVM classifier performed the second best with 98% accuracy, and lastly, the kNN classifier performed the worst with 97% accuracy. While these results may seem similar, the distinctive features of each classifier accentuate the differences in accuracy.

Starting with kNN, that algorithm's performance is completely dependent on the value of k. The algorithm only makes test document classifications based on similarity to a small subset of documents out of the whole training set. As the value of K increases, it is safe to assume that the accuracy of the classifier could increase because of an increase in knowledge of the training set. However, at a certain point increasing the value of K becomes meaningless if K accounts for many documents in the training. The algorithm would just be classifying new documents based on whichever class contains more documents. At that point, one might as well use a classifier that examines all the documents in the training set, such as naïve Bayes.

The linear SVM classifier performed slightly better than kNN for a few reasons. kNN can produce a nonlinear boundary, while linear SVM produces only produces a linear boundary. Therefore, one can assume that this data set is linear in nature. While it is technically true that the kNN classifier can produce a boundary that resembles linearity, it would have difficulty with the test documents that are similar to both classes, because the k closest documents could easily come from either class. Even if k was sufficiently large, the k closest documents still probably wouldn't have a linear spread. On the other

Alena McCain
U00833503
brand.18@wright.edu

hand, the linear SVM classifier can observe all the documents of different classes that are similar and make an appropriate linear boundary between them., the linear SVM classifier will usually perform better than kNN when given a dataset with a linear distribution. However, it should be noted that the performance of these two classifiers could easily be reversed, if the underlying data set was not suspected to be linear.

Lastly, naïve Bayes performed better than linear SVM and kNN because naïve Bayes uses an entirely probabilistic model to make classifications. Therefore, it can make class predictions based on the entire set of documents as opposed to just the documents that are close to the class boundary, or a small subset of k documents. However, naïve Bayes does have a minor drawback because it assumes the probability of the words in each document are independent of each other. However, in this case, that is probably not an issue because the classes appear to be quite distinct. This means they probably don't share many similar words, making the documents easier to classify with a word independence assumption.

It should be noted that even though the kNN classifier produced the lowest accuracy it doesn't require training time, making it the fastest of the three classifiers. Depending on how accurate this classifier would need to be in the real world, one might be willing to make the trade-off between the classifiers to improve processing time at the cost of accuracy.

When looking at the linear SVM and kNN confusion matrices, one can estimate that 4 to 6 of the test documents are near the class boundary. This is because linear SVM incorrectly classified 4 documents, kNN incorrectly classified 6 documents, and theoretically, some of those documents could be the same. However, it is more likely that only four of the documents are close to the class boundary because as stated previously the underlying data set appears to be linear, which means SVM would produce the more natural boundary. When looking at the naïve Bayes confusion matrix, it can be assumed that one test document is very similar to both classes. Even the naïve Bayes classifier could not classify it correctly using probabilities, as opposed to boundaries like the other classifiers.

When looking at all three of the confusion matrices, they all had more difficulty with false negatives than false positives. This can also be seen because the precision is lower than recall in all 3 classifiers. This means that more hockey documents also contain words that overlap with Microsoft Windows documents. Since the class label is simply an operating system, it could have a wider array of specific topics. For example, there could be a document on a hockey video game that runs on Microsoft Windows. It could easily be labeled as either class depending on whether one wants to focus on the videogame itself, or the operating system it is played on. This observation doesn't make a distinction between classifiers, but it does give some ideas regarding the nature of the underlying data set and shows how human interpretation can be subjective.