Machine Learning

Ho Fong Law n10107321, Kiki Mutiara

Question 1)

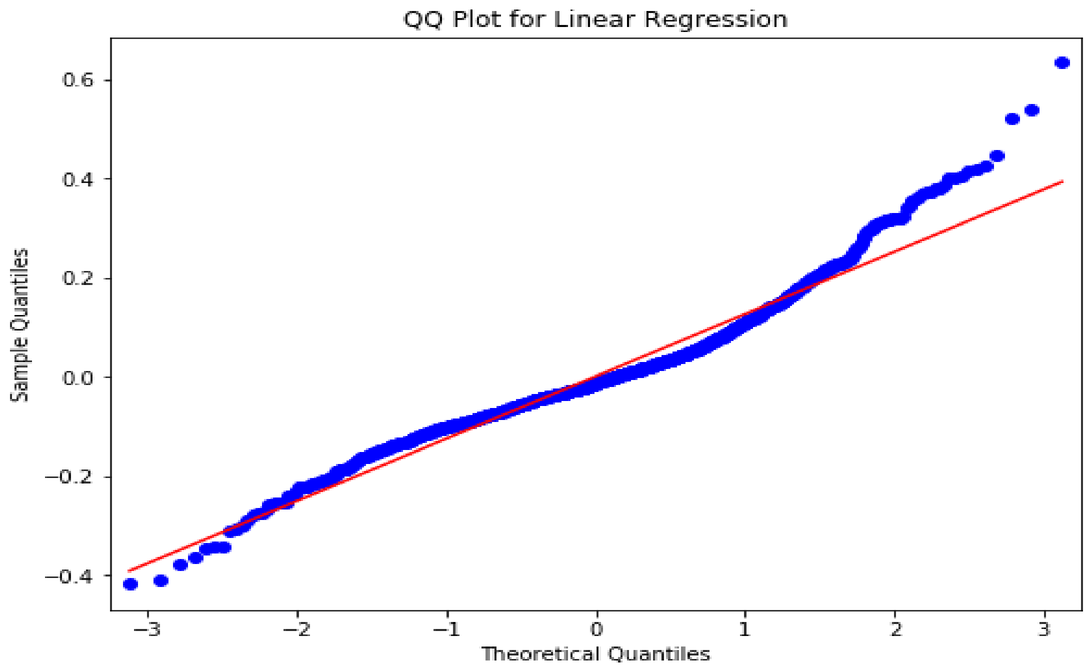
First, import the CSV that we require. exclude the non-analysing columns such as country community etc. After removed selected columns. According to the data from CSV. There are some '?' symbols that can't analysis also. Therefore, we should discard any of this for improvement of the analyses. On the other hand, some information included 'nan' possibility. Thus, we rescan data set and remove them also for finally data shape

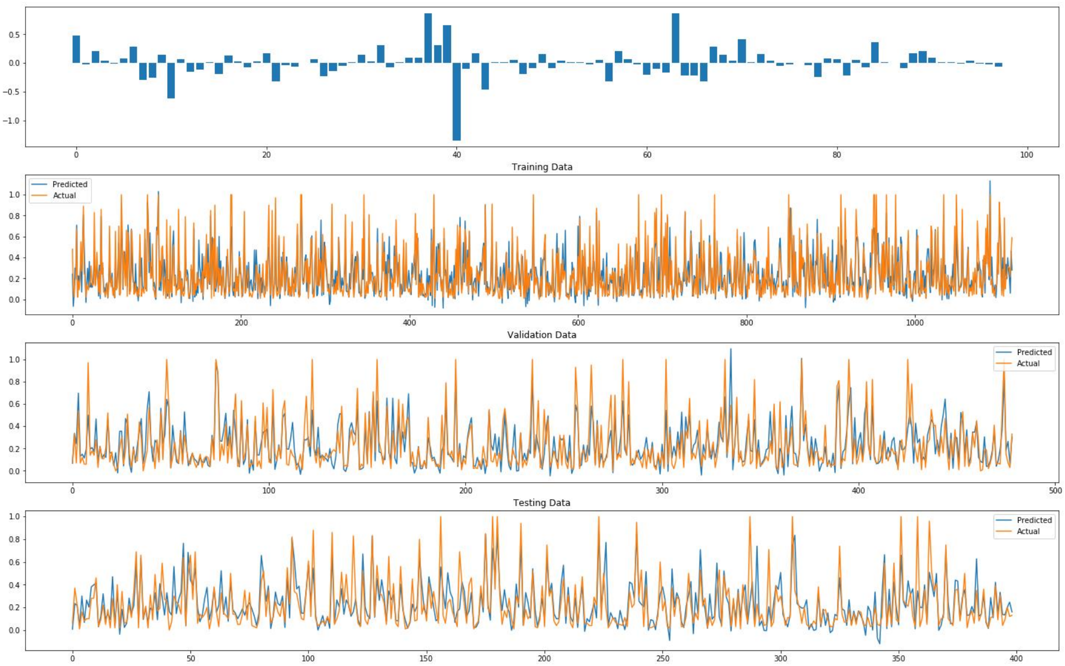
Base on the assignment requirement and steps we did before. We removed 23 items plus 4 non-analysis items that. however, we still left 1994 rows for further prediction.

After finalizing data sets, we separate the model into training, validation and testing set.80% for the training set and 20% will belong validation set. Inside the training set, we would further separate the 30% to the testing set for experimenting effectiveness of prediction

The first model we handle is linear, based on the QQ plot analysis, we know data is not ideal for diversity. We tested out the data separated with L1 and L2 models to find out the best alpha and lowest root mean square. After the process, we got the best result when we apply alpha to 0.1. And graph showing ideal prediction on criminal per capital with 0.145 RMSE.

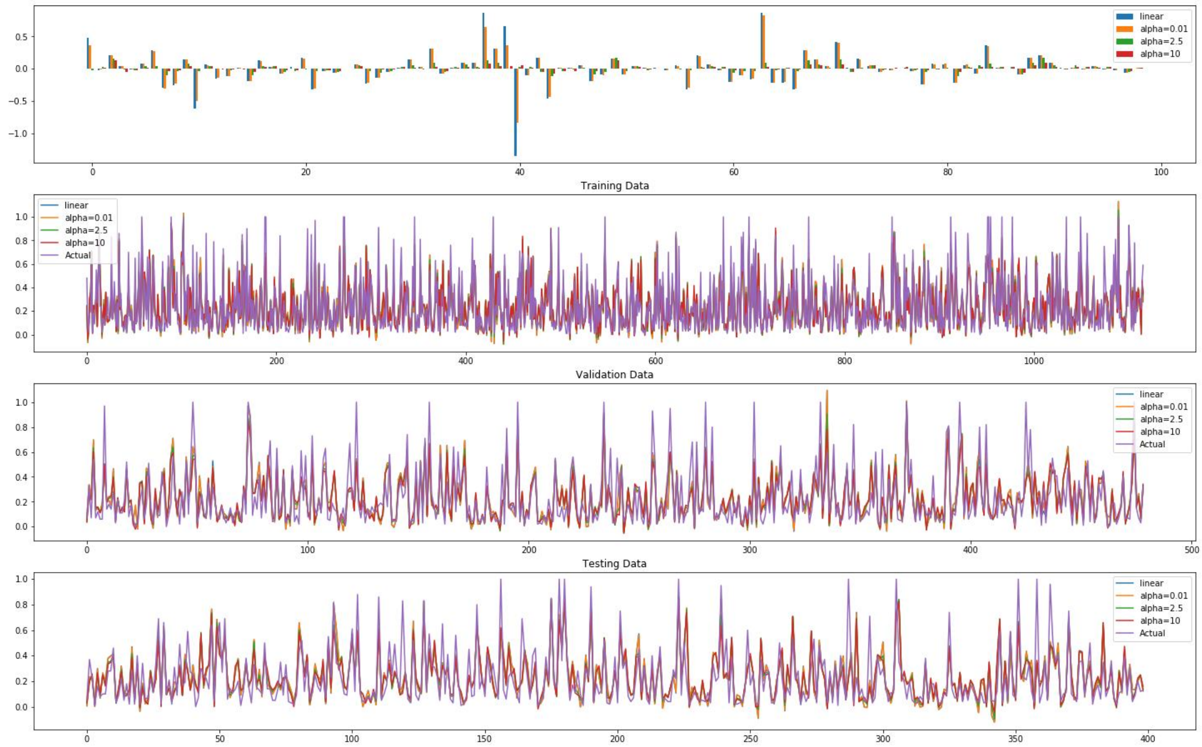
```
Console 1/A

L1_L2 = 0,  alpha = 4.484210526315789,  RMSE = 0.22212113288674284
L1_L2 = 0,  alpha = 4.742105263157895,  RMSE = 0.22343825775165155
L1_L2 = 0,  alpha = 5.0,  RMSE = 0.22468750826105321
L1_L2 = 1,  alpha = 0.1,  RMSE = 0.1450319807236902
L1_L2 = 1,  alpha = 0.35789473684210527,  RMSE = 0.16169320035583978
L1_L2 = 1,  alpha = 0.6157894736842106,  RMSE = 0.17287343607727929
L1_L2 = 1,  alpha = 0.8736842105263158,  RMSE = 0.18123910831581522
L1_L2 = 1,  alpha = 1.1315789473684212,  RMSE = 0.18780282602 86434
L1_L2 = 1,  alpha = 1.3894736842105266,  RMSE = 0.193118300986 0508
L1_L2 = 1,  alpha = 1.6473684210526318,  RMSE = 0.19753036017045572
L1_L2 = 1,  alpha = 1.905263157894737,  RMSE = 0.20126804564130615
L1_L2 = 1,  alpha = 2.1631578947368424,  RMSE = 0.20448993369163365
L1_L2 = 1,  alpha = 2.421052631578948,  RMSE = 0.20730926641931827
L1_L2 = 1,  alpha = 2.678947368421053,  RMSE = 0.20980895355555917
L1_L2 = 1,  alpha = 2.936842105263158,  RMSE = 0.21205097829726688
L1_L2 = 1,  alpha = 3.1947368421052635,  RMSE = 0.21408251635 4237
L1_L2 = 1,  alpha = 3.452631578947369,  RMSE = 0.21594003737 32471
L1_L2 = 1,  alpha = 3.710526315789474,  RMSE = 0.21765212410269646
L1_L2 = 1,  alpha = 3.9684210526315793,  RMSE = 0.21924145284244012
L1_L2 = 1,  alpha = 4.226315789473684,  RMSE = 0.22072621158069664
L1_L2 = 1,  alpha = 4.484210526315789,  RMSE = 0.22212113288674284
L1_L2 = 1,  alpha = 4.742105263157895,  RMSE = 0.22343825775165155
L1_L2 = 1,  alpha = 5.0,  RMSE = 0.22468750826105321

Best Model: L1_L2 = 0, alpha = 0.1, RMSE = 0.1450319807236902

IPython console  History
```





Lasso regression come after, we separately applying the 0.01, 0.1 and 0.5 alpha for model training. Turn out 0.5 not providing any foresight. The reason because the data set values all using tiny value therefore If affect prediction on a larger number. According to the graph we could observe, the alpha 0.01 given out the best result with only 0.15 root mean square of the testing set.

At the final, Ridge regression, we used 2.5,5 and 10 for an alpha, as we recognise the ridge regression with alpha 2.5 return the best result with only 0.141 RMSE.



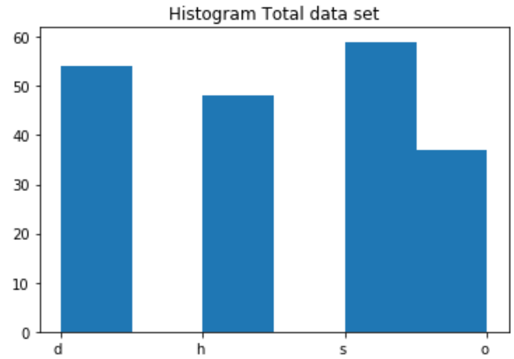
In conclusion with comparing three methods, we found ridge regression giving the most suitable prediction as it provided the lowest RMSE value.

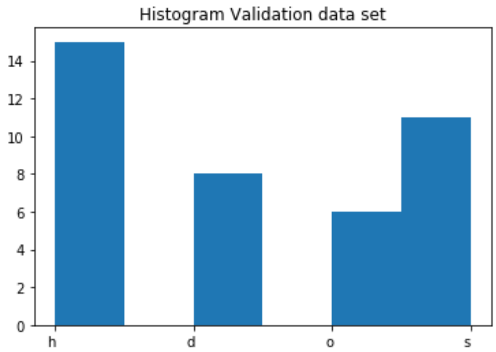
```
Valudation set :Lasso(alpha=0.01, copy_X=True, fit_intercept=False, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False),  RMSE = 0.15785435946844778

Testing set :Lasso(alpha=0.01, copy_X=True, fit_intercept=False, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False),  RMSE = 0.152667809696975

Valudation set :Lasso(alpha=0.1, copy_X=True, fit_intercept=False, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False),  RMSE = 0.272556398549236

Testing set :Lasso(alpha=0.1, copy_X=True, fit_intercept=False, max_iter=1000,
      normalize=False, positive=False, precompute=False, random_state=None,
      selection='cyclic', tol=0.0001, warm_start=False),  RMSE = 0.2631322306569677

Valudation set :Ridge(alpha=0.01, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.13534757904076267

Testing set :Ridge(alpha=0.01, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.14229946773334695

Valudation set :Ridge(alpha=2.5, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.13264021801031237

Testing set :Ridge(alpha=2.5, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.1410048304387525

Valudation set :Ridge(alpha=10, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.1347048521149847

Testing set :Ridge(alpha=10, copy_X=True, fit_intercept=False, max_iter=None,
      normalize=False, random_state=None, solver='auto', tol=0.001),  RMSE = 0.14177129787065124

In [2]:
```

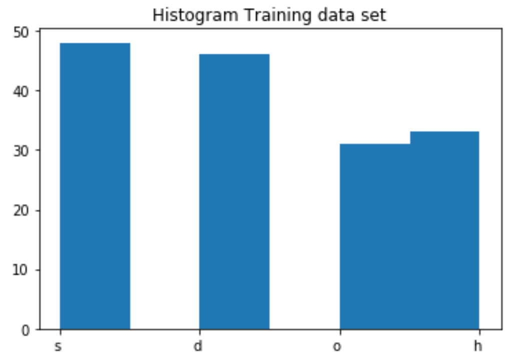
Question 2)

In the model that we see, we going to import forest data for getting data inside, first, we do the same action as the last question. We divided into training data, testing data and validation data. which we store testing in x_test,y_test and training in X, Y. We then separate 20 % into validation data.

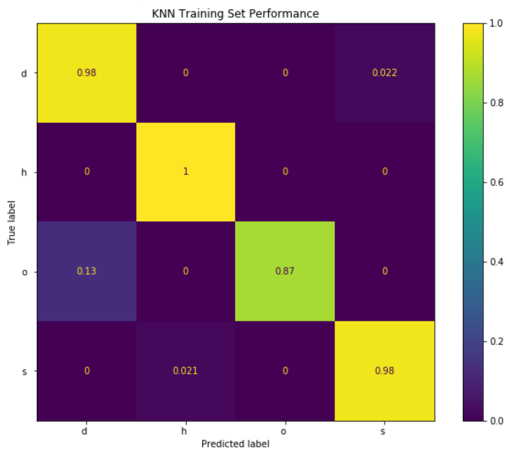
For the first two functions, we separate plot function into two which provided for validation model and test model. Those function return accuracy of prediction in the different prediction model

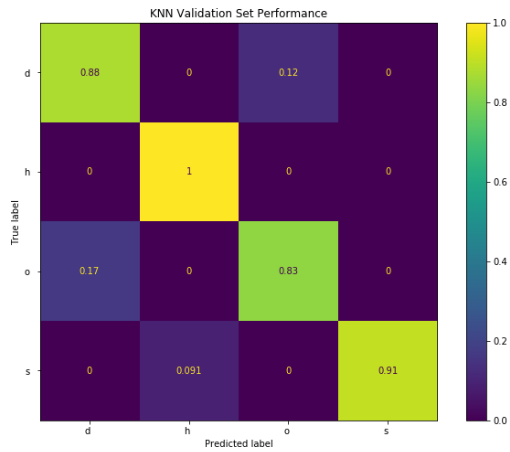


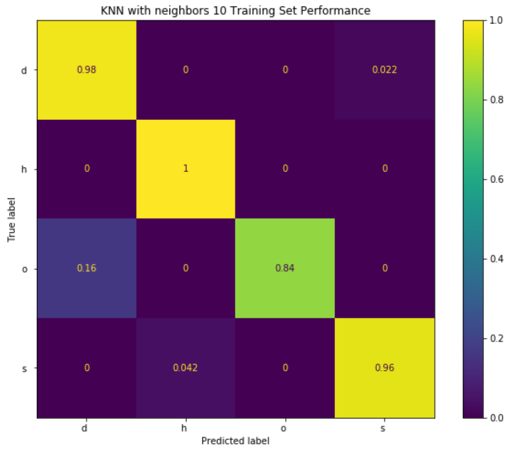


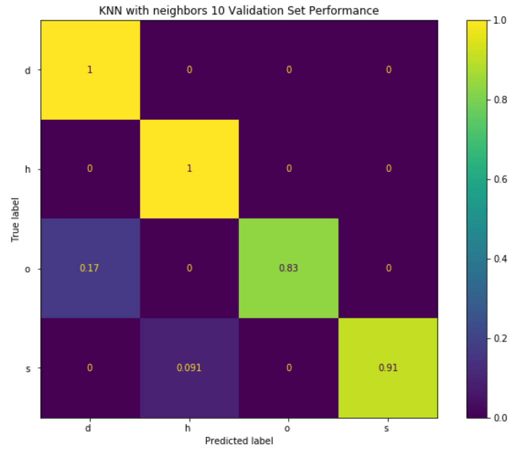


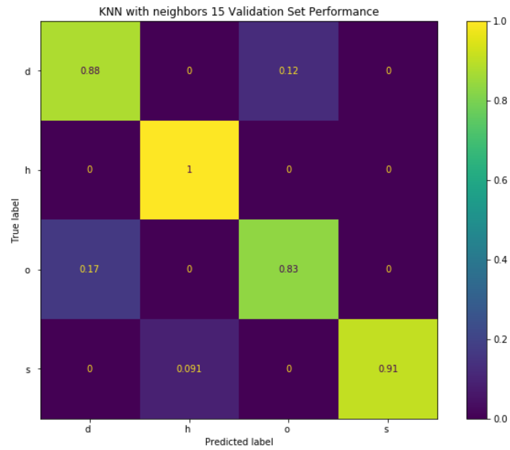
In KNNN model, we got to adjust model with a neighbour, which I give it to 5,10 and 15 all three predict a great accuracy in validation set but 10 neighbours are the best. Therefore, I put the 10 neighbours adding with distance for the optimizer, it turns out to have overfitted problems. Thus, will the keep the selection without distance optimizer

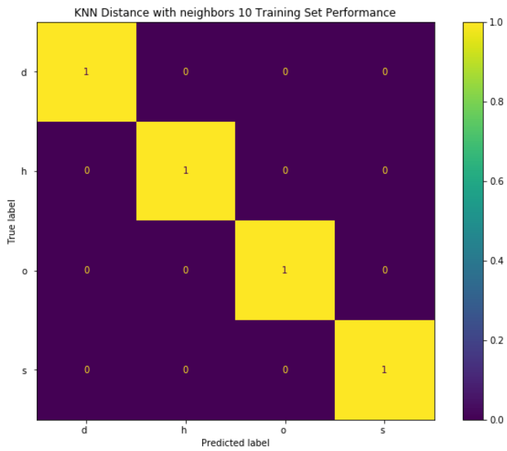


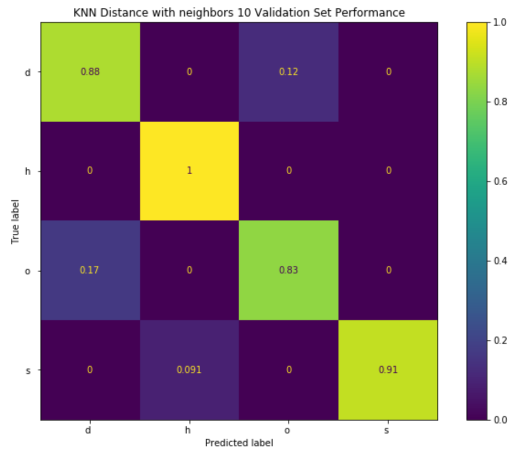




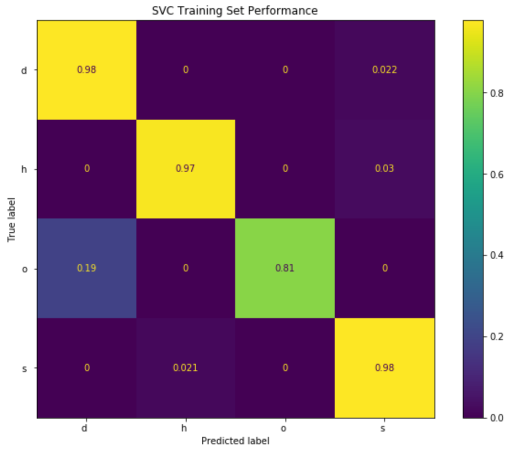


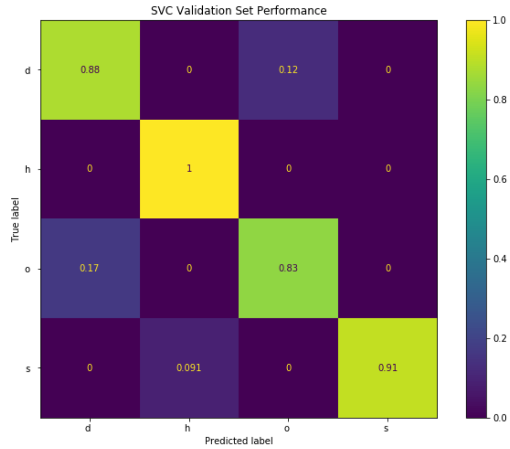


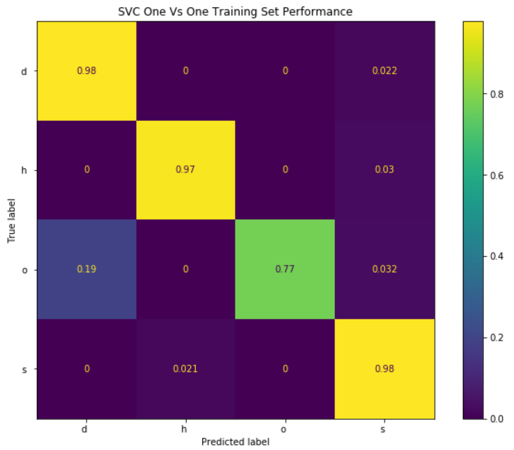


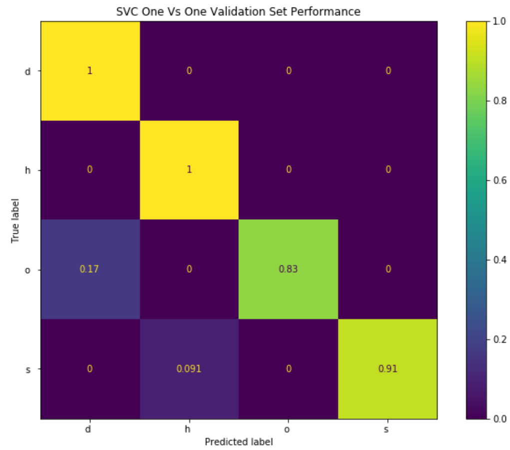


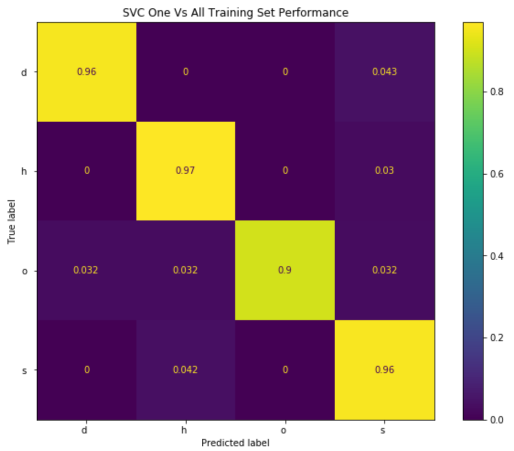
In Svc model, we use SVC only, SVC in one vs one and SVC in one vs all model to predict validation set. Turn out SVC one vs one has the best answer with 0.95 accuracies.

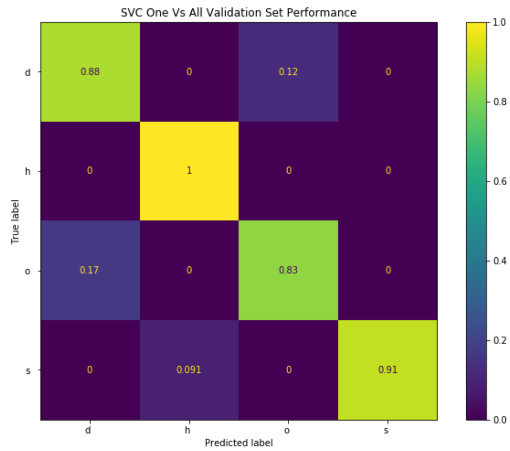




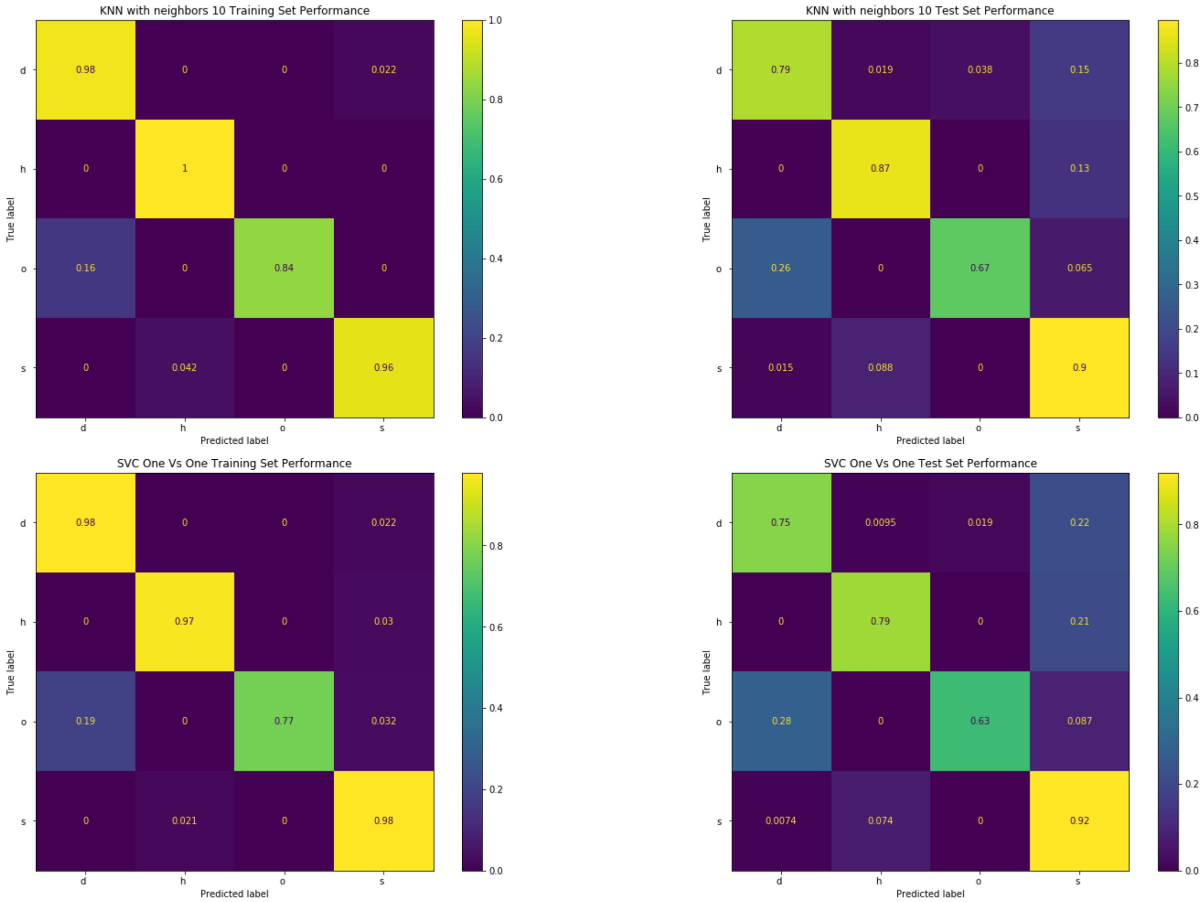


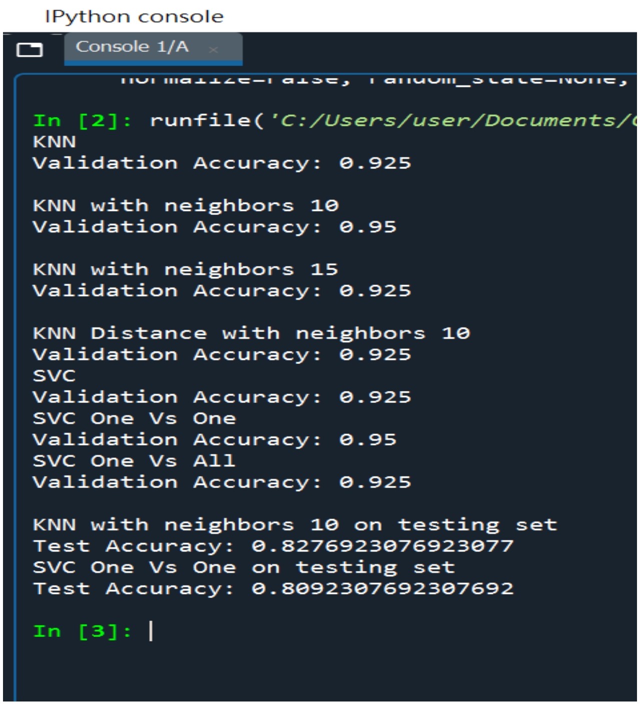






At last, Base on what we choose, we must compare the testing result on KNN with 10 neighbour and SVC one vs one model. The result shows the KNN model is slightly better on predict testing set with 2% improvement.




```
IPython console
Console 1/A
normalize=False, random_state=None,

In [2]: runfile('C:/Users/user/Documents/
KNN
Validation Accuracy: 0.925

KNN with neighbors 10
Validation Accuracy: 0.95

KNN with neighbors 15
Validation Accuracy: 0.925

KNN Distance with neighbors 10
Validation Accuracy: 0.925
SVC
Validation Accuracy: 0.925
SVC One Vs One
Validation Accuracy: 0.95
SVC One Vs All
Validation Accuracy: 0.925

KNN with neighbors 10 on testing set
Test Accuracy: 0.8276923076923077
SVC One Vs One on testing set
Test Accuracy: 0.8092307692307692

In [3]:
```

Question3)

For this task is how we use limited data to train and predict numbers from street house numbers. The training model was given 100 examples and others with 1000 examples which each number were separate in average.

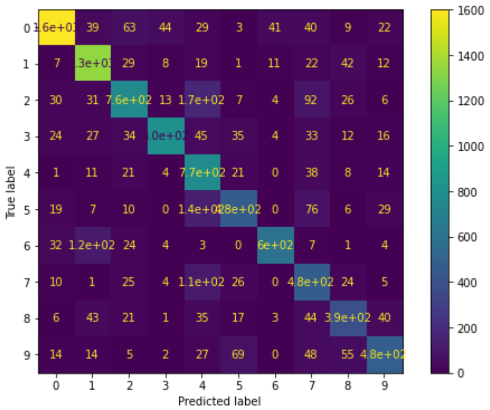
First requirement, we need training model and start build a model with non-data augmentation. According to the questions, first we need import data from mat files and transform it displayable pictures. What we have to do is reorder data set secuquence.1000,32,32,3 is the data set shape after we transform. For model training we got various size include 16 32 64 numbers for parameters and Adam optimizer would also be tested. Kernel size we would limit in 3 or 5 difference as final argument. For selection we listed above can provided the best prediction parameter and import them for training. Beside of that, our deep learning model individually using convert 2d twice for connect surround pixel. Then Max Pooling help with resize to the shape we want. As our observation, we notice photo have not much unnecessary data and in low pixel. Therefore, drop is unnecessary for us to add. Training accuracy reached over 90%. while testing accuracy also reach 80%. Provided that prediction are effective by model set 64 kernel size 5 with ADAM optimize.

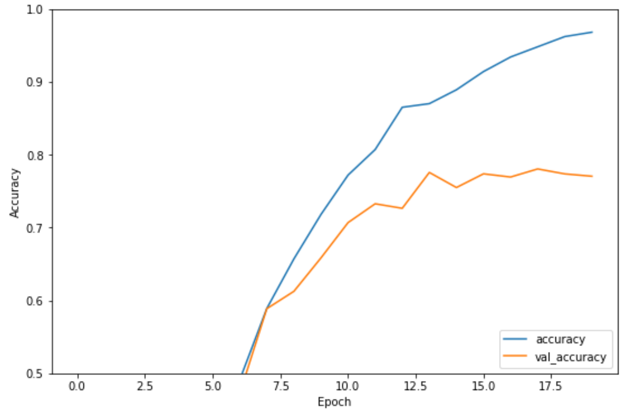
```
(1000, 32, 32, 3)
[[[0.5803922  0.5647059  0.5686275 ]
  [0.5764706  0.56078434 0.5647059 ]
  [0.5647059  0.54509807 0.56078434]
  ...
  [0.56078434 0.52156866 0.5254902 ]
  [0.56078434 0.52156866 0.5176471 ]
  [0.56078434 0.52156866 0.5176471 ]]

 [[0.5921569  0.5686275  0.5764706 ]
  [0.5882353  0.5647059  0.57254905]
  [0.5686275  0.54901963 0.5647059 ]
  ...
  [0.56078434 0.52156866 0.5254902 ]
  [0.5568628  0.5176471  0.5137255 ]
  [0.5568628  0.5176471  0.50980395]]

 [[0.59607846 0.57254905 0.5803922 ]
  [0.5921569  0.5686275  0.58431375]
  [0.57254905 0.5529412  0.5764706 ]
  ...
  [0.5058824  0.47058824 0.49019608]
  [0.5137255  0.47843137 0.48235294]
  [0.5176471  0.4862745  0.47843137]]

 ...

 [[0.5254902  0.5254902  0.53333336]
  [0.5176471  0.52156866 0.5372549 ]
  [0.50980395 0.5137255  0.5294118 ]
  ...
  [0.5058824  0.48235294 0.49019608]
  [0.52156866 0.49803922 0.49803922]
  [0.54509807 0.50980395 0.5137255 ]]
```

```
32/32 [==============================] - 0s 3ms/step - loss: 1.6639 - accuracy: 0.4270
313/313 [==============================] - 1s 2ms/step - loss: 1.7807 - accuracy: 0.3698
--- Starting trial: run-1
{'num_units': 16, 'kernel_size': 3, 'optimizer': 'sgd'}
Epoch 1/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3838 - accuracy: 0.1110
Epoch 2/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3544 - accuracy: 0.1180
Epoch 3/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3164 - accuracy: 0.1510
Epoch 4/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2926 - accuracy: 0.1790
Epoch 5/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2829 - accuracy: 0.1810
Epoch 6/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2687 - accuracy: 0.1770
Epoch 7/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2646 - accuracy: 0.1690
Epoch 8/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2569 - accuracy: 0.1820
Epoch 9/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2522 - accuracy: 0.1870
Epoch 10/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2487 - accuracy: 0.1750
Epoch 11/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2452 - accuracy: 0.1760
Epoch 12/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2397 - accuracy: 0.1730
Epoch 13/20
32/32 [==============================] - 0s 4ms/step - loss: 2.2443 - accuracy: 0.1800
Epoch 14/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2347 - accuracy: 0.1740
Epoch 15/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2428 - accuracy: 0.1630
```

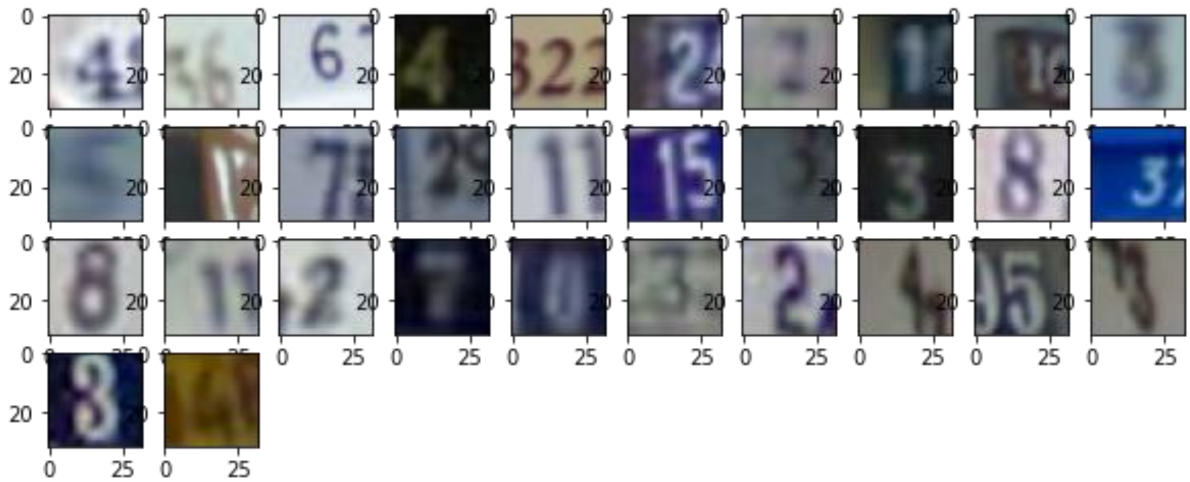
```
--- Starting trial: run-2
{'num_units': 16, 'kernel_size': 5, 'optimizer': 'adam'}
Epoch 1/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3519 - accuracy: 0.1660
Epoch 2/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3038 - accuracy: 0.1750
Epoch 3/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2783 - accuracy: 0.1750
Epoch 4/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2532 - accuracy: 0.1730
Epoch 5/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2503 - accuracy: 0.1830
Epoch 6/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2478 - accuracy: 0.1750
Epoch 7/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2337 - accuracy: 0.1880
Epoch 8/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2105 - accuracy: 0.1820
Epoch 9/20
32/32 [==============================] - 0s 3ms/step - loss: 2.1823 - accuracy: 0.2290
Epoch 10/20
32/32 [==============================] - 0s 3ms/step - loss: 2.1292 - accuracy: 0.2350
Epoch 11/20
32/32 [==============================] - 0s 3ms/step - loss: 2.0796 - accuracy: 0.2690
Epoch 12/20
32/32 [==============================] - 0s 3ms/step - loss: 1.9841 - accuracy: 0.3110
Epoch 13/20
32/32 [==============================] - 0s 3ms/step - loss: 1.8679 - accuracy: 0.3440
Epoch 14/20
32/32 [==============================] - 0s 3ms/step - loss: 1.6785 - accuracy: 0.4280
Epoch 15/20
32/32 [==============================] - 0s 3ms/step - loss: 1.5129 - accuracy: 0.5190
Epoch 16/20
```

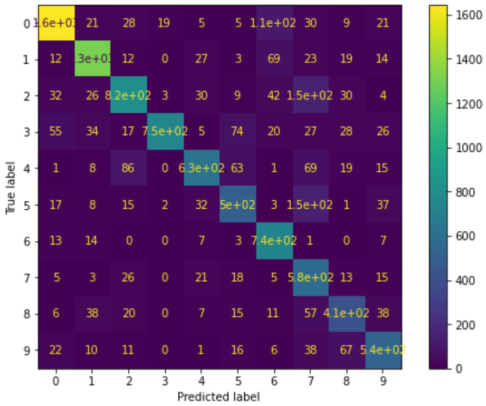
```
--- Starting trial: run-11
{'num_units': 64, 'kernel_size': 5, 'optimizer': 'sgd'}
Epoch 1/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3743 - accuracy: 0.1360
Epoch 2/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3310 - accuracy: 0.1770
Epoch 3/20
32/32 [==============================] - 0s 3ms/step - loss: 2.3016 - accuracy: 0.1770
Epoch 4/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2852 - accuracy: 0.1770
Epoch 5/20
32/32 [==============================] - 0s 4ms/step - loss: 2.2745 - accuracy: 0.1630
Epoch 6/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2555 - accuracy: 0.1740
Epoch 7/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2538 - accuracy: 0.1790
Epoch 8/20
32/32 [==============================] - 0s 4ms/step - loss: 2.2472 - accuracy: 0.1770
Epoch 9/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2426 - accuracy: 0.1750
Epoch 10/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2372 - accuracy: 0.1800
Epoch 11/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2340 - accuracy: 0.1940
Epoch 12/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2294 - accuracy: 0.1790
Epoch 13/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2367 - accuracy: 0.2110
Epoch 14/20
32/32 [==============================] - 0s 3ms/step - loss: 2.2253 - accuracy: 0.1930
```

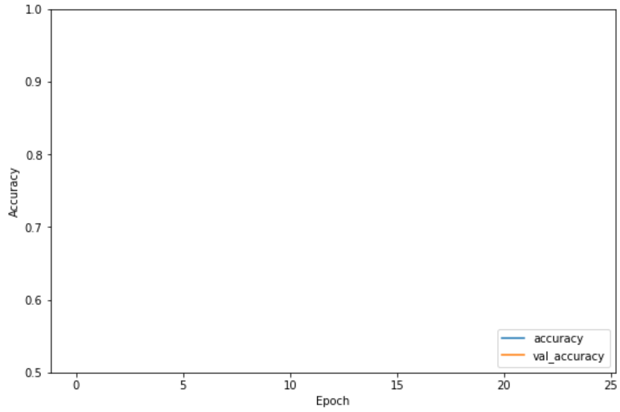




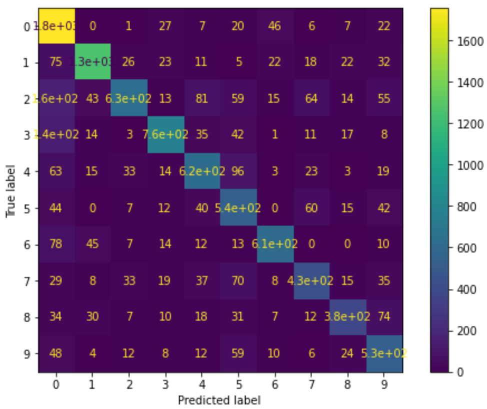
Next part, we going to use augmentation for training. By generate graph with different angle, size and zoom ratio by little between range in 5%. Below is the accuracy after augmentation. As we see graph cannot show info properly, but confusion graph provided a summarize that each number class prediction is in high percentage.

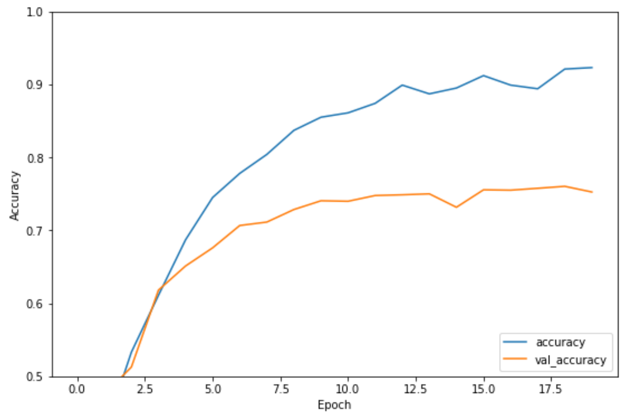




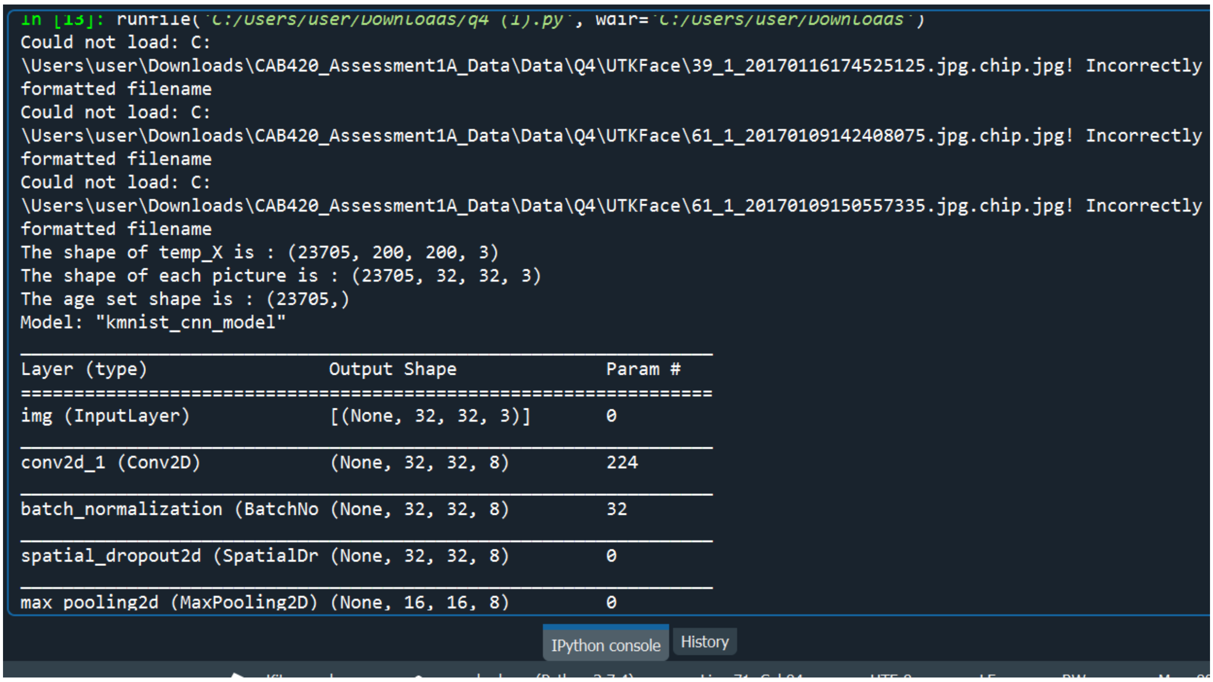


Last part with requirements in fine tuning. For training this model, we would reference lesson resource which is VGG CIFAP small. As data set provided the size of each image is 32x32, Kmnist and mnist need size with 28x28.For simple design guideline. We prefer to use the same input size CIFAP for fine tuning. Small version is enough for training as we only have limited data for resource. Overall model has surprising 90 % test accuracy and 70% validation accuracy.

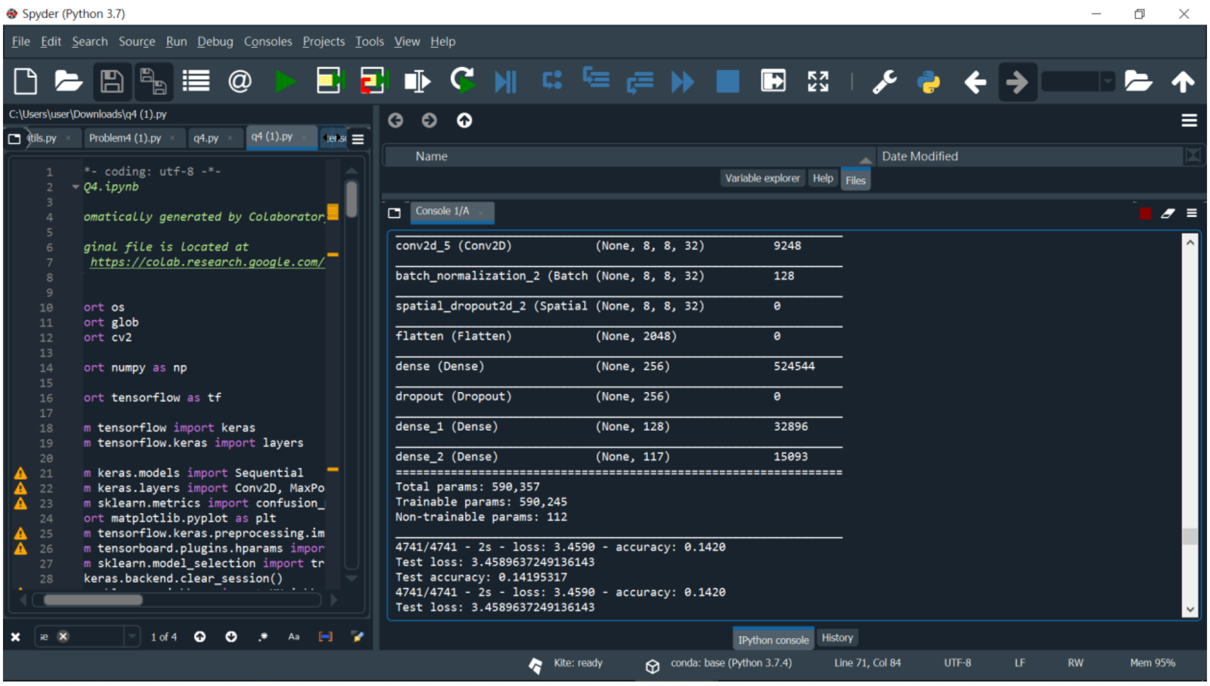




Question 4) we analysis a huge amount of human image and training it with two methods, the first one using training, validation and testing set for evaluating. Import the UTKfile and import each domain with age, gender, race and image into data. Review each picture and check where they could correct display and reshape from 200X200 to 32x32.This purpose speed up image processing. At the end, we display graph shape how long with the age data set length.



Now we could divide data into 3 parts and start to build a model. The model this time will be used with 3 convert 2d to expand the data information and randomly drop out unnecessary one to prevent overfitting, then we flatten the image and dense them into 256 and output the with dense 117.After building the model, we put our data to fit in and evaluate model accuracy.



Result show, using the first method have low with only 14% prediction rate. Which means this method is difficult to analyse how old the human in the pictures is.

Next, Cross-validation data are the next method. Similar to the last question it is necessary to import data first, however, based on the race to classify data is the additional step for pre-training. We resize the image and turn it to float 32. For the last step, we put each cross-validation data set which if race equal to 0 then cross-validation will be given those left for training into the model. Calculate each testing set accuracy. Overall, 60 % testing set have a correct prediction. Far better than the first method

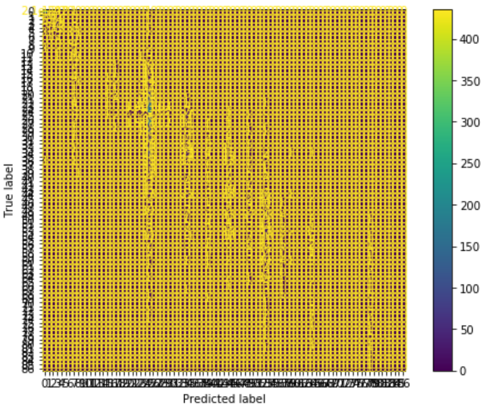
Console 1/A

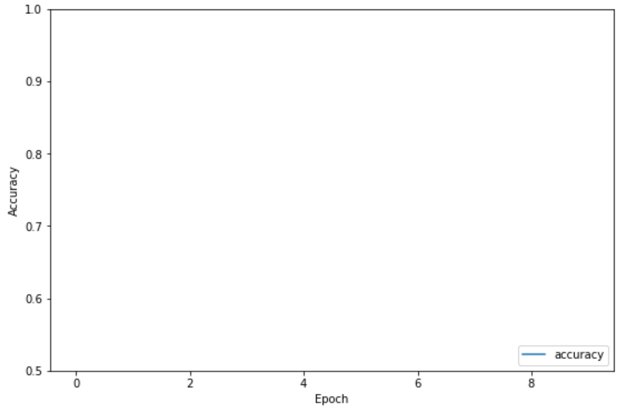
```
Test accuracy: 0.14195317
(10078, 32, 32, 3)
(4526, 32, 32, 3)
(3434, 32, 32, 3)
(3975, 32, 32, 3)
(1692, 32, 32, 3)
(13627, 32, 32, 3)
(19179, 32, 32, 3)
(20271, 32, 32, 3)
(19730, 32, 32, 3)
(22013, 32, 32, 3)
history_0 done
history_1 done
history_2 done
history_3 done
history_4 done
10078/10078 - 5s - loss: 3.2809 - accuracy: 0.1368
Test loss: 3.2809216940679775
Test accuracy: 0.1368327
4526/4526 - 2s - loss: 3.2013 - accuracy: 0.1608
Test loss: 3.2013185417235976
Test accuracy: 0.16084842
3434/3434 - 1s - loss: 2.5948 - accuracy: 0.2533
Test loss: 2.594809595206899
Test accuracy: 0.25334886
```

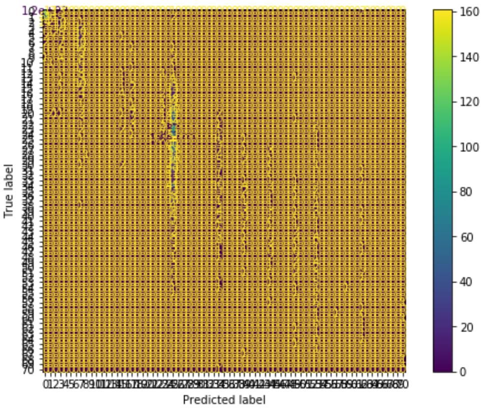
IPython console History

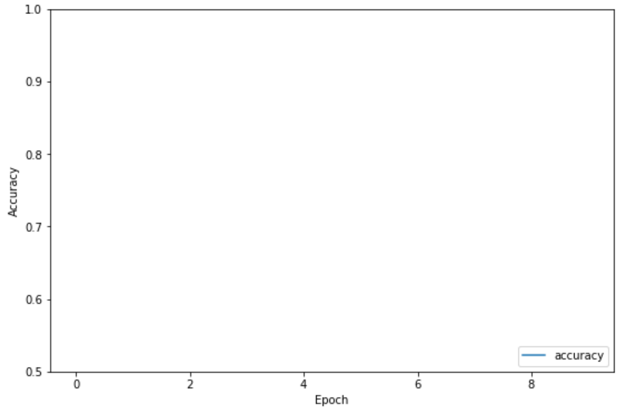
Kite: ready conda: base (Python 3.7.4) Line 71, Col 84 UTF-8 LF RW Mem 94%

```
(22013, 32, 32, 3)
history_0 done
history_1 done
history_2 done
history_3 done
history_4 done
10078/10078 - 5s - loss: 3.2809 - accuracy: 0.1368
Test loss: 3.2809216940679775
Test accuracy: 0.1368327
4526/4526 - 2s - loss: 3.2013 - accuracy: 0.1608
Test loss: 3.2013185417235976
Test accuracy: 0.16084842
3434/3434 - 1s - loss: 2.5948 - accuracy: 0.2533
Test loss: 2.594809595206899
Test accuracy: 0.25334886
3975/3975 - 2s - loss: 2.9285 - accuracy: 0.2174
Test loss: 2.9285113103134828
Test accuracy: 0.21735848
1692/1692 - 1s - loss: 2.8231 - accuracy: 0.2134
Test loss: 2.8231303342408887
Test accuracy: 0.21335697
Scores from each Iteration:  [0.1368327, 0.16084842, 0.25334886, 0.21735848, 0.21335697]
Average K-Fold Score : 0.19634908

In [14]: runfile('C:/Users/user/Downloads/q4 (1).py', wdir='C:/Users/user/Downloads')
```


True label
Predicted label
0
50
100
150
200
250
300
350
400


Accuracy
Epoch
accuracy
0.5
0.6
0.7
0.8
0.9
1.0
0
2
4
6
8


True label
Predicted label
0
20
40
60
80
100
120
140
160


Accuracy
Epoch
accuracy
0.5
0.6
0.7
0.8
0.9
1.0
0
2
4
6
8