Code Report

Introduction

Although this course was taught exclusively with MatLab, I have decided to use Python as my language of choice because it is more familiar and there are plenty of machine learning libraries that would make my work easier. I have also used the textbook as reference for coding guidance. I have not created enough graphs to present the data.

Tools and Libraries:

- Python
- PyCharm & Repl.it
- Sklearn
- Numpy
- Pandas
- Matplotlib

Task 1

Task 1 was to preprocess data. Preprocessing data in machine learning is a very crucial thing to do so when I ran into trouble, it was easy to find solutions on the internet.

I use pandas to read the data and sklearn's "train_test_split" method to split my data into training and test which also shuffles them.

```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.297, shuffle=True)
```

I, then, use sklearn's normalization method to normalize my data.

```
X_test = preprocessing.normalize(X_test)

X_train = preprocessing.normalize(X_train)
```

Task 2

For the k-fold cross validation, sklearn's methods, KFold and cross_val_score, is the best suited as it is simple and intuitive. A pipeline was created to make it easier to pass data through the fold and it also gives a accuracy score.

```
kfold = StratifiedKFold(n_splits=10).split(X_train, Y_train)
scores = []
for k, (train, test) in enumerate(kfold):
    pipeline.fit(X_train[train], Y_train[train])
    score = pipeline.score(X_train[test], Y_train[test])
    scores.append(scores)
    print('Fold: %2d, Accuracy: %.3f' % (k+1, score))
scores = cross_val_score(estimator=pipeline, X=X_train, y=Y_train, cv=10, n_jobs=1)
```

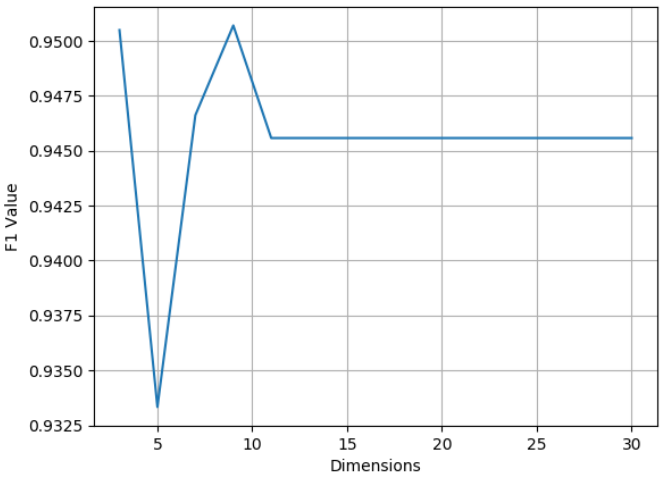
For the decision tree, I pass it through a GridSearchCV method because it is the easiest to implement as the decision tree classifier could be passed through it.

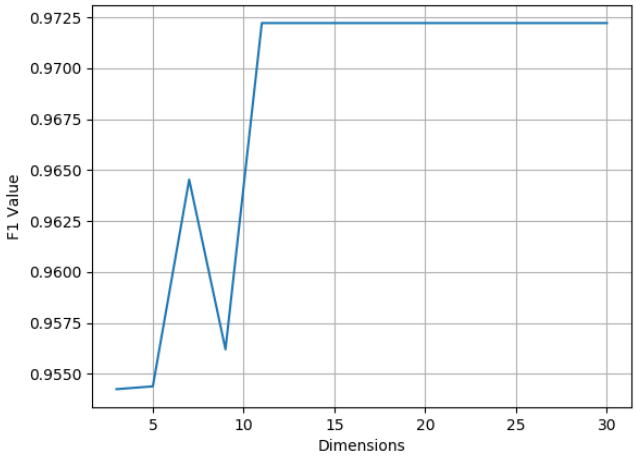
```
gs = GridSearchCV(estimator=DecisionTreeClassifier(random_state=0),
param_grid=[{'max_depth': [1, 2, 3, 4, 5, 6, 7, None]}], scoring='accuracy', cv=10)
```

I pass through n numbers for n components for the PCA through a loop which then finds the accuracy score as well as the F1, precision, and recall scores for each PCA.

```
n = [3, 5, 7, 9, 11]
pipe_svc.fit(X_train, Y_train)
f1_scores = []
for num in n:
    X_train, X_test, Y_train, Y_test = process_data()
    pca = PCA(n_components=num, random_state=1)
    train = pca.fit_transform(X_train)
    scores = cross_val_score(gs, train, Y_train, cv=10)
    print('PCA ' + str(num) + ' accuracy: %.3f +/- %.3f' % (np.mean(scores), np.std(scores)))
    y_pred = pipe_svc.predict(X_test)
    f1 = f1_score(Y_test, y_pred)
    f1_scores.append(f1)
    print('F1: %.3f' % f1)
    print('Precision: %.3f' % precision_score(y_true=Y_test, y_pred=y_pred))
    print('Recall: %.3f' % recall_score(y_true=Y_test, y_pred=y_pred))
```

The graphs below vary significantly because of the shuffling of data, however PCA 11 is the most consistent.





Task 3

Task 3 was the easiest to implement because the methods used in this method is basically the same as Task 2. Instead of passing a decision tree, an instance of SVC was created with different kernels. I use 11 as my components for the PCA as it seemed to have the greatest accuracy. Below is an example of an SVM method, the difference between these methods is the kernel.

```
svm1 = SVC(kernel='rbf')
svm1.fit(X_train, Y_train)
scores = cross_val_score(svm1, X_train, Y_train, cv=10, scoring='accuracy')
print('\nRBF')
print('Train Accuracy: %.3f' % svm1.score(X_train, Y_train))
print('Test Accuracy: %.3f\n' % svm1.score(X_test, Y_test))
kfold = StratifiedKFold(n_splits=10).split(X_train, Y_train)
scores = []
for k, (train, test) in enumerate(kfold):
    pipeline.fit(X_train[train], Y_train[train])
    score = pipeline.score(X_train[test], Y_train[test])
    scores.append(scores)
    print('Fold: %2d, Accuracy: %.3f' % (k + 1, score))
scores = cross_val_score(estimator=pipeline, X=X_train, y=Y_train, cv=10, n_jobs=1)
print("CV accuracy scores: %s" % scores)
print("CV accuracy: %.3f +/- %.3f" % (np.mean(scores), np.std(scores)))
```

Task 5

I was not sure how to approach the overfitting problem, so I don't believe I implemented any precautions for it. In terms of time complexity, I think the decision tree took longer than the SVM. Maybe it is because of how I created the code, but it took significantly longer to run than everything else. I think for a project like this, I would implement the SVM because it doesn't

take long to run and it’s not over complicated. It also provided great accuracy compared to the decision tree.