STA 629 - Homework 3

Anthony Bernardi

October 22nd, 2024

1 Problem 1

1.1 Question and Initial Code Work

This problem uses the training and test sets of zip code handwriting data from the previous Homework. In this question, we are limiting consideration to the digits between 3 and 8. We will compare and contrast the following classifiers in terms of test misclassification error.

- K Nearest Neighbors
- Regularized Logistic Regression
- Linear SVMs
- Kernel SVMs

Use cross-validation to select the best tuning parameters for each classifier. Interpret the results and compare the performance of the classifiers. Which performs best? Why?

In this problem, we import the data in the same manner as the previous homework. The data is imported and the digits are filtered to only include the digits 3 through 8 as before, with the code included below.

```
import pandas as pd
import numpy as np

df_whole = pd.read_csv('/home/adbucks/Downloads/zip.train', delim_whitespace = Tru
```

```
print(df_whole.iloc[:,0].unique())

print(df_whole.iloc[:,0].values)

digits = range(3, 9)

df_whole.columns = ['Digit'] + ['Pixel' + str(i) for i in range(1, 257)]
print(df_whole.head())

# getting only digits 3 through 8
df_train = df_whole.loc[df_whole.iloc[:,0].isin(digits)]

'''
print(df_train.head())
print(df_train.iloc[:,0].unique())
print(df_train.shape)
'''

# ensuring that we have the same as before

# we can now do this with our future testing data
df_te_whole = pd.read_csv('/home/adbucks/Downloads/zip.test', delim_whitespace = T

# renaming the columns in the same way
df_te_whole.columns = ['Digit'] + ['Pixel' + str(i) for i in range(1, 257)]
print(df_te_whole.head())

df_test = df_te_whole.loc[df_te_whole.iloc[:,0].isin(digits)]

'''
print(df_test.head())
print(df_test.iloc[:,0].unique())
print(df_test.shape)
'''
# we can now start divying up the data into the response vector and our feuature m
# and of course with a train and test split
```

We then split the data into the response vector and the features matrix, for both the training and testing data.

```
X_train = df_train.iloc[:, 1:]
y_train = df_train.iloc[: , 0]

# doing the same for the test now
X_test = df_test.iloc[:, 1:]
y_test = df_test.iloc[: , 0]
```

1.2 K Nearest Neighbors

We will now use Grid Search and Cross Validation to find the optimal tuning parameter for the K Nearest Neighbors classifier, given the data. The code for this is below.

```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier

# establishing the parameter grid
param_grid = {'n_neighbors': np.arange(1, 25) ,
              'weights': ['uniform', 'distance'],
              'metric': ['euclidean', 'manhattan']}

knn_gs = GridSearchCV(KNeighborsClassifier(), param_grid,
                      cv = 5)

knn_grid = knn_gs.fit(X_train, y_train)
print("Finding the best KNN model...")
print(knn_grid.best_params_)
print(knn_grid.best_score_)
```

Using formatted printing, we see that the optimal K Nearest Neighbors model has a K value of 4, with standard euclidean distance.

We then use this optimal model to predict the test data, and calculate the misclassification error.

```
knn_best = KNeighborsClassifier(n_neighbors = 4, metric = 'euclidean', weights = '
```

```
y_pred = knn_best.predict(X_test)

# evaluating our best model now
from sklearn.metrics import classification_report, confusion_matrix

print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

In this case, we get a 95% accuracy rate, which is quite good.
We can now move on to Regularized Logistic regression.

1.3 Regularized Logistic Regression

Again, we will use grid search and cross-validation to find the optimal tuning parameter for the Regularized Logistic Regression model.

```
from sklearn.linear_model import LogisticRegression

# establishing the parameter grid again

lr_param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                 'penalty': ['l1', 'l2']}

lr_gs = GridSearchCV(LogisticRegression(), lr_param_grid,
                     cv = 5)

lr_grid = lr_gs.fit(X_train, y_train)
print("Finding the best Logistic Regression model...")
print(lr_grid.best_params_)
print(lr_grid.best_score_)

# we see that an l2 penalty with a C value of 0.1 is the best model.
# we will now use that model to make predictions.

lr_best = LogisticRegression(C = 0.1, penalty = 'l2').fit(X_train, y_train)
y_pred_lr = lr_best.predict(X_test)

# evaluating our best model now
```

```
print(confusion_matrix(y_test, y_pred_lr))
print(classification_report(y_test, y_pred_lr))
```

We see that grid search yields a best model with an L2 penalty and a C value of 0.1. This gives a test accuracy of 93%.

We will have to compare this to an elastic net with a grid search for the optimal L1 penalty ratio, but we can move on to the Linear SVM case for now.

1.4 Linear SVM

We will continue using grid search and cross-validation to tune hyperparameters for the Linear SVM model, with the code below.

```
from sklearn.svm import LinearSVC

# establishing our parameter grid as before

lin_svc_param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                      'penalty': ['l1', 'l2'],
                      'loss': ['hinge', 'squared_hinge']}

lin_svc_gs = GridSearchCV(LinearSVC(), lin_svc_param_grid,
                          cv = 5)

lin_svc_grid = lin_svc_gs.fit(X_train, y_train)

print("Finding the best Linear SVM model...")
print(lin_svc_grid.best_params_)
print(lin_svc_grid.best_score_)
```

We see that the best model in the Linear SVM case has a C parameter value of 0.01 with an L2 penalty and a hinge loss function.

We will now use this model to predict the test data and calculate the misclassification error.

```
lin_svc_best = LinearSVC(C = 0.01, loss = 'squared_hinge', penalty = 'l2').fit(X_t
y_pred_lin_svc = lin_svc_best.predict(X_test)
```

```
 evaluating the best model now
print(confusion_matrix(y_test, y_pred_lin_svc))
print(classification_report(y_test, y_pred_lin_svc))
```

This gives a test accuracy of 93%, which is the same as the regularized logistic regression model. This might be due to the higher bias of the linear SVM model, we will compare this to the kernel SVM model below.

1.5 Kernel SVM

Grid search and cross validation done as before, with the code below.

```
from sklearn.svm import SVC

# establishing our parameter grid

svc_param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000],
                  'kernel': ['linear', 'poly', 'rbf', 'sigmoid']}

svc_gs = GridSearchCV(SVC(), svc_param_grid,
                      cv = 5)

svc_grid = svc_gs.fit(X_train, y_train)

print("Finding the best Kernel SVM model...")
print(svc_grid.best_params_)
print(svc_grid.best_score_)

# we have a C value of 10 and a polynomial kernel with an impressive 98% train acc
svc_best = SVC(C = 10, kernel = 'poly').fit(X_train, y_train)

y_pred_svc = svc_best.predict(X_test)

# evaluating on the test data now
print(confusion_matrix(y_test, y_pred_svc))
print(classification_report(y_test, y_pred_svc))
```

We note that the optimal model in this case is a polynomial kernel with a C value of 10. This gives a test accuracy of 95%, which is the same as the

K Nearest Neighbors model, both being the best performing models using Cross-Validation.

1.6 Comparison of Models

It seems worth noting that both the K Neareest Neighbors and Kernel SVM models performed the best in terms of test accuracy, with a 95% accuracy rate. This is likely due to the fact that the K Nearest Neighbors model is a non-parametric model, and the Kernel SVM model is a non-linear model. This is in contrast to the Linear SVM and Regularized Logistic Regression models, which are both linear models.

When modeling this type of problem given the current data, it seems prudent to trade off some bias and interpretability for a more flexible model with higher variance. The trade off seems worthwhile in this case.

2 Problem 2

2.1 Question and Initial Code Work

Using the same data from the earlier problem, write a function to perform 5-fold Cross-Validation for the tuning parameter for regularized Logistic Regression. Compare and contrast the results for minimizing the Cross-Validation error for hyperparameter tuning with employing the One Standard Error Rule.

We will also use the One Standard Error Rule for the three loss functions.

- Misclassification Error
- Binomial Deviance Loss
- Hinge Loss

From the first section of the Homework, we know that for this Regularized Logistic Regression model, the best model had an L2 penalty and a C value of 0.1. We will use this as a starting point for the tuning parameter. We will also say that these parameters were selected when minimizing misclassification loss, giving us our first set of parameters for this problem.

We will now proceed with contrasting this result with the One Standard Error Rule, and the loss functions that we will use in conjunction with that.

We have our cross-validation for regularized logistic regression from earlier in the assignment, and we will now include the One Standard Error Rule.

We will start with the case of Hinge Loss, with the code given below.

2.2 Hinge Loss

```
from sklearn.svm import LinearSVC

lin_svc = LinearSVC(loss = 'hinge', max_iter = 10000)

# now establishing hte parameter grid
param_grid_lin_svc = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

lin_svc_gs = GridSearchCV(lin_svc , param_grid = param_grid_lin_svc, cv = 5)

# fitting
lin_svc_grid = lin_svc_gs.fit(X_train, y_train)

# getting the best parameters
print("The best parameters using Hinge Loss are: ", lin_svc_grid.best_params_)
print("The best score using Hinge Loss is: ", lin_svc_grid.best_score_)

# can dig a little deeper to do the one standard error rule
mean_test_score = lin_svc_grid.cv_results_['mean_test_score']
std_test_score = lin_svc_grid.cv_results_['std_test_score']

# finding the best to use the standard error rule
best_score_idx = np.argmax(mean_test_score)
best_score = mean_test_score[best_score_idx]
best_score_std = std_test_score[best_score_idx]

# getting the threshold now
thresh = best_score - best_score_std

# we now want to find the argmax of C values that are within one standard error of
candidate_indices = [i for i, score in enumerate(mean_test_score) if score > thres
simplest_idx = candidate_indices[np.argmax(mean_test_score[candidate_indices])]
```

```
# getting the best C value in line with the rule
best_C = lin_svc_grid.cv_results_['param_C'][simplest_idx]
best_score_1se = mean_test_score[simplest_idx]

print("The best C value using the one standard error rule is: ", best_C)
print("The best score using the one standard error rule is: ", best_score_1se)
```

The Linear SVC in this case is used to utilize the hinge loss more effectively, given the nature of the data and as it relates to this problem.

We see that in this case, the best C value using the One Standard Error Rule is 0.1, which is the same as the best C value using the Cross-Validation method.

We will now move on to the Binomial Deviance Loss function, which will largely be done in the same way.

2.3 Binomial Deviance Loss

```
from sklearn.metrics import log_loss, make_scorer

lin_ll = LogisticRegression(solver = 'liblinear', max_iter = 10000)

# now establishing the parameter grid

param_g_ll = {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000]}

# trying to DIY a scoring function
scorer = make_scorer(log_loss, greater_is_better = False, needs_proba = True)

gs_ll = GridSearchCV(lin_ll, param_grid = param_g_ll, cv = 5, scoring = scorer)

# fitting the model
gs_ll.fit(X_train, y_train)

mean_test_scores_ll = -gs_ll.cv_results_['mean_test_score'] # flipping sign due to
std_test_scores_ll = gs_ll.cv_results_['std_test_score']

# now trying to find the minimum
best_score_idx_ll = np.argmin(mean_test_scores_ll)
```

```
best_score_ll = mean_test_scores_ll[best_score_idx_ll]
best_score_std_ll = std_test_scores_ll[best_score_idx_ll]

# now finding the threshold
thresh_ll = best_score_ll + best_score_std_ll

# finding the simplest model that is within one standard error of the best model
candidate_indices_ll = [i for i, score in enumerate(mean_test_scores_ll) if score
simplest_idx_ll = min(candidate_indices_ll, key = lambda idx: param_g_ll['C'][idx]

# getting the best C value in line with the rule
best_params_ll_1se = gs_ll.cv_results_['params'][simplest_idx_ll]
best_score_ll_1se = mean_test_scores_ll[simplest_idx_ll]

print("The best parameters using Binomial Deviance Loss are: ", best_params_ll_1se
print("The best score using Binomial Deviance Loss is: ", best_score_ll_1se)
```

This method also gives a C value of 0.1, so there seems to be a consensus when using different methods and loss functions regarding this parameter for regularized logistic regression.

It should be noted that the above code gives a score of 0.20, which is our log loss value. This is either an error and oversight by the author, or a larger phenomenon worth investigating.

2.4 Comparison of Methods

It seems that the One Standard Error Rule is a useful tool for tuning hyperparameters in the context of regularized logistic regression. This is especially true when using different loss functions, as it seems to give a consistent result.

This speaks to the advantage of the One Standard Error Rule, as this can give comparable results, even when moving across different loss functions.

It would be interesting to see how this rule performs in the context of other models, and how it might be used in the context of other hyperparameters, perhaps in models with higher variance or lower bias than Logistic Regression. Also it seems that it would be worthwhile to deploy these methods in a binomial classification case to measure performance.

Overall, the one Standard Error method seems to be well worth pursuing, especially as the number of observations and predictors increases.