Assignment 3 – Report

Name: Zhe Lin - (linz38)

001422116

Content

Q1.

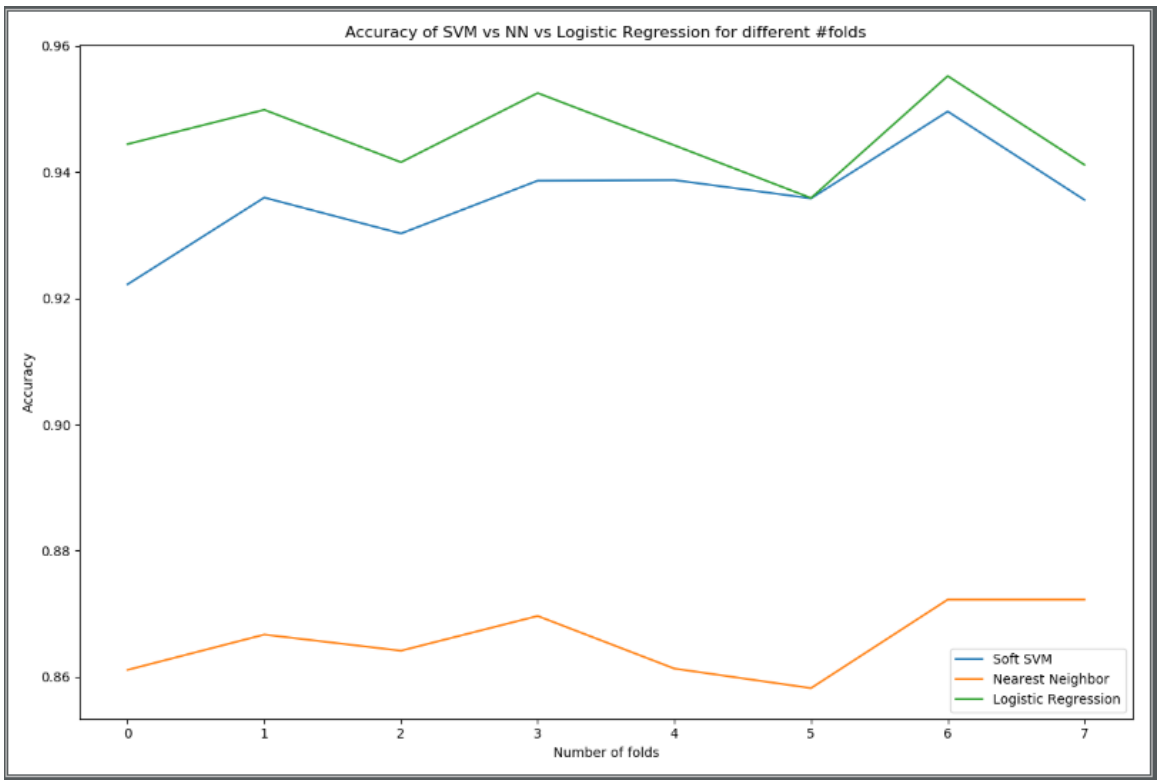


Figure 1. base classifier

1. For increasing folds, the training dataset will close to total dataset, it has smaller bias. Consequently, the accuracy tends to increase with the number of folds. When we choose folds, we need minimize the between-dataset variance. We could not say that more folds are better, if we choose more folds, it will take more time to evaluate it.

2. Because nearest neighbor only considers one point. Due to the dense points, it is easily affected by noise and cause errors, so that it cannot completely represent the category of similar data.

Q2.

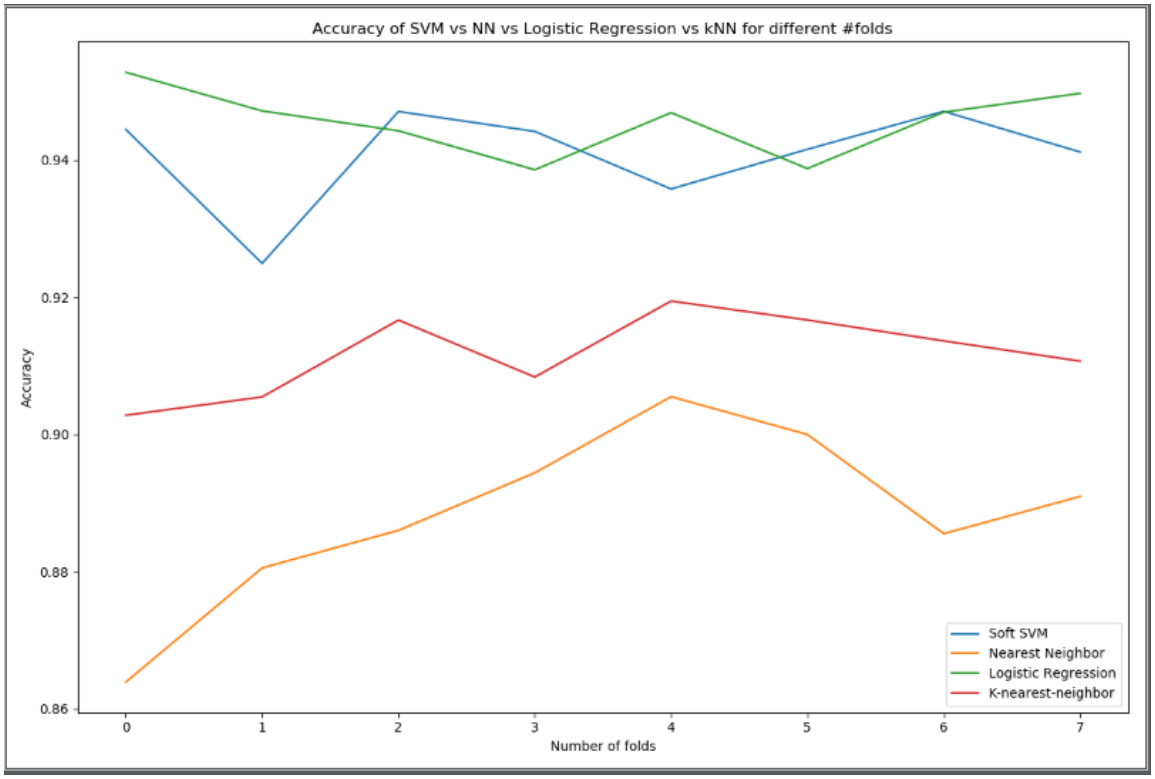


Figure 2. kNN

```
# This is our k-nearst neighbor classifer
class MyKnearstNeighbor(BaseEstimator, ClassifierMixin):

    def __init__(self, demo_param='demo'):
        self.demo_param = demo_param
        return

    def fit(self, X, Y):
        # Check that X and y have correct shape
        X, Y = check_X_y(X, Y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(Y)
        self.X_ = X
        self.Y_ = Y

        # self.k = 11
        return self

    def predict(self, X):
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'Y_'])
        # Input validation
        X = check_array(X)

        Y = []
        for x in X:
            dist = np.argpartition([i[0] for i in euclidean_distances(self.X_, [
x])], 11)[:11]
```

```
            # Count number of occurrences of each value
            bincount = np.bincount([self.Y_[i] for i in dist])
            # get the max index
            Y.insert(len(Y), np.argmax(bincount))
        return Y
```

Yes, K-nearest-neighbor works better than nearest-neighbor in this test. Because, as mentioned before, classifier only take the closest data point. Due to intensive data points, the closest point may not fully represent the category of similar data, so it will be affected by noise and cause errors. However, kNN classifier takes the number of k closest data points, these data can reflect the category in this small space.

Q3.

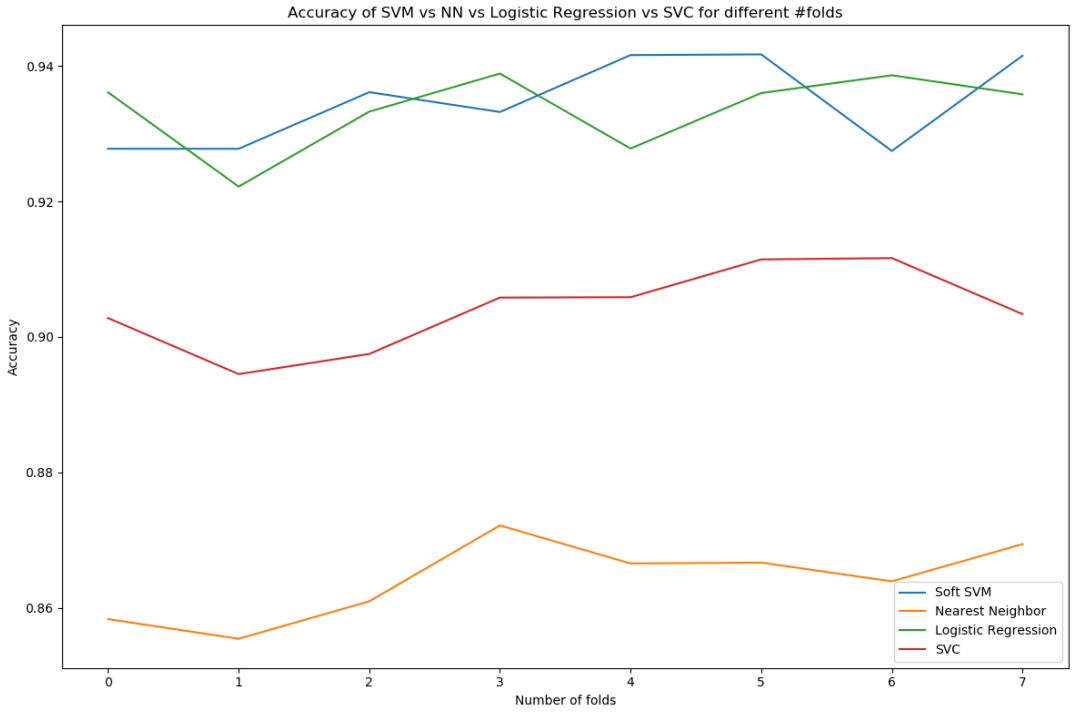


Figure 3. SVC

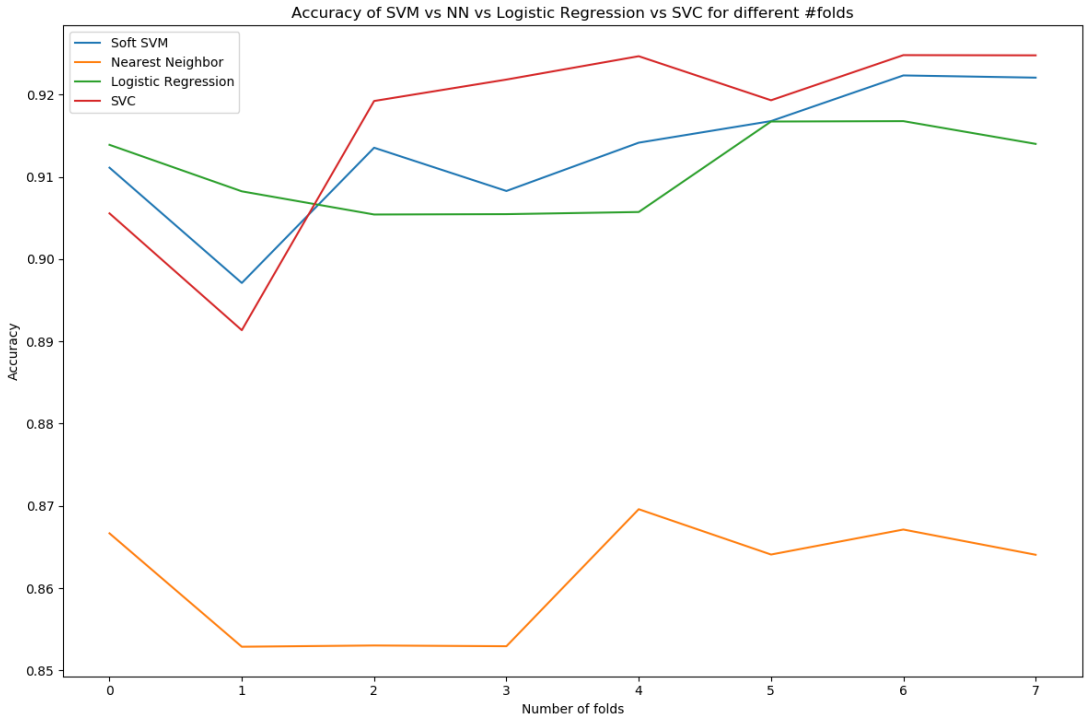


Figure 4.SVC_2

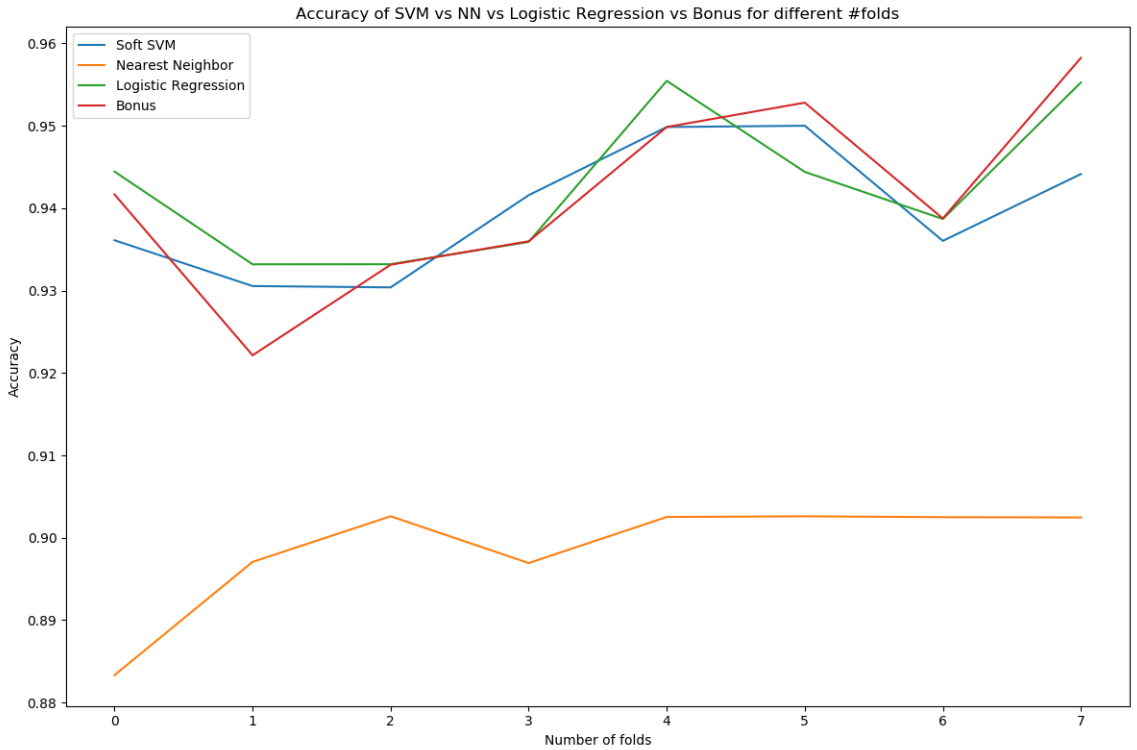
```
1.  # This is our SVC
```

```
class MyKernelVMClassifier(BaseEstimator, ClassifierMixin):

    def __init__(self, demo_param='demo'):
        self.demo_param = demo_param
        return

    def fit(self, X, Y):
        # Check that X and y have correct shape
        X, Y = check_X_y(X, Y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(Y)
        self.X_ = X
        self.Y_ = Y

        # the recommand C values: 10, 100, 1000
        self.clf = SVC(kernel='rbf', gamma='scale', max_iter=1000, C=1000)
        self.clf.fit(X, Y)
        return self


    def predict(self, X):
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'Y_'])
        # Input validation
        X = check_array(X)
        return self.clf.predict(X)
```

1. In general, the kernelized version of SVM is not work better than linear SVM. For kernelized version of SVM, in high dimensional spaces, it has a large number of features, the distance between points will become less meaningful, which leads the less meaningful of the kernel. For linear SVM, it is easier to create boundary.
2. The kernelized version of SVM address the ‘bias’ issue of linear SVM.

Bonus:

Before we applied SVM classifier, we can apply Univariate Feature Selection. Univariate feature selection will select the informative features, it will help SVM classifier to increase the weight attributed to the significant features.




```
# Bonus Question
class MyBonusQuestion(BaseEstimator, ClassifierMixin):

    def __init__(self, demo_param='demo'):
        self.demo_param = demo_param
        return

    def fit(self, X, Y):
        # Check that X and y have correct shape
        X, Y = check_X_y(X, Y)
        # Store the classes seen during fit
        self.classes_ = unique_labels(Y)
        self.X_ = X
        self.Y_ = Y

        self.selector = SelectPercentile(f_classif, percentile=10)
        self.selector.fit(X, Y)
        self.clif = LogisticRegression(random_state=0, solver='lbfgs', multi_cla
ss='multinomial')
        self.clif.fit(self.selector.transform(X), Y)
        # Return the classifier
        return self

    def predict(self, X):
```

```
        # Check is fit had been called
        check_is_fitted(self, ['X_', 'Y_'])
        # Input validation
        X = check_array(X)

        return self.clif.predict(self.selector.transform(X))
```