STA 629 - Homework 2

Anthony Bernardi

October 3rd, 2024

1 Problem 1

1.1 Question and Primary Code Work

Use the training and test sets of the zip code data from the Textbook's website to answer the following question and compare models. For Problem 1, we will limit the handwritten digit analysis to digits 3 through 8. We will then compare the following classifiers in terms of test misclassification error.

1. Naive Bayes Classifier
2. KNN Classifier
3. LDA
4. Logistic Regression
5. Regularized Logistic Regression
6. Linear SVMs
7. Kernel SVMs

We will now include the relevant Python code for cleaning and preparing the data from the textbook website.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

```
# getting the zip code data from the textbook's website
# we can try to read in our training data, in ASCII text format
# for expedience, we are limiting our consideration to digits between 3 and 8
# that said, we will read in the individual digits
# and then will concatenate them

df_whole = pd.read_csv('/home/adbucks/Downloads/zip.train'
, delim_whitespace = True)

# we will now filter the data to only include the digits 3 and 8
print(df_whole.iloc[:,0].unique())

# now slicing the data frame
print(df_whole.iloc[:,0].values)

digits = range(3, 9)

# need to standardize the column names to further prepare the data
# want to show the first column name as the digit, and the rest
# as the pixel values
# we can do this by renaming the columns
df_whole.columns = ['Digit'] + ['Pixel' + str(i) for i in range(1, 257)]
print(df_whole.head())


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)

# looks good, so we can say that we have our training data of interest

# we can now repeat the process and ensure that we have the right records
# for the test data
df_te_whole = pd.read_csv('/home/adbucks/Downloads/zip.test'
, delim_whitespace = True)
```

```
# doing the same checks before filtering in the same way
'''
print(df_te_whole.head())
print(df_te_whole.iloc[:,0].unique())
print(df_te_whole.iloc[:,0].values)
print(df_te_whole.shape)
'''
# we can now re-name 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)

# now we have our training and test data, and we can compare models
# we'll do some additional implementation for the naive bayes classifier
import random
from sklearn.metrics import confusion_matrix

# we'll just parse out the training and testing data now
X_train = df_train.iloc[:, 1:]
y_train = df_train.iloc[:, 0]

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

We will now include the relevant Python code for the Naive Bayes Classifier implementation.

```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB().fit(X_train, y_train)
```

```
# can now try and predict
y_nb_pred = gnb.predict(X_test)
# quick check
print(y_nb_pred)

# we can now check the accuracy of the model
nb_accuracy = gnb.score(X_test, y_test)
print(nb_accuracy)

# for posterity we can also create a confusion matrix
nb_confusion = confusion_matrix(y_test, y_nb_pred)
print(nb_confusion)
```

This code will have the same format, returning the accuracy score and confusion matrix for each of our methods, so we will include the code for the KNN Classifier now,

```
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors = 3).fit(X_train, y_train)

# we can now predict
y_knn_pred = knn.predict(X_test)
print(y_knn_pred)

# can try and get the score here as well
knn_accuracy = knn.score(X_test, y_test)
print(knn_accuracy)

# and the confusion matrix
knn_confusion = confusion_matrix(y_test, y_knn_pred)
print(knn_confusion)
```

We will now include the code for the LDA implementation.

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

lda = LinearDiscriminantAnalysis().fit(X_train, y_train)
```

```
# using default solver and parameters

# now we can try and predict
y_lda_pred = lda.predict(X_test)

# and get the accuracy
lda_accuracy = lda.score(X_test, y_test)
print(lda_accuracy)

# and the confusion matrix
lda_confusion = confusion_matrix(y_test, y_lda_pred)
print(lda_confusion)
```

Logistic Regression

```
from sklearn.linear_model import LogisticRegression

log_reg = LogisticRegression(random_state = 42).fit(X_train, y_train)
# default for other arguments

# now we can predict
y_log_pred = log_reg.predict(X_test)

# and get the accuracy
log_accuracy = log_reg.score(X_test, y_test)
print(log_accuracy)

# and the confusion matrix
log_confusion = confusion_matrix(y_test, y_log_pred)
print(log_confusion)
```

We will now do Regularized Logistic Regression, with an L1 penalty rather than the default L2 penalty.

```
log_reg_l1 = LogisticRegression(penalty = 'l1', solver = 'saga'
, random_state = 42).fit(X_train, y_train)
# adjust solver to deal with l1 penalty
```

```
y_log_l1_pred = log_reg_l1.predict(X_test)

log_l1_accuracy = log_reg_l1.score(X_test, y_test)
print(log_l1_accuracy)

# just confirming the confusion matrix is the same
log_l1_confusion = confusion_matrix(y_test, y_log_l1_pred)
print(log_l1_confusion)
```

We will also include the Elastic Net Logistic Regression implementation, for penalty comparison.

```
log_reg_en = LogisticRegression(penalty = "elasticnet", solver = 'saga'
, l1_ratio = 0.5, random_state = 42).fit(X_train, y_train)
# matching solver with elastic net penalty

y_log_en_pred = log_reg_en.predict(X_test)

log_en_accuracy = log_reg_en.score(X_test, y_test)
print(log_en_accuracy)

# confusion matrix
log_en_confusion = confusion_matrix(y_test , y_log_en_pred)
print(log_en_confusion)
```

We will now include the code for the Linear SVC implementation.

```
from sklearn.svm import LinearSVC

lin_svm = LinearSVC(random_state = 42).fit(X_train, y_train)
# default arguments everywhere else, penalty and hinge loss

y_lin_svm_pred = lin_svm.predict(X_test)

lin_svm_accuracy = lin_svm.score(X_test, y_test)
print(lin_svm_accuracy)

# confusion matrix
lin_svm_confusion = confusion_matrix(y_test , y_lin_svm_pred)
print(lin_svm_confusion)
```

We will now include the code for the Kernel SVC implementation, with the radial basis function kernel first, as the default argument for the function.

```
from sklearn.svm import SVC

rbf_svm = SVC(random_state = 42).fit(X_train, y_train) # default kernel is rbf

y_rbf_svm_pred = rbf_svm.predict(X_test)

rbf_svm_accuracy = rbf_svm.score(X_test, y_test)
print(rbf_svm_accuracy)

# confusion matrix

rbf_svm_confusion = confusion_matrix(y_test, y_rbf_svm_pred)
print(rbf_svm_confusion)
```

We will now include the code for the Kernel SVC implementation, with the polynomial kernel.

```
poly_svm = SVC(kernel = "poly", random_state = 42).fit(X_train, y_train)

y_poly_svm_pred = poly_svm.predict(X_test)

poly_svm_accuracy = poly_svm.score(X_test, y_test)
print(poly_svm_accuracy)

# confusion matrix
poly_svm_confusion = confusion_matrix(y_test , y_poly_svm_pred)
print(poly_svm_confusion)
```

1.2 Results and Analysis

We ought to note that accuracy for the Naive Bayes Classifier was the lowest at 0.7265, and the KNN Classifier having the highest, with a k of 3 chosen based on domain knowledge and past experience of the classifier algorithm. Comparing accuracy and error rates for k values of 5 and 9 revealed no major difference, and even an accuracy drop off when k is set to 9. It is also worth

noting that, after rounding, both the KNN and Polynomial Kernel SVC had similar accuracy scores of around 0.95.

It is also worth noting that L1-penalized and Elastic Net Logistic Regression were improvements on the standard Logistic Regression model, with the Elastic Net ratio picked to have a 50-50 split between L1 and L2 penalties, as a naive baseline estimate.

Based on the fact that the two top performing models were the KNN Classifier and the Polynomial Kernel SVC, that leads me to infer that the optimal separating hyperplane for the data is non-linear. Further, I believe the Naive Bayes Classifier performed poorly in part, by assuming a Gaussian distribution. In this type of problem, I think a preference ought to be given to non-parametric and non-linear methods, so long as the computational expense is not too high.

The fact that both the RBF and Polynomial kernel methods showed improvement when compared to the Linear Kernel SVC also supports the idea that the data is not linearly separable.

2 Problem 2

We will now use each digit, using the same data as in Problem 1. Now, however, we will include all digits from 0 to 9. We will use only multi-class SVMs, and will compare the 'One Versus One' and 'One Versus All' methods. We will then analyze and interpret the models based on their test error along with their confusion matrices.

2.1 Question and Primary Code Work

We will now split the training and testing data using the entire textbook data set, rather than filtering as in Question 1.

```
# we'll start with the OVA approach for all of the data
# preparing the data first
X_train_w = df_whole.iloc[:, 1:]
y_train_w = df_whole.iloc[: , 0]
X_test_w = df_te_whole.iloc[:, 1:]
y_test_w = df_te_whole.iloc[:,0]
```

Thankfully, much of this work was done in the first problem, so we can now move on to the implementation of the OVA and OVO methods.

```
ova_svm = SVC(decision_function_shape = 'ovr'
, random_state = 42).fit(X_train_w, y_train_w)

y_ova_pred = ova_svm.predict(X_test_w)

ova_accuracy = ova_svm.score(X_test_w, y_test_w)
print(ova_accuracy)

# confusion matrix
ova_confusion = confusion_matrix(y_test_w, y_ova_pred)
print(ova_confusion)
```

We will now include the code for the OVO method.

```
from sklearn.multiclass import OneVsOneClassifier

ovo_sm2 = OneVsOneClassifier(SVC(random_state = 42))

ovo_sm2.fit(X_train_w, y_train_w)

y_ovo_pred2 = ovo_sm2.predict(X_test_w)

ovo_accuracy2 = ovo_sm2.score(X_test_w, y_test_w)
print(ovo_accuracy2)

# confusion matrix
ovo_confusion2 = confusion_matrix(y_test_w, y_ovo_pred2)
print(ovo_confusion2)
```

This allows us to implement and compare the two methods. We will now discuss how they compare in terms of both test error and confusion matrices.

2.2 Results and Analysis

We can see that there is a slight jump in terms of test accuracy when using the OVO method rather than OVA. The OVA method had an accuracy of

0.9472, while the OVO method had an accuracy of 0.9482. This might not seem like a large difference, but it is worth noting that the OVO method is more computationally expensive, as it requires fitting $n(n-1)/2$ models, where n is the number of classes.

The confusion matrices for both methods can be found printed out below, and we will discuss the differences between them now. We will write out the matrices as they appear from the python script.

Here is the confusion matrix for the OVA method:

$$\begin{bmatrix} 355 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 254 & 1 & 0 & 5 & 0 & 3 & 1 & 0 & 0 \\ 2 & 0 & 184 & 2 & 3 & 2 & 0 & 1 & 4 & 0 \\ 2 & 0 & 3 & 147 & 0 & 10 & 0 & 0 & 4 & 0 \\ 0 & 1 & 3 & 0 & 188 & 1 & 1 & 1 & 1 & 4 \\ 2 & 0 & 0 & 3 & 1 & 151 & 0 & 0 & 1 & 2 \\ 3 & 0 & 3 & 0 & 2 & 2 & 159 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 5 & 0 & 0 & 138 & 1 & 2 \\ 2 & 0 & 2 & 2 & 0 & 2 & 1 & 1 & 155 & 1 \\ 0 & 0 & 0 & 1 & 6 & 0 & 0 & 0 & 0 & 169 \end{bmatrix}$$

This is for the OVO method:

$$\begin{bmatrix} 355 & 0 & 2 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 254 & 1 & 0 & 5 & 0 & 3 & 1 & 0 & 0 \\ 2 & 0 & 184 & 2 & 2 & 2 & 0 & 1 & 5 & 0 \\ 2 & 0 & 3 & 147 & 0 & 10 & 0 & 0 & 4 & 0 \\ 0 & 1 & 3 & 0 & 188 & 1 & 1 & 1 & 1 & 4 \\ 2 & 0 & 0 & 3 & 1 & 151 & 0 & 0 & 1 & 2 \\ 2 & 0 & 3 & 0 & 2 & 2 & 159 & 0 & 2 & 0 \\ 0 & 0 & 1 & 0 & 5 & 0 & 0 & 138 & 1 & 2 \\ 2 & 0 & 1 & 2 & 0 & 2 & 0 & 0 & 157 & 2 \\ 0 & 0 & 0 & 1 & 4 & 0 & 0 & 0 & 2 & 169 \end{bmatrix}$$

If we wanted, we could also visualize these matrices using a heatmap, but the raw numbers are sufficient for our purposes. We could even subtract the matrices to see where the differences lie, and how small the differences are.

For understanding, we will want to note that the diagonal elements of the confusion matrix represent the number of correctly classified instances

for each class, and the columns of the matrix represent the predicted digit by the model, where the rows represent the "ground truth" of our data.

When comparing this to the important context given by the data, that a 2.5% error rate is "excellent" we can see that both models get somewhat close. The OVO method is slightly better and has a more accurate confusion matrix, but the trade-off of computational expense in any OVO classification method must be considered.

As far as which digits are most commonly mis-classified, we can see that the digits 3 and 8 are most commonly mis-classified, with 10 and 11 errors respectively. This is a common issue with handwritten digit recognition, as these digits can be written in a similar way.

We can use the following code with Numpy to find the most commonly mis-classified digits.

```
ova_confusion_no_diag = ova_confusion.copy()
np.fill_diagonal(ova_confusion_no_diag, 0)
col_sums = np.sum(ova_confusion_no_diag, axis = 0)
print("Column Sums for OVA: ", col_sums)

# and we can do the same for the OVO confusion matrix
ovo_confusion_no_diag = ovo_confusion2.copy()
np.fill_diagonal(ovo_confusion_no_diag, 0)
col_sums_ovo = np.sum(ovo_confusion_no_diag, axis = 0)
print("Column Sums for OVO: ", col_sums_ovo)
```

This allows us to see that the most commonly mis-classified digits are 4, 5, and 8. Intuitively this makes sense, as these digits can be written in a similar way. Digits like 1, 6, and 7 are less likely to be mis-classified in this case.