**Assignment 3 – Report**

Name: Zhe Lin - (linz38)

001422116

Content

[Q1. 2](#_Toc4146215)

[Q2. 3](#_Toc4146216)

[Q3. 5](#_Toc4146217)

[Bonus: 7](#_Toc4146218)

# Q1.

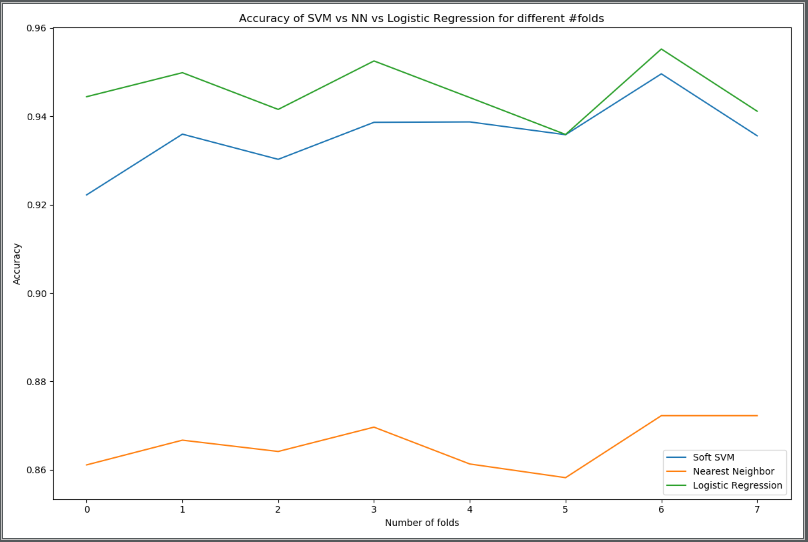


Figure 1. base classifier

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.

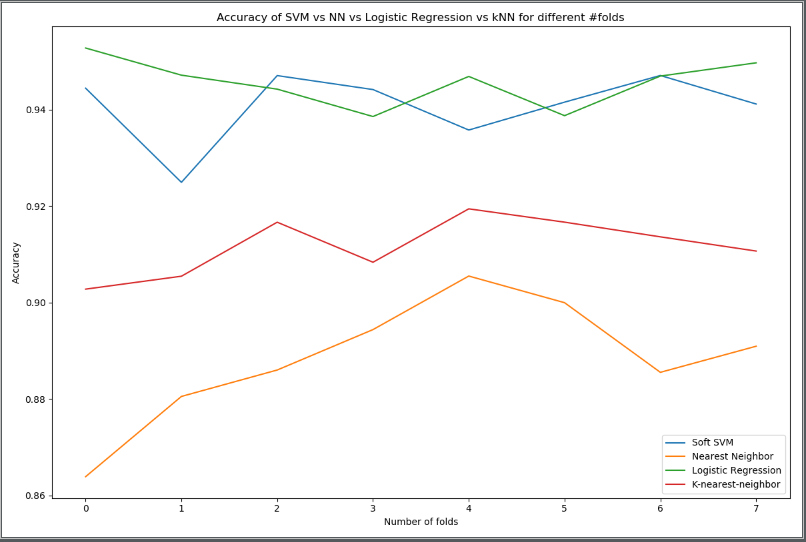


Figure 2. kNN

1. # This is our k-nearst neighbor classifer
2. **class** MyKnearstNeighbor(BaseEstimator, ClassifierMixin):
4. **def** \_\_init\_\_(self, demo\_param='demo'):
5. self.demo\_param = demo\_param
6. **return**
8. **def** fit(self, X, Y):
9. # Check that X and y have correct shape
10. X, Y = check\_X\_y(X, Y)
11. # Store the classes seen during fit
12. self.classes\_ = unique\_labels(Y)
13. self.X\_ = X
14. self.Y\_ = Y
16. # self.k = 11
17. **return** self
19. **def** predict(self, X):
20. # Check is fit had been called
21. check\_is\_fitted(self, ['X\_', 'Y\_'])
22. # Input validation
23. X = check\_array(X)
25. Y = []
26. **for** x **in** X:
27. dist = np.argpartition([i[0] **for** i **in** euclidean\_distances(self.X\_, [x])], 11)[:11]
28. # Count number of occurrences of each value
29. bincount = np.bincount([self.Y\_[i] **for** i **in** dist])
30. # get the max index
31. Y.insert(len(Y), np.argmax(bincount))
32. **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.

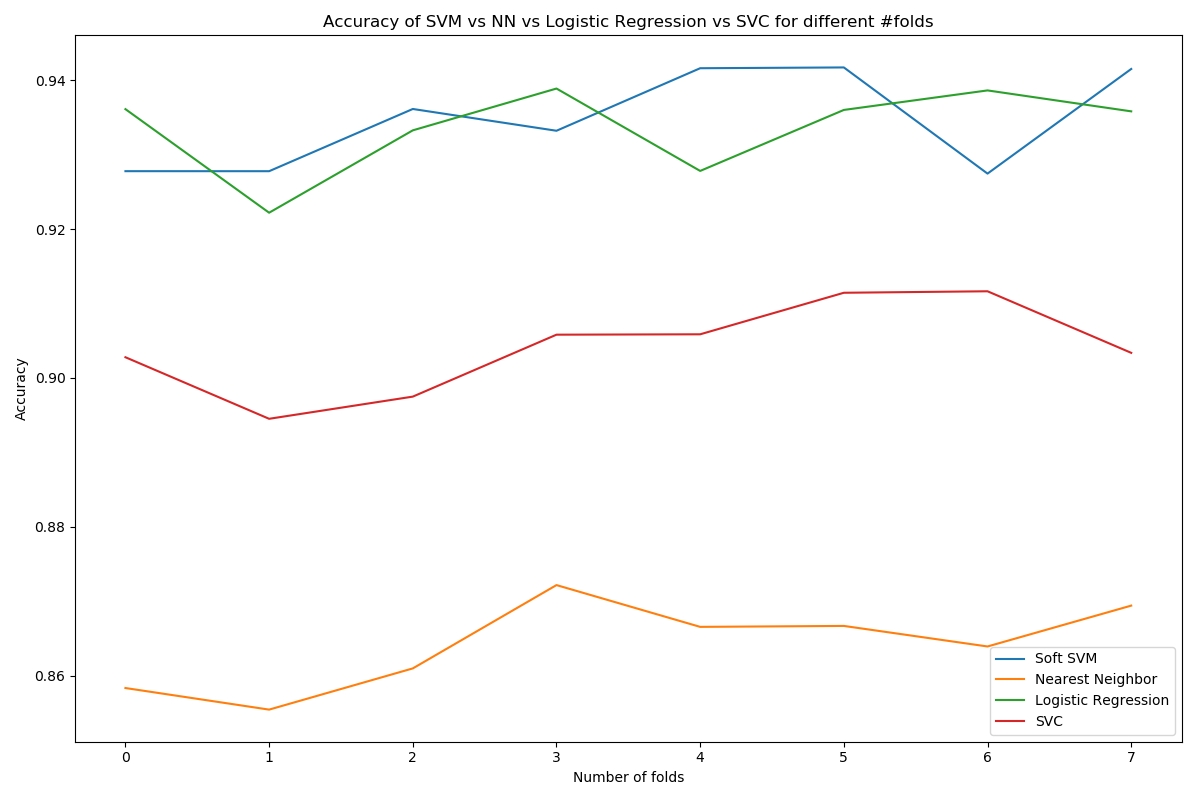


Figure 3. SVC

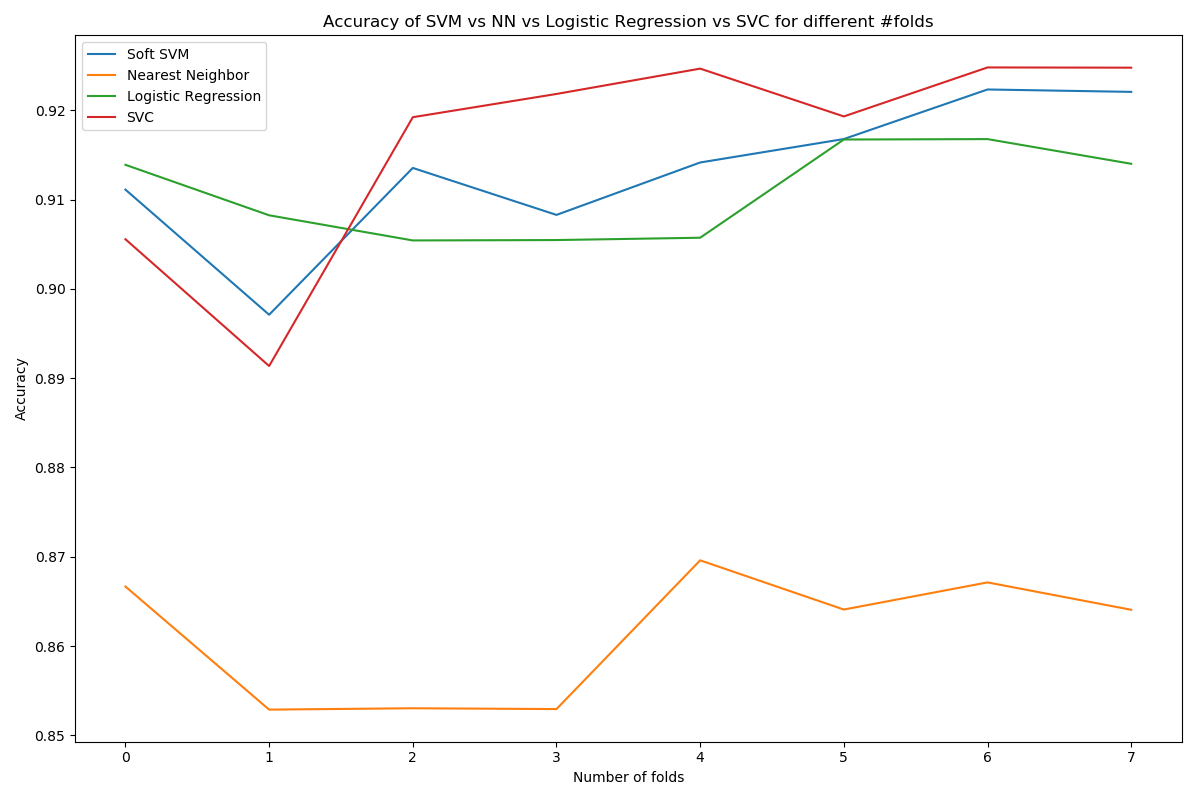


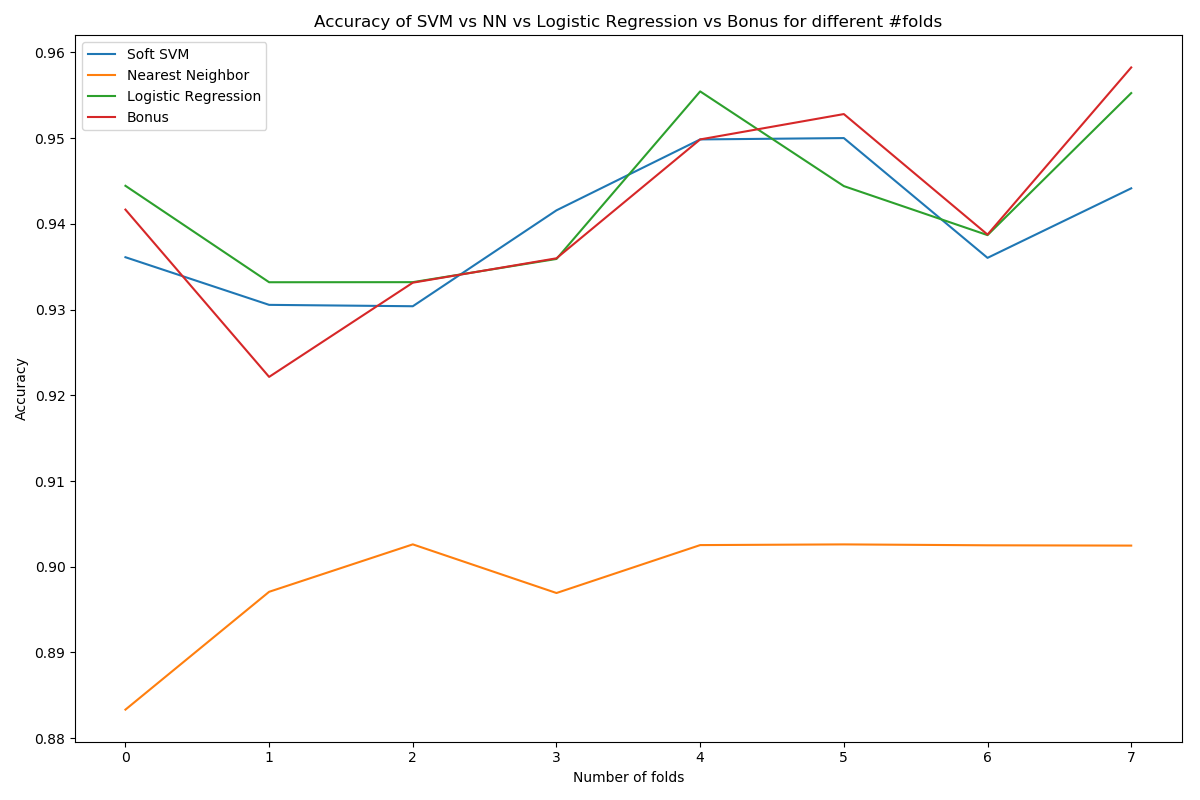
Figure 4.SVC\_2

1. # This is our SVC
2. **class** MyKernelVMClassifier(BaseEstimator, ClassifierMixin):
4. **def** \_\_init\_\_(self, demo\_param='demo'):
5. self.demo\_param = demo\_param
6. **return**
8. **def** fit(self, X, Y):
9. # Check that X and y have correct shape
10. X, Y = check\_X\_y(X, Y)
11. # Store the classes seen during fit
12. self.classes\_ = unique\_labels(Y)
13. self.X\_ = X
14. self.Y\_ = Y
16. # the recommand C values: 10, 100, 1000
17. self.clf = SVC(kernel='rbf', gamma='scale', max\_iter=1000, C=1000)
18. self.clf.fit(X, Y)
19. **return** self

22. **def** predict(self, X):
23. # Check is fit had been called
24. check\_is\_fitted(self, ['X\_', 'Y\_'])
25. # Input validation
26. X = check\_array(X)
27. **return** self.clf.predict(X)
28. 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.
29. 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.



1. # Bonus Question
2. **class** MyBonusQuestion(BaseEstimator, ClassifierMixin):
4. **def** \_\_init\_\_(self, demo\_param='demo'):
5. self.demo\_param = demo\_param
6. **return**
8. **def** fit(self, X, Y):
9. # Check that X and y have correct shape
10. X, Y = check\_X\_y(X, Y)
11. # Store the classes seen during fit
12. self.classes\_ = unique\_labels(Y)
13. self.X\_ = X
14. self.Y\_ = Y
16. self.selector = SelectPercentile(f\_classif, percentile=10)
17. self.selector.fit(X, Y)
18. self.clif = LogisticRegression(random\_state=0, solver='lbfgs', multi\_class='multinomial')
19. self.clif.fit(self.selector.transform(X), Y)
20. # Return the classifier
21. **return** self
23. **def** predict(self, X):
24. # Check is fit had been called
25. check\_is\_fitted(self, ['X\_', 'Y\_'])
26. # Input validation
27. X = check\_array(X)
29. **return** self.clif.predict(self.selector.transform(X))