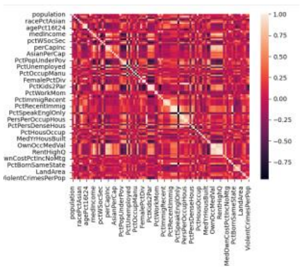
Hajun Song Assessment 1A

N9894403

Problem 1 Regression

Pre-processing performance justification

Before going further into regression, following figure1 demonstrates the correlation between variables.



Following *figure 1,* demonstrates correlation between variables however, since there are too many variables, this results not able to see individual relationships clearly but able to visually identify groups of correlated variables.

Standardising the data set allows to make a significant difference regarding to the performance and ease of visualisation. Helper function "*def standardise(data):* "allows to compute the mean and standard deviation on the training data and use these to standardise the validation and testing sets. Importance of standardisation/normalization can be demonstrated on further discussions.

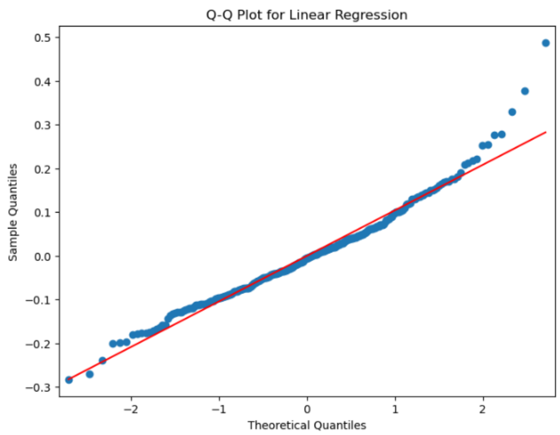
Details of the three trained models.

Linear, Ridge and Lasso regression models are used for supervised learning tasks. Linear regression tries to find a linear relationship between input variables and output variables and its objective is to minimize the sum of squared errors between the predicted values and the actual values. Ridge regression is a regularized linear regression model that has penalty terms to prevent over-fitting. Penalty term is equivalent to the sum of squared of the coefficients. Regularization parameter lambda controls the strength of the penalty meaning larger lambda, more coefficients will shrink towards zero. Lasso regression is another type of regularized linear regression model that has similar characteristics as ridge regression. Lambda is equivalent to the absolute value of the coefficients. Having optimal lambda choice can be done when value of lambda minimizes the RMSE on the validation set.

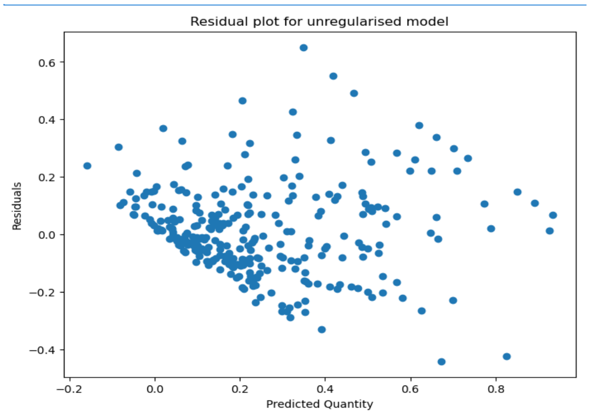
An evaluation comparing the three models.

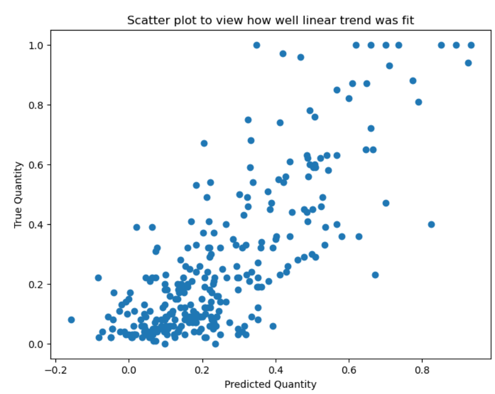
Below figure 2 demonstrates Q-Q plot for linear regression model of unregularized data.

Linear Model Validation Data: RMSE = 0.15491210066258207 - validation
Linear Model Validation Data: RMSE = 0.291599337438958. - Training



This Q-Q plot indicates issues at the end. Plot indicates that there is non-linear trend in this data or non-constant variance. This can be visualized by looking at residual plot for unregularized plot. Residual plot indicates that there are significant numbers of outliers therefore it is breaking the non-linear trend assumption. Spread of the residuals is non-constant, indicating there is non-constant variance. If predicted quantity and true quantity are plotted, the majority of the true quantities are plotted at zero, perhaps we can assume that there was some sort of hardware/socio-economic failure. Therefore, there is no need to predict zero values. However, since there are too many variables representing zero, this will remain as data as hopefully represents as some meaningful data.





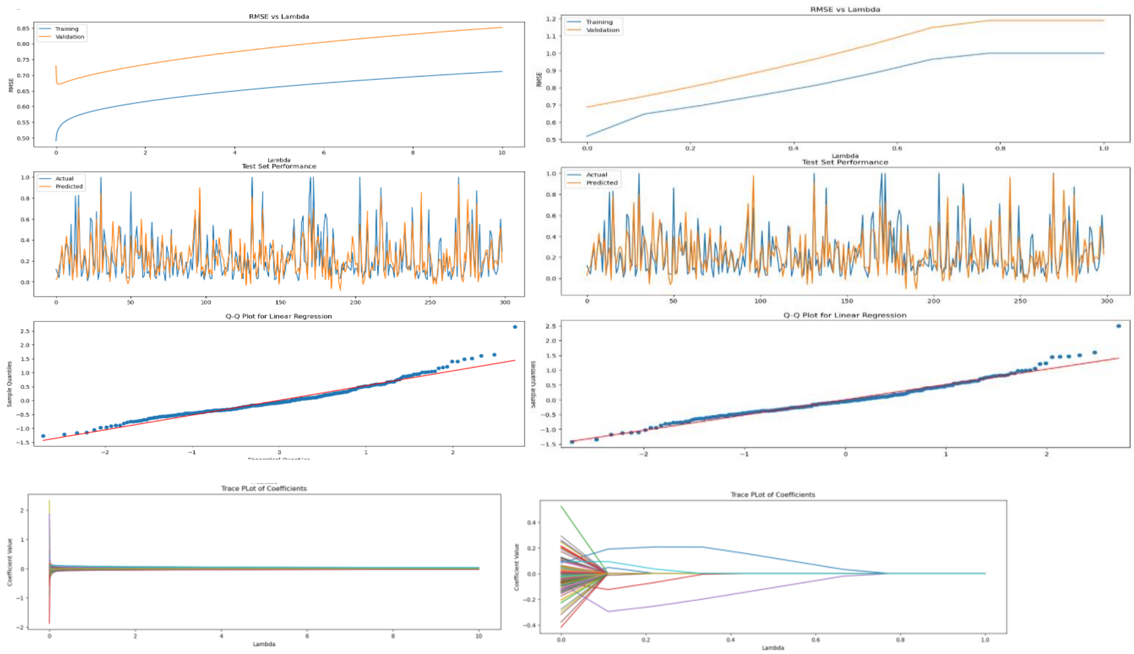
Before Ridge/Lasso regression, data has been standardized which allows the outliers will remains as outliers, point that is not outlined before standardization will remain an outlier afterwards.

For Ridge Regression

```
Best R Squared = 0.7185647427447908
Best Adjusted = 0.5757042060670196
Best RMSE (val) = 0.1553748181777985
Best RMSE (test) = 0.15560311713881106
Best coefficients on the normalised model
Best slope = [[ 0.00654498]
```

For Lasso Regression

```
Best R Squared = 0.7320517436259425
Best Adjusted = 0.5960374002888575
Best RMSE (val) = 0.16616256537828702
Best RMSE (test) = 0.16882789195092898
Best coefficients on the normalised model
Best slope = [[ 0.0951327 ]
```



For ridge, RMSE vs lambda indicates that when there is little bit of regularization, there is noticeable improvement in performance then starts to over regularize. Q –Q plot and residuals are bad as previous model. As lambda increased, coefficients gradually tend towards 0 but it does not. Best RMSE on validation/training value looks much more generalized than before.

For LASSO, RMSE vs lambda indicates that as soon as regularization starts, results to over regularize. Q-Q plot and residuals looks same as other models but in terms of coefficients, it indicates that some coefficients values go 0. However, the best RMSE on validation/training value are much more generalized than other models. Also, R squared and best adjusted of LASSO has better value thus, this indicates that LASSO model has higher performance in terms of accuracy.

Ethical Concerns

1. Bias and Fairness – Model should not discriminate against certain groups (race, sex, ethnicity, gender, age, religion)

2. Privacy – Collection of data to train the model should be used in a manner that respects the privacy of individuals.

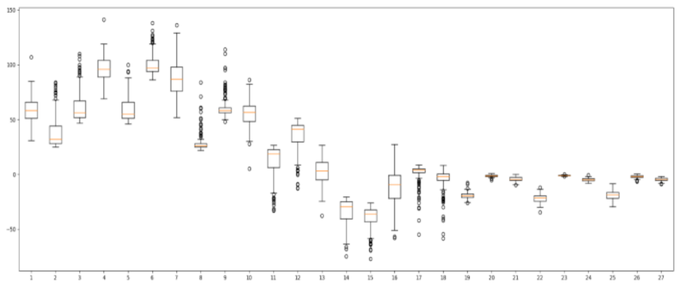
3. Transparency and Accountability - Model needs to be transparent and accountable.

4. Legal Enforcement – It is crucial to consider how trained model might affect the law enforcement agencies.

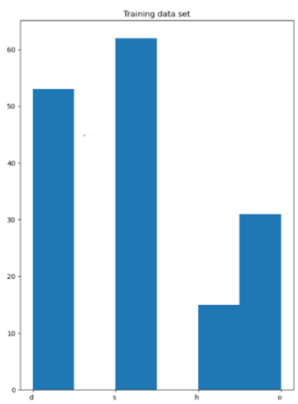
Problem 2: Classification

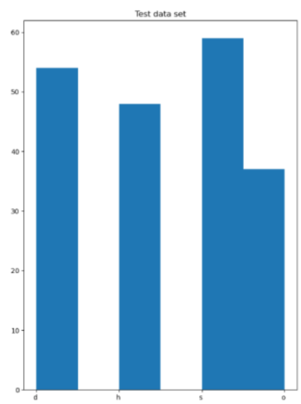
Pre-processing performance justification

Checking if there is any scale issues for the data. Below box-plot figure demonstrates the scale of the data.



This box plot indicates there's not significantly imbalance dimension at the moment. However, standardizing the data was done and resulted in not much difference. Standardization did not make much impact on the data,



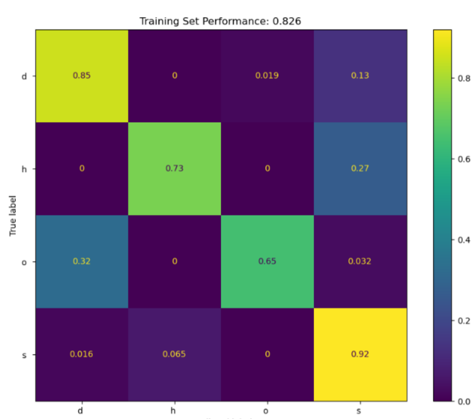


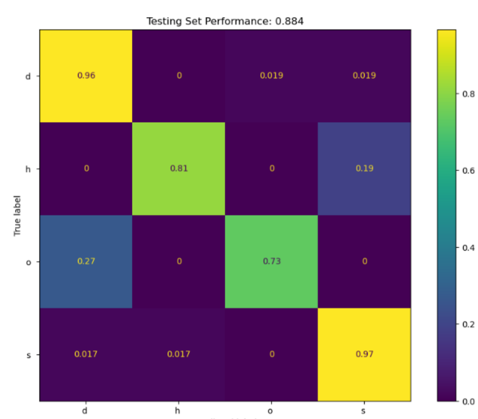
It is important to notice that there is class imbalance, where there D and S has similar distribution set and where H,E does not.

Details of hyper-parameters selection method, discussion in relation to characteristics of the data.

CKNN - When value of n_neighbours was set as 20 (88%), it is indicating that there are some sort of class imbalance as seen below however still shows decent accuracy. This requires to reduce the n_neighbours parameter to increase the chance of being able to get these classes correct resulting increase in accuracy as well.

```
              precision    recall  f1-score   support

           d       0.83      0.96      0.89        54
           h       0.97      0.81      0.89        48
           o       0.96      0.73      0.83        37
           s       0.85      0.97      0.90        59

    accuracy                           0.88       198
   macro avg       0.90      0.87      0.88       198
weighted avg       0.90      0.88      0.88       198
```



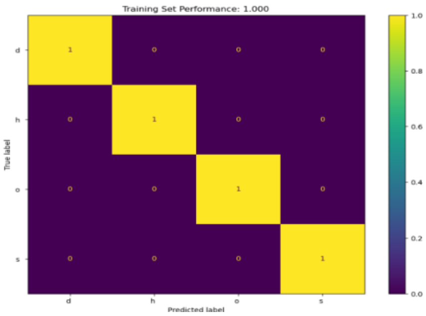


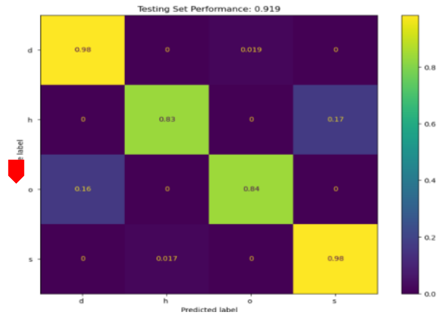
Finding the optimal n_neighbours value can be done using for loops with setting K value parameters of "values_of_k = [1, 2, 4, 8, 16, 32, 64, 128] "

This allows to test each parameters of the k value to n_neighbours that results best accuracy and training/testing set performance. Optimal K value resulted as

	precision	recall	f1-score	support
d	0.90	0.98	0.94	54
h	0.98	0.83	0.90	48
o	0.97	0.84	0.90	37
s	[illegible].88	0.98	0.93	59
accuracy			0.92	198
macro avg	0.93	0.91	0.92	198
weighted avg	0.92	0.92	0.92	198

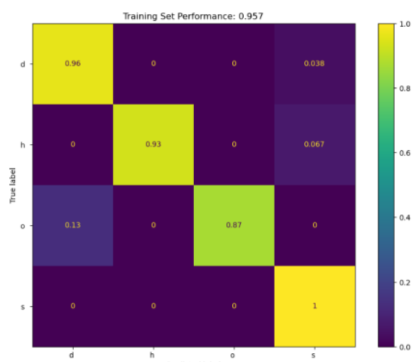
with

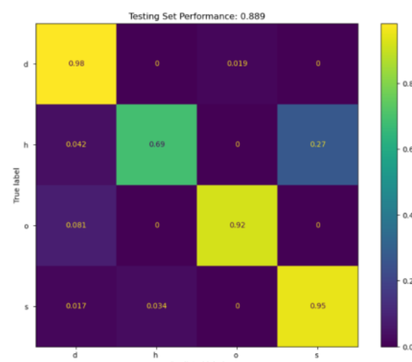




Random Forest – having initially having paramenter of n_estimators =100 and max_depth =4

	precision	recall	f1-score	support
d	0.90	0.98	0.94	54
h	0.94	0.69	0.80	48
o	0.97	0.92	0.94	37
s	0.[illegible]	0.95	0.87	59
accuracy			0.89	198
macro avg	0.91	0.88	0.89	198
weighted avg	0.90	0.89	0.89	198

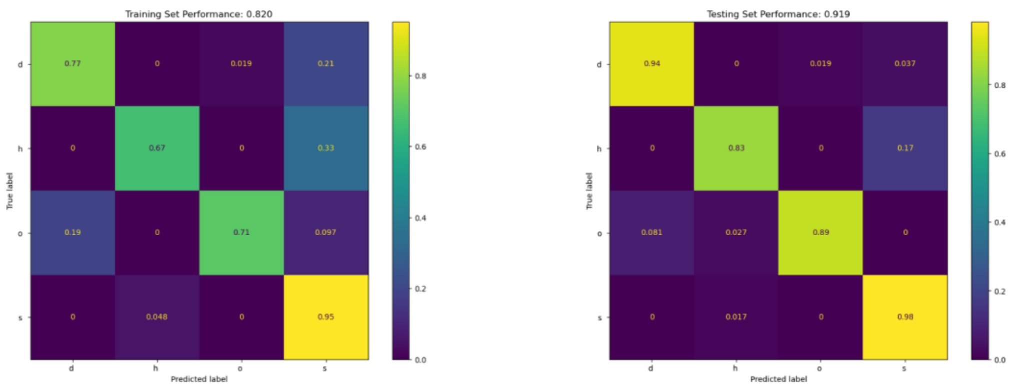




Resulting 89% accuracy. This model can be improved by going in deeper allowing to make more decisions, allowing to find more classes by having fine grained. Macro Average went down . It is hard to find where which n_estimators or max_depth where it will find the best solution. In this case, GridSearchCV is used with certain parameters to find optimal output. By performing gridsearch, best parameters was maxdepth of 4 and n_estimators of 100.

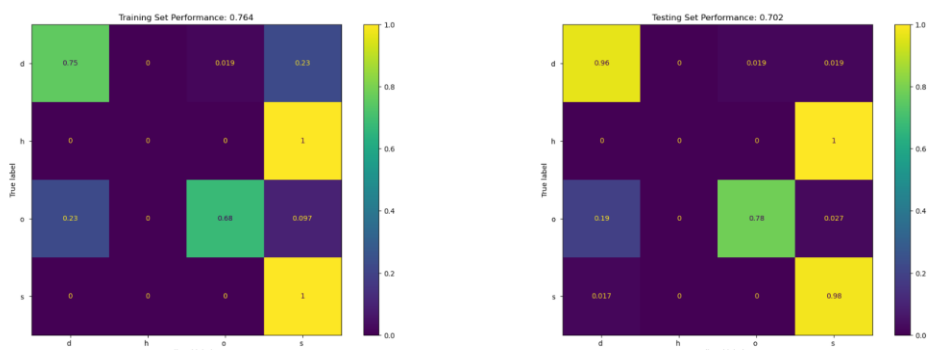
SVM – The sklearn has SVM classes will automatically extend 1 v 1 encoding when it get's shown in multi-class data.

	precision	recall	f1-score	support
d	0.94	0.94	0.94	54
h	0.95	0.83	0.89	48
o	0.97	0.89	0.93	37
s	0.85	0.98	0.91	59
accuracy			0.92	198
macro avg	0.93	0.91	0.92	198
weighted avg	0.92	0.92	0.92	198



Changing the hyper parameter 'class_weight' as 'balanced' make huge difference compared when it is not. See below figure that demonstrates poor macro average with poor fit.

	precision	recall	f1-score	support
d	0.87	0.96	0.91	54
h	0.00	0.00	0.00	48
o	0.97	0.78	0.87	37
s	0.54	0.98	0.69	59
accuracy			0.70	198
macro avg	0.59	0.68	0.62	198
weighted avg	0.58	0.70	0.62	198



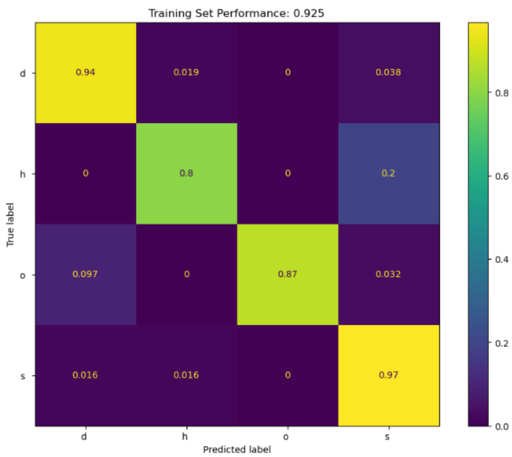
Comparison of these three modles can be done by looking at the precision score of each model.

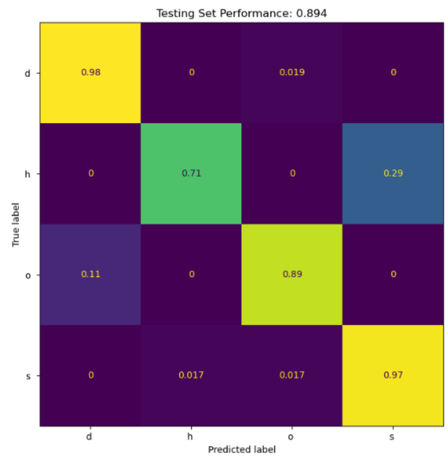
Parameter used for SVM is

```
param_grid = [
  {'C': [0.1, 1, 10, 100], 'kernel': ['linear']},
  {'C': [0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001], 'kernel': ['rbf']},
  {'C': [0.1, 1, 10, 100], 'degree': [3, 4, 5], 'kernel': ['poly']},
 ]
svm = SVC(class_weight='balanced')
```

Many different values of C, with three different kernels, trying some parameters gama as kernel scale and polynomial degree, and these parameters will result best rank score. It turns out that best

```
{'C': 100, 'degree': 3, 'kernel': 'poly'}
              precision    recall  f1-score   support

           d       0.93      0.98      0.95        54
           h       0.97      0.71      0.82        48
           o       0.94      0.89      0.92        37
           s       0.80      0.97      0.88        59

    accuracy                           0.89       198
   macro avg       0.91      0.89      0.89       198
weighted avg       0.90      0.89      0.89       198
```





1. CKNN Model :

	precision	recall	f1-score	support
d	0.90	0.98	0.94	54
h	0.98	0.83	0.90	48
o	0.97	0.84	0.90	37
s	0.88	0.98	0.93	59
accuracy			0.92	198
macro avg	0.93	0.91	0.92	198
weighted avg	0.92	0.92	0.92	198

2. Random Forest:

	precision	recall	f1-score	support
d	0.88	0.98	0.93	54
h	0.94	0.69	0.80	48
o	0.97	0.92	0.94	37
s	0.81	0.93	0.87	59
accuracy			0.88	198
macro avg	0.90	0.88	0.88	198
weighted avg	0.89	0.88	0.88	198

3. SVM:

{'C': 100, 'degree': 3, 'kernel': 'poly'}

	precision	recall	f1-score	support
d	0.93	0.98	0.95	54
h	0.97	0.71	0.82	48
o	0.94	0.89	0.92	37
s	0.80	0.97	0.88	59
accuracy			0.89	198
macro avg	0.91	0.89	0.89	198
weighted avg	0.90	0.89	0.89	198

Based on these evaluation metrices, it indicates that CKNN has the best overall performance with an F1-score of 0.92. Random Forest has an F1-score of 0.89, and SVM has an F1-score of 0.88. This suggests that CKNN has the highest accuracy in predicting the crime categories compared to the other two models. However, it's worth noting that the difference in performance between the models is relatively small. Additionally, it's important to consider other factors, such as computational efficiency and interpretability, when choosing the final mode

Problem 3: Training and Adapting Deep Networks

The network structure of VGG is set to an RGB image of 32 x 32 x 3 image. Average RGB value is calcuated for all images on the training set image. Image then returns to input as an input to VGG convolution network. A 3 x 3 filter is used in this network. The first set has 8 filters, second set has 16 filters and third set has 32 filters. Each set is followed by batch normalization layer and RELU activation function and then max pooling layers.

Dense layer : After final max pooling layer, there is dense layers with 512 neurons and batch normalization layer followed by activation RELU function.

Output Layer: The output layer consists of a dense layer with 10 neurons and a softmax activation function.

By looking at this, the VGG architecture model has a total of 6 convolutional layers, 3 max pooling layers, 1 dense layer, and 1 output layer. The use of batch normalization can help to improve the stability and speed of training, and the small filter sizes in the convolutional layers can help to capture fine-grained features in the input images.

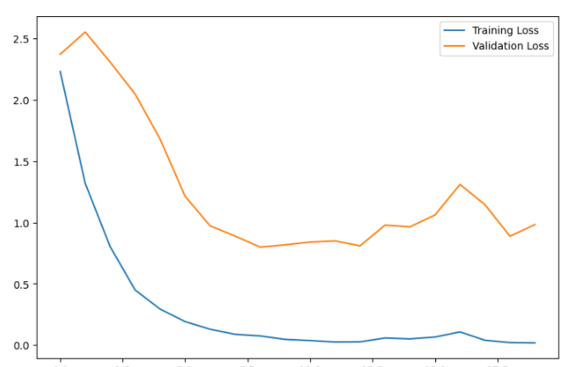
Typically, training a neural network involves iterating over the training dataset multiple times, also known as epochs. During each epoch, the network adjusts its parameters based on the error or loss between the predicted outputs and the actual outputs. The number of epochs can vary depending on the desired accuracy and the training time available. Since there is a very limited training sample (1000), the model will struggle to generalize well for the unsee data. With unbalanced data, the model will likely overfit resulting performing well on the training sample but poor on the testing data. Data augmentation will allow to reduce the overfitting of the model.

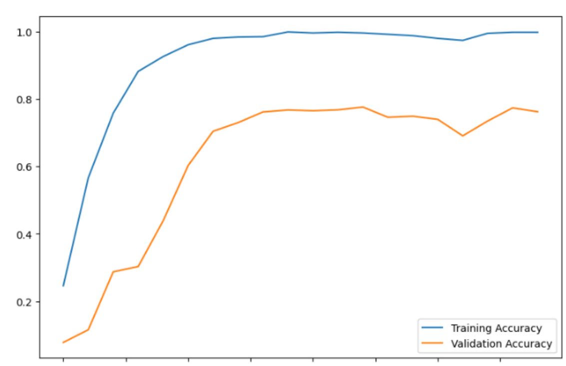
Discussion of the data augmentations use

In order to get more meaningful results out of VGG model, argumentation of the images is needed by randomly rotating the image data, zooming in/out the image data, shifting the image up and down, shifting the image channels and shearing the data. It is important to note that image data is not horizontally shifted since there are also distracting digits in the image data.

Comparison between the two DCNNs

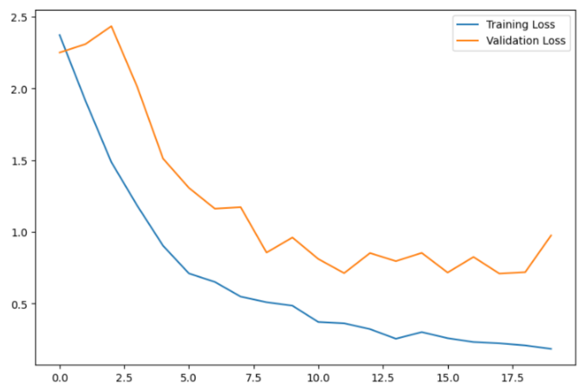
VGG Without Augmentation:

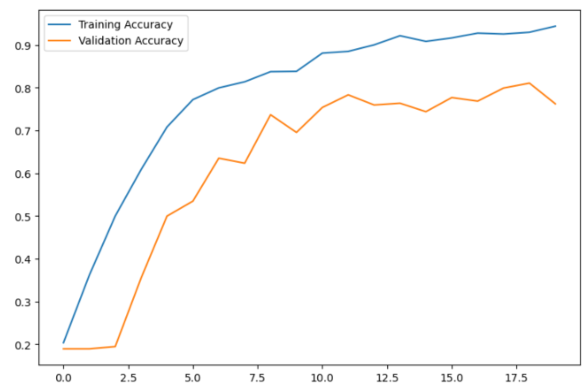




Training / Validation Loss gradually decreases over time on the other hand accuracy of both training and validation increases. Time taken for training the data took 37.415682554244995 seconds. Choice of batch size was 16 and epoch was 20. This is because smaller batch size provides much more accurate updates to the model parameters for each epoch since it uses more frequent gradient updates. A much smaller batch size can potentially result in over-fitting since it may memorize the training samples rather than generalizing well to the new data. The above two graphs also indicate there is over-fitting is performed by looking at their gaps between. For the test set accuracy, this model recorded 0.7622 with inference time of 1.04 seconds.

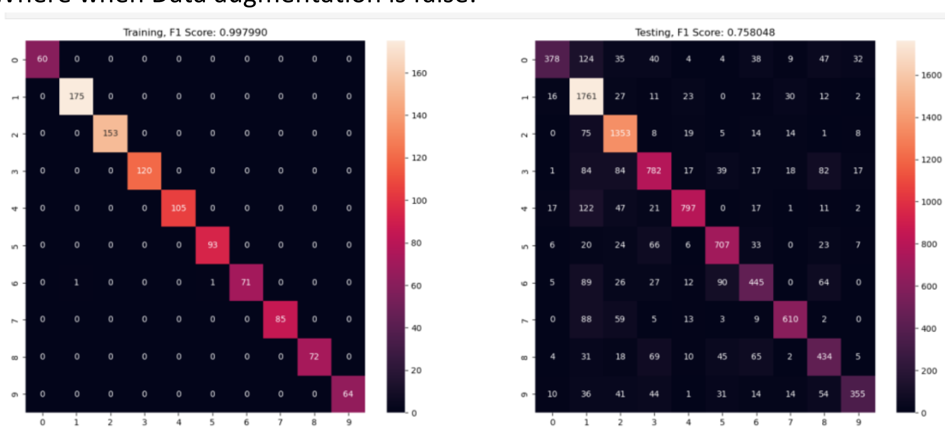
VGG With Data Augmentation:



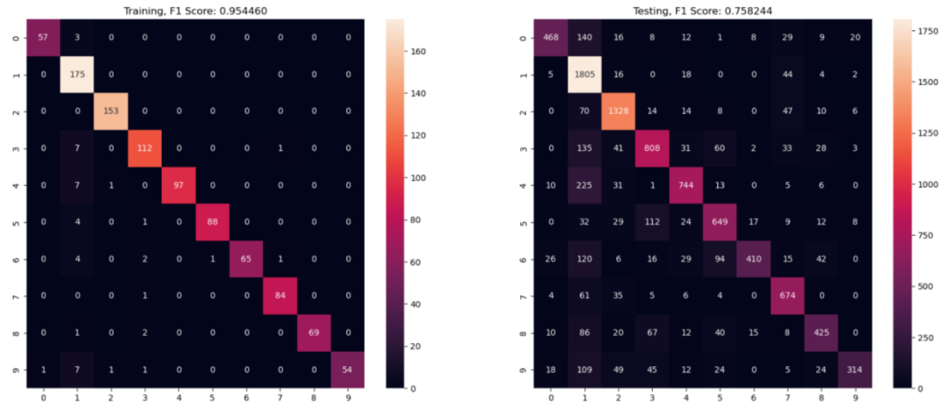


Training / Validation Loss graph and accuracy graph similarly describes the behavior of the model however, one of the crucial difference is that model's over-fitting is reduced, which the gap between training and validation is reduced. This can be visualized in F1 Score in heat map.

Where when Data augmentation is false:



And when it is true:



Smaller difference between training F1 score and testing F1 score describes over-fitting is mitigated.

Time taken for training time for this model took 29.4599711894989 seconds, accuracy of 0.6361 and inference time is 5 seconds. Reason for decreased in accuracy is because augmentation distort the original image data that results the harder for model to accurately classify images. However, since there is no huge difference between two models’ accuracy, this won’t be a crucial factor.

Appendix:

CAB420_A1A_Q1_Template

April 18, 2023

1 CAB420 Assignment 1A Question 1: Template

1.1 Linear Regression

[56]:
```
# import all the important packages

# numpy handles pretty much anything that is a number/vector/matrix/array
import numpy as np
# pandas handles dataframes (exactly the same as tables in Matlab)
import pandas
# matplotlib emulates Matlabs plotting functionality
import matplotlib.pyplot as plt
# seaborn, because of excellent heatmaps
import seaborn as sns
# stats models is a package that is going to perform the regression analysis
from statsmodels import api as sm
from scipy import stats
from sklearn.linear_model import LassoCV
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import mean_squared_error, r2_score
# os allows us to manipulate variables on out local machine, such as paths and␣
↪environment variables
import os
# self explainatory, dates and times
from datetime import datetime, date
# a helper package to help us iterate over objects
import itertools
from sklearn.linear_model import Lasso
from sklearn.linear_model import RidgeCV
from sklearn.model_selection import KFold
```

[57]:
```
train = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q1/
↪communities_train.csv')
val = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q1/communities_val.
↪csv')
test = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q1/communities_test.
↪csv')
```

```
X_train = train.iloc[:,0:-1]
y_train = train.iloc[:,-1]
X_val = val.iloc[:,0:-1]
y_val = val.iloc[:,-1]
X_test = test.iloc[:,0:-1]
y_test = test.iloc[:,-1]

X_Train = np.array(X_train, dtype=np.float64)
y_Train = np.array(y_train, dtype=np.float64)

X_Val = np.array(X_val, dtype=np.float64)
y_Val  = np.array(y_val, dtype=np.float64)

X_Test = np.array(X_test, dtype=np.float64)
y_Test = np.array(y_test, dtype=np.float64)

X_train_constant = sm.add_constant(X_Train)

X_val_constant = sm.add_constant(X_Val)
```

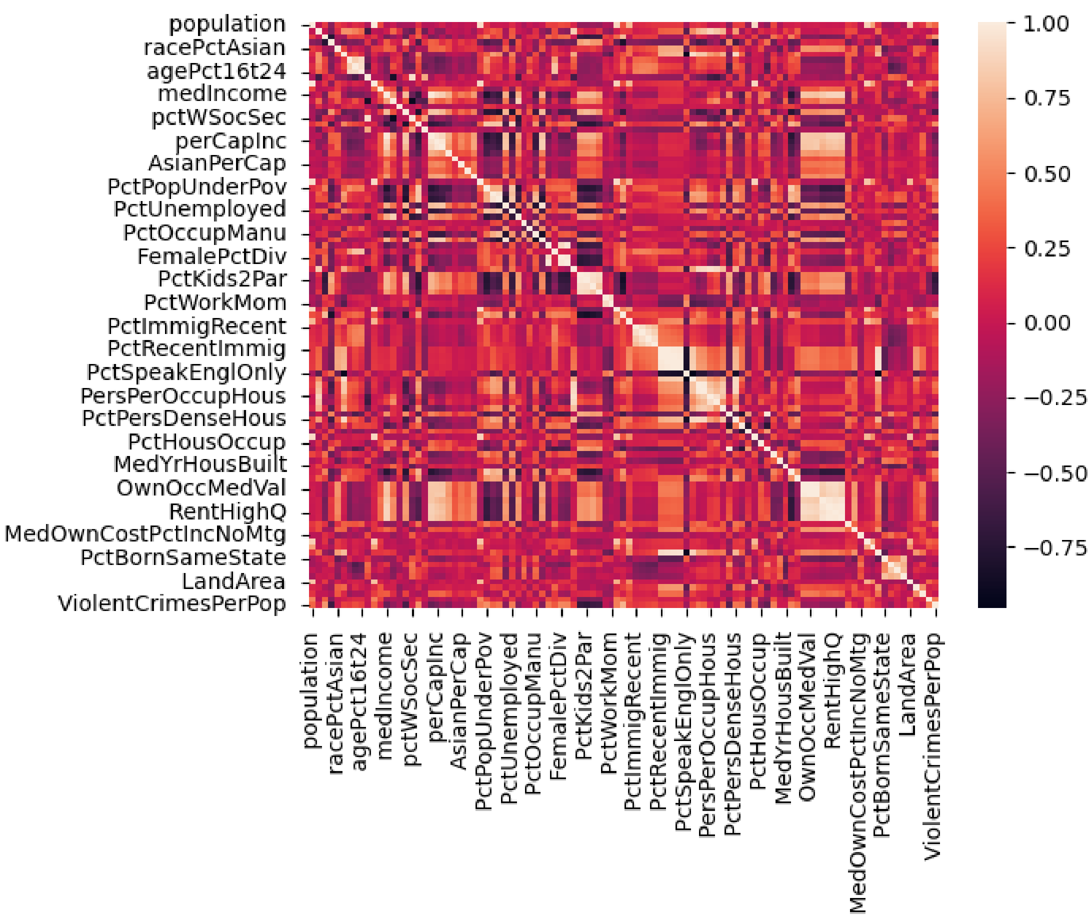
[58]:
```
X_train.head()
```

[58]:
```
   population  householdsize  racepctblack  racePctWhite  \
0        0.01           0.33          0.00          0.94
1        0.01           0.09          0.02          0.89
2        0.01           0.53          0.02          0.92
3        0.01           0.36          0.00          0.98
4        0.01           0.68          0.01          0.98

   racePctAsian  racePctHisp  agePct12t21  agePct12t29  agePct16t24  \
0          0.21         0.11         0.26         0.37         0.22
1          0.23         0.13         0.07         0.71         0.27
2          0.21         0.03         0.98         1.00         1.00
3          0.02         0.00         0.42         0.45         0.29
4          0.04         0.01         0.71         0.60         0.62

   agePct65up  …  NumStreet  PctForeignBorn  PctBornSameState  \
0        0.74  …        0.0            0.44              0.73
1        0.15  …        0.0            0.24              0.37
2        0.20  …        0.0            0.17              0.32
3        0.53  …        0.0            0.01              0.81
4        0.39  …        0.0            0.10              0.71

   PctSameHouse85  PctSameCity85  PctSameState85  LandArea  PopDens  \
0            0.90           0.73            0.85      0.01     0.45
1            0.25           0.46            0.34      0.00     1.00
```

```
2              0.13           0.04             0.18       0.01       0.32
3              0.50           0.65             0.76       0.04       0.11
4              0.67           0.73             0.55       0.08       0.06

   PctUsePubTrans   LemasPctOfficDrugUn
0            0.47                  0.0
1            0.06                  0.0
2            0.01                  0.0
3            0.00                  0.0
4            0.07                  0.0

[5 rows x 100 columns]
```

[59]:

```python
dataplot=sns.heatmap(train.corr())
plt.show()
```



[60]:

```python
#too many variabls resulting not able to see individual relationships clearly.
↪However, it is able identify groups of correlated predictos.
```

[61]:

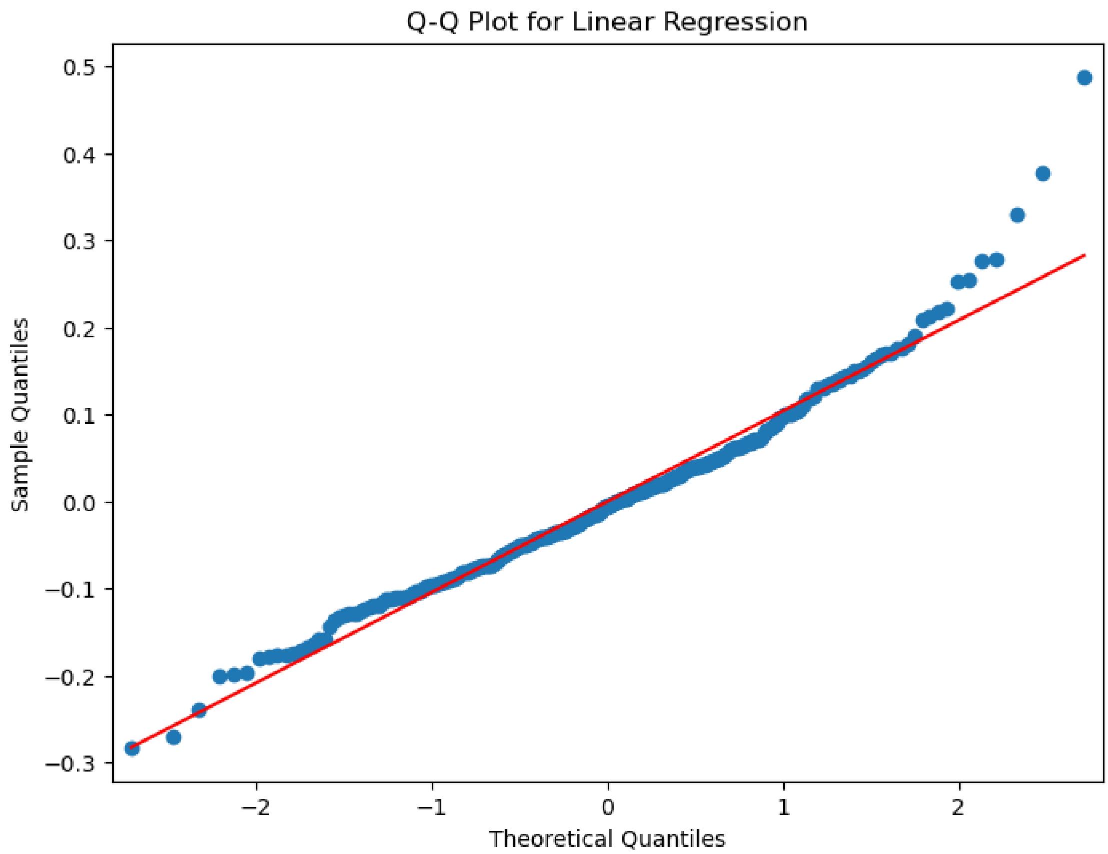
```python
model = sm.OLS(y_Train, X_train_constant)
# fit the model
model_1_fit = model.fit()
pred = model_1_fit.predict(X_val_constant)
print('Linear Model Validation Data: RMSE = {}'.format(
  np.sqrt(mean_squared_error(y_Val, pred))))
print('Linear Model Validation Data: RMSE = {}'.format(
  np.sqrt(mean_squared_error(y_Train, pred))))
print(model_1_fit.summary())
print(model_1_fit.params)
fig, ax = plt.subplots(figsize=(8,6))
sm.qqplot(model_1_fit.resid, ax=ax, line='s')
plt.title('Q-Q Plot for Linear Regression')
plt.show()
```

```
Linear Model Validation Data: RMSE = 0.15491210066258207
Linear Model Validation Data: RMSE = 0.291599337438958
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.759
Model:                            OLS   Adj. R-squared:                  0.637
Method:                 Least Squares   F-statistic:                     6.207
Date:                Mon, 17 Apr 2023   Prob (F-statistic):           7.72e-28
Time:                        18:19:51   Log-Likelihood:                 251.07
No. Observations:                 298   AIC:                            -300.1
Df Residuals:                     197   BIC:                             73.27
Df Model:                         100
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.4667      0.620      0.753      0.452      -0.755       1.689
x1             0.9391      1.225      0.767      0.444      -1.476       3.354
x2            -0.4403      0.280     -1.571      0.118      -0.993       0.112
x3             0.0398      0.157      0.253      0.800      -0.270       0.350
x4            -0.1056      0.158     -0.667      0.506      -0.418       0.207
x5            -0.0843      0.093     -0.902      0.368      -0.269       0.100
x6            -0.3792      0.154     -2.464      0.015      -0.683      -0.076
x7            -0.2405      0.316     -0.761      0.447      -0.864       0.383
x8             0.7263      0.486      1.494      0.137      -0.232       1.685
x9            -0.1488      0.510     -0.292      0.771      -1.155       0.857
x10            0.1141      0.271      0.420      0.675      -0.421       0.649
x11           -0.8932      1.167     -0.765      0.445      -3.195       1.409
x12            0.0757      0.047      1.615      0.108      -0.017       0.168
x13           -0.4012      0.607     -0.661      0.510      -1.599       0.797
x14           -0.2532      0.262     -0.967      0.335      -0.769       0.263
x15            0.0585      0.062      0.945      0.346      -0.064       0.181
x16           -0.2106      0.195     -1.080      0.282      -0.595       0.174
```

x17	0.2093	0.309	0.677	0.499	-0.401	0.819
x18	0.1046	0.150	0.698	0.486	-0.191	0.400
x19	-0.1134	0.103	-1.097	0.274	-0.317	0.091
x20	0.5219	0.553	0.944	0.346	-0.569	1.612
x21	0.6423	0.601	1.068	0.287	-0.544	1.829
x22	-0.5968	0.490	-1.217	0.225	-1.564	0.370
x23	0.0147	0.071	0.206	0.837	-0.126	0.155
x24	-0.0768	0.057	-1.346	0.180	-0.189	0.036
x25	0.1090	0.061	1.774	0.078	-0.012	0.230
x26	-0.0325	0.052	-0.630	0.529	-0.134	0.069
x27	0.0474	0.068	0.693	0.489	-0.087	0.182
x28	-0.2538	0.410	-0.618	0.537	-1.063	0.556
x29	-0.3099	0.208	-1.492	0.137	-0.719	0.100
x30	-0.1895	0.210	-0.904	0.367	-0.603	0.224
x31	0.2734	0.291	0.939	0.349	-0.301	0.847
x32	0.2973	0.239	1.243	0.215	-0.174	0.769
x33	-0.1935	0.127	-1.529	0.128	-0.443	0.056
x34	0.2072	0.256	0.810	0.419	-0.297	0.712
x35	-0.0980	0.090	-1.088	0.278	-0.276	0.080
x36	-0.0400	0.128	-0.313	0.755	-0.292	0.212
x37	0.0123	0.145	0.085	0.933	-0.273	0.298
x38	-0.0347	0.274	-0.127	0.899	-0.574	0.505
x39	-0.3890	0.658	-0.591	0.555	-1.687	0.909
x40	0.0897	0.212	0.423	0.673	-0.329	0.508
x41	-0.5718	0.792	-0.722	0.471	-2.133	0.990
x42	0.7797	1.347	0.579	0.563	-1.877	3.437
x43	-0.3109	0.528	-0.589	0.557	-1.352	0.731
x44	-0.1168	0.550	-0.212	0.832	-1.202	0.968
x45	-0.2644	0.517	-0.511	0.610	-1.285	0.756
x46	-0.0438	0.156	-0.281	0.779	-0.351	0.264
x47	-0.0107	0.121	-0.088	0.930	-0.249	0.228
x48	0.2036	0.145	1.408	0.161	-0.082	0.489
x49	-0.2839	0.160	-1.775	0.077	-0.599	0.032
x50	0.3303	0.367	0.900	0.369	-0.393	1.054
x51	0.0256	0.142	0.180	0.858	-0.255	0.306
x52	-0.1462	0.315	-0.464	0.643	-0.767	0.475
x53	0.0192	0.144	0.133	0.894	-0.264	0.303
x54	0.0639	0.194	0.329	0.742	-0.319	0.446
x55	-0.2740	0.183	-1.495	0.136	-0.635	0.087
x56	0.2105	0.151	1.395	0.165	-0.087	0.508
x57	-0.0182	0.460	-0.040	0.968	-0.925	0.889
x58	-1.2406	0.801	-1.549	0.123	-2.820	0.339
x59	1.9770	0.967	2.045	0.042	0.070	3.884
x60	-0.8199	0.704	-1.164	0.246	-2.209	0.569
x61	-0.0664	0.170	-0.390	0.697	-0.402	0.269
x62	0.0644	0.210	0.307	0.759	-0.349	0.478
x63	1.6951	0.827	2.049	0.042	0.064	3.327
x64	-1.9578	0.922	-2.123	0.035	-3.776	-0.140

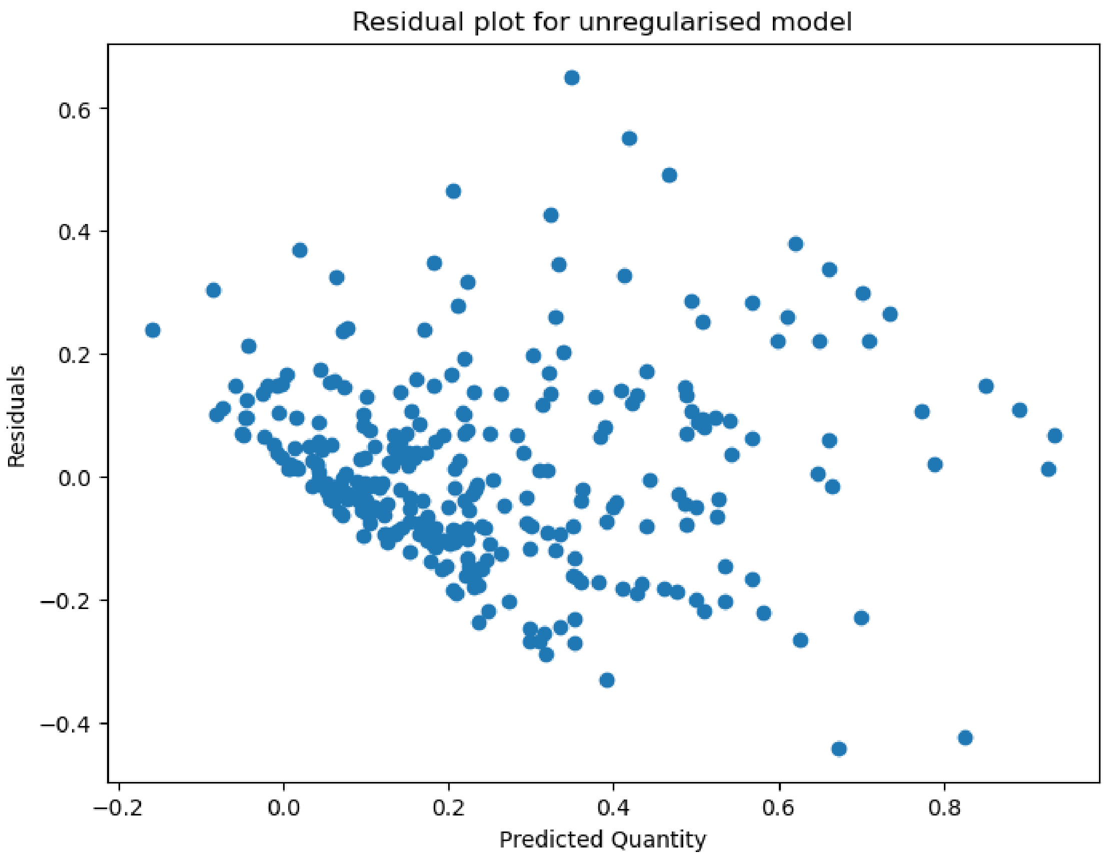
```
x65            2.2627      0.796      2.843      0.005       0.693       3.832
x66           -0.7332      0.491     -1.493      0.137      -1.701       0.235
x67           -0.2858      0.245     -1.168      0.244      -0.769       0.197
x68           -0.3760      1.000     -0.376      0.707      -2.349       1.597
x69            0.2604      0.230      1.130      0.260      -0.194       0.715
x70            0.0895      0.165      0.542      0.589      -0.236       0.415
x71            0.0272      0.053      0.509      0.611      -0.078       0.132
x72            0.2728      0.304      0.897      0.371      -0.327       0.873
x73           -0.0267      0.108     -0.247      0.806      -0.240       0.187
x74            0.1617      1.048      0.154      0.878      -1.905       2.228
x75            0.1167      0.066      1.771      0.078      -0.013       0.247
x76           -0.0012      0.075     -0.015      0.988      -0.149       0.147
x77           -0.0140      0.081     -0.173      0.863      -0.173       0.145
x78           -0.0229      0.114     -0.202      0.840      -0.247       0.201
x79            0.1009      0.065      1.556      0.121      -0.027       0.229
x80            0.4015      0.636      0.632      0.528      -0.852       1.655
x81           -0.3409      1.055     -0.323      0.747      -2.421       1.739
x82           -0.2208      0.655     -0.337      0.736      -1.512       1.070
x83           -0.1543      0.185     -0.832      0.406      -0.520       0.211
x84           -0.5515      0.522     -1.057      0.292      -1.581       0.478
x85            0.3303      0.245      1.346      0.180      -0.154       0.814
x86            0.1341      0.453      0.296      0.768      -0.760       1.028
x87            0.1688      0.108      1.559      0.121      -0.045       0.382
x88           -0.0455      0.112     -0.408      0.684      -0.265       0.175
x89           -0.0840      0.080     -1.055      0.293      -0.241       0.073
x90            0.0212      0.272      0.078      0.938      -0.514       0.557
x91            0.0591      0.121      0.487      0.627      -0.180       0.298
x92            0.1705      0.270      0.632      0.528      -0.361       0.702
x93           -0.1216      0.126     -0.961      0.338      -0.371       0.128
x94           -0.2140      0.186     -1.149      0.252      -0.581       0.153
x95            0.0470      0.119      0.396      0.693      -0.187       0.281
x96            0.1682      0.136      1.236      0.218      -0.100       0.437
x97           -0.2789      0.204     -1.370      0.172      -0.680       0.122
x98           -0.1849      0.086     -2.155      0.032      -0.354      -0.016
x99           -0.0130      0.062     -0.209      0.834      -0.135       0.109
x100           0.0897      0.047      1.907      0.058      -0.003       0.182
==============================================================================
Omnibus:                       37.227   Durbin-Watson:                   1.837
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               67.083
Skew:                           0.704   Prob(JB):                     2.71e-15
Kurtosis:                       4.849   Cond. No.                         967.
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[ 4.66714682e-01  9.39083877e-01 -4.40295812e-01  3.98107805e-02
 -1.05617082e-01 -8.42959763e-02 -3.79186520e-01 -2.40495493e-01
```

```
 7.26305980e-01 -1.48831647e-01  1.14113225e-01 -8.93210876e-01
 7.56750892e-02 -4.01232189e-01 -2.53190525e-01  5.85271682e-02
-2.10591004e-01  2.09285595e-01  1.04641420e-01 -1.13430089e-01
 5.21870264e-01  6.42307002e-01 -5.96827098e-01  1.46784827e-02
-7.68334594e-02  1.08951281e-01 -3.25178102e-02  4.73522770e-02
-2.53802751e-01 -3.09909412e-01 -1.89469670e-01  2.73358778e-01
 2.97280376e-01 -1.93514370e-01  2.07217115e-01 -9.80102715e-02
-3.99547933e-02  1.22625455e-02 -3.46873706e-02 -3.88983107e-01
 8.97226597e-02 -5.71784089e-01  7.79699819e-01 -3.10924685e-01
-1.16779812e-01 -2.64367080e-01 -4.38190293e-02 -1.06882445e-02
 2.03649179e-01 -2.83872456e-01  3.30258796e-01  2.55825395e-02
-1.46170752e-01  1.91595196e-02  6.38613861e-02 -2.73985143e-01
 2.10479807e-01 -1.82205215e-02 -1.24056893e+00  1.97697163e+00
-8.19850972e-01 -6.63669976e-02  6.44300143e-02  1.69514027e+00
-1.95776015e+00  2.26267603e+00 -7.33224823e-01 -2.85836630e-01
-3.75962218e-01  2.60356081e-01  8.94595147e-02  2.71533840e-02
 2.72828377e-01 -2.66757843e-02  1.61681270e-01  1.16659787e-01
-1.15033150e-03 -1.39962894e-02 -2.29433814e-02  1.00935056e-01
 4.01545998e-01 -3.40929039e-01 -2.20780777e-01 -1.54275257e-01
-5.51470684e-01  3.30252125e-01  1.34129795e-01  1.68812942e-01
-4.54603666e-02 -8.39959522e-02  2.12460157e-02  5.90563931e-02
 1.70537461e-01 -1.21591289e-01 -2.14029335e-01  4.69524856e-02
 1.68220276e-01 -2.78913430e-01 -1.84869115e-01 -1.29596889e-02
 8.96764818e-02]
```

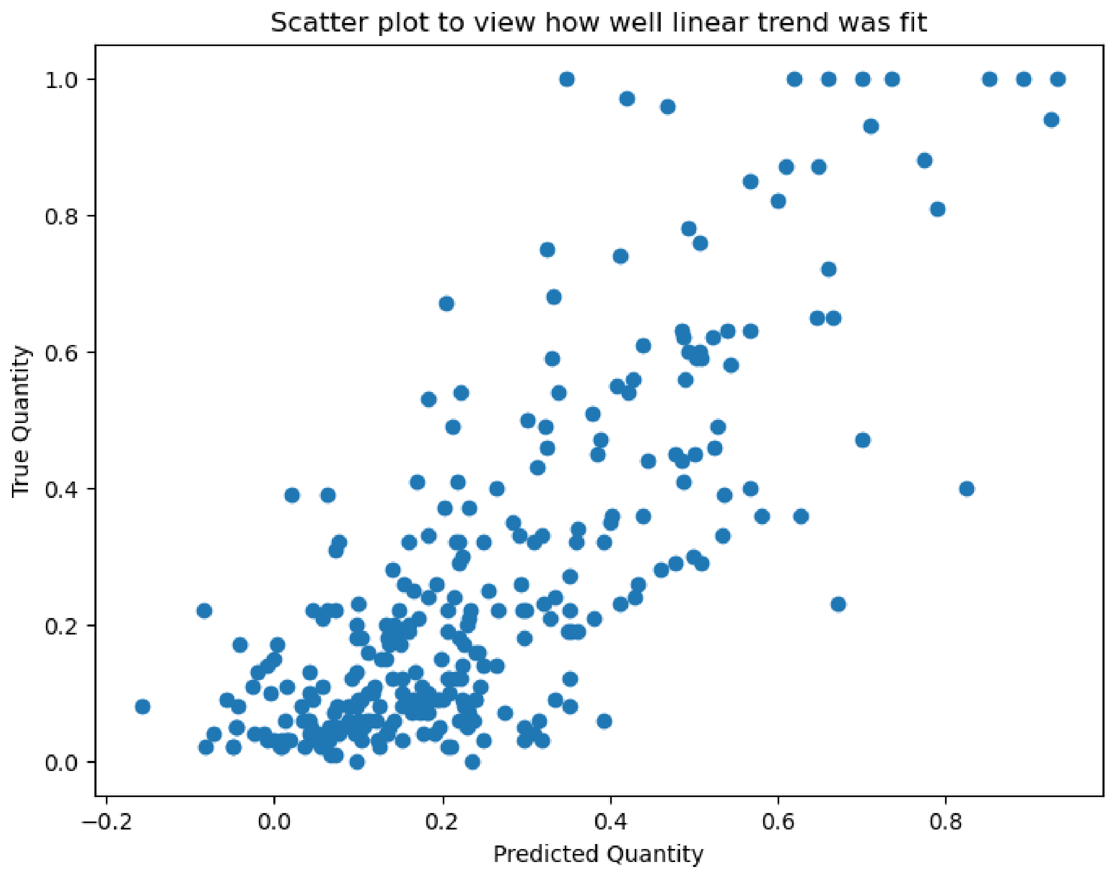


1.2 unregularised plot

[62]:
```
fig, ax = plt.subplots(figsize=(8,6))
plt.scatter(pred, y_Val - pred)
plt.title('Residual plot for unregularised model')
plt.xlabel('Predicted Quantity')
plt.ylabel('Residuals')
plt.show()
```



[63]:
```
fig, ax = plt.subplots(figsize=(8,6))
plt.scatter(pred, y_Val)
plt.title('Scatter plot to view how well linear trend was fit')
plt.xlabel('Predicted Quantity')
plt.ylabel('True Quantity')
plt.show()
```



2 Standardised Data

[64]:
```python
def standardise(data):
    """ Standardise/Normalise data to have zero mean and unit variance

    Args:
      data (np.array):
        data we want to standardise (usually covariates)

      Returns:
        Standardised data, mean of data, standard deviation of data
    """
    mu = np.mean(data, axis=0)
    sigma = np.std(data, axis=0)
    scaled = (data - mu) / sigma
    return scaled, mu, sigma
```

[65]:
```
X_train_std, mu_train_x, sigma_train_x = standardise(X_Train)
Y_train_std, mu_train_y, sigma_train_y = standardise(y_Train)
X_val_std = (X_Val - mu_train_x)/sigma_train_x
Y_val_std = (y_Val - mu_train_y)/sigma_train_y
X_test_std = (X_Test - mu_train_x)/sigma_train_x
Y_test_std = (y_Test - mu_train_y)/sigma_train_y
```

3 Ridge Regression

[77]:
```
def rmse(actual, pred):
  return np.sqrt(mean_squared_error(actual, pred))

def r_squared(actual, predicted):
  r2 = r2_score(actual, predicted)
  return r2

def adj_r2(actual, predicted, n, p):
  r2 = r2_score(actual, predicted)
  adjr2 = 1 - (1 - r2) * (n - 1) / (n - p - 1);
  return adjr2

def evaluate_regularisation(X_Train, y_Train, X_Val, y_Val, X_Test, y_Test,
                            response_mu, response_sigma, alpha_list, L1_L2):


  # Ridge: L1_L2 = 0
  # Lasso: L1_L2 = 1
  # create the model
  model = sm.OLS(y_Train, X_Train)
  # initialise the value for best RMSE that is obnoxiously large, as we want␣
↪this be
  # overwritten each time RMSE is smaller, since smaller is better and we want␣
↪to
  # update our best models each time the RMSE is smaller.
  best_rmse = 10e12
  best_alpha = []
  best_coeffs = []

  rmse_val = []
  rmse_train = []
  coeffs = []          # only needed for trace plots

  for alpha in alpha_list:
    model_cross_fit = model.fit_regularized(alpha=alpha, L1_wt=L1_L2)
    train_pred = model_cross_fit.predict(X_Train)
    val_pred = model_cross_fit.predict(X_Val)
```

```
  # want to append the rmse value to a list, as will plot all values later on
  rmse_train.append(np.sqrt(mean_squared_error(y_Train, train_pred)))
  rmse_val.append(np.sqrt(mean_squared_error(y_Val, val_pred)))
  coeffs.append(model_cross_fit.params)
  # if this is the model with the lowest RMSE, lets save it
  # the [-1] index says get the last value from the list (which is t`he most␣
↪recent RMSE)
  if rmse_val[-1] < best_rmse:
    best_rmse = rmse_val[-1]
    best_alpha = alpha
    best_coeffs = model_cross_fit.params


# extract the gradient and the bias from the coefficients
# The reshape will make sure the slope is a column vector
slope = np.array(best_coeffs[0:]).reshape(-1,1)

# the intercept coefficient is the last index variable, which was included␣
↪with the
# sm.add_constant() method
# use the @ operator to perform vector/matrix multiplication
pred_val_rescaled = (X_Val @ slope) * response_sigma + response_mu
pred_test_rescaled = (X_Test @ slope) * response_sigma + response_mu
pred_train_rescaled = (X_Train @ slope) * response_sigma + response_mu
best_r2 = r_squared(y_Train * response_sigma + response_mu,␣
↪pred_train_rescaled)
best_adj_r2 = adj_r2(y_Train * response_sigma + response_mu,␣
↪pred_train_rescaled,
                         X_train.shape[0], X_train.shape[1])
best_val_rmse = np.sqrt(mean_squared_error(y_val* response_sigma +␣
↪response_mu, pred_val_rescaled))
best_test_rmse = np.sqrt(mean_squared_error(y_test* response_sigma +␣
↪response_mu, pred_test_rescaled))
print('Best R Squared = {}'.format(best_r2))
print('Best Adjusted = {}'.format(best_adj_r2))
print('Best RMSE (val) = {}'.format(best_val_rmse))
print('Best RMSE (test) = {}'.format(best_test_rmse))
print('Best coefficients on the normalised model')
print('Best slope = {}'.format(slope))
print('best_alpha = {}'.format(alpha))


# now plotting some data
fig, axs = plt.subplots(5, figsize=(15, 25))

# plot the first values of alpha vs RMSE for train and validation data
```

```
    axs[0].plot(np.array(alpha_list), rmse_train)
    axs[0].plot(np.array(alpha_list), rmse_val)
    axs[0].legend(['Training', 'Validation'])
    axs[0].set_title('RMSE vs Lambda')
    axs[0].set_xlabel('Lambda')
    axs[0].set_ylabel('RMSE')

    # plot prediction and true values for test set
    axs[1].plot((y_Test*response_sigma + response_mu))
    axs[1].plot((X_Test @ slope) * response_sigma + response_mu)
    axs[1].legend(['Actual', 'Predicted'])
    axs[1].set_title('Test Set Performance')

    # plotting the Q-Q plot
    train_pred = (X_Train @ slope).reshape(y_train.shape)
    resid = y_Train - train_pred
    sm.qqplot(resid, ax=axs[2], line='s')
    axs[2].set_title('Q-Q Plot for Linear Regression')

    # plot the residuals as well
    axs[3].scatter(train_pred, resid)
    axs[3].set_title('Residuals for training set')
    axs[3].set_xlabel('Predicted')
    axs[3].set_ylabel('Residuals')

    # trace plot of coefficients
    axs[4].plot(np.array(alpha_list), coeffs)
    axs[4].set_title('Trace PLot of Coefficients')
    axs[4].set_xlabel('Lambda')
    axs[4].set_ylabel('Coefficient Value')
```

3.1 Optimal Lambda

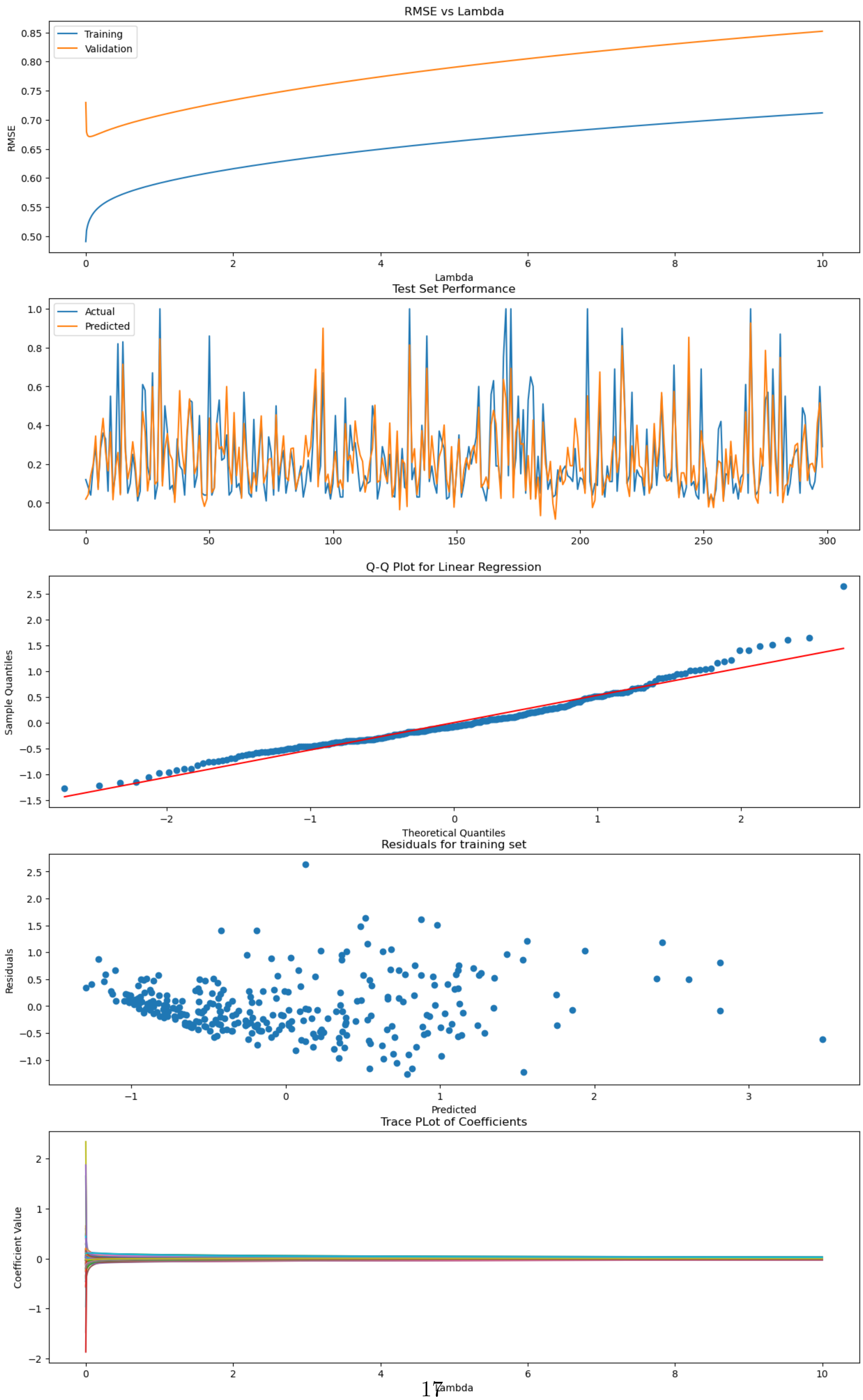
[78]:
```
alpha_list = np.linspace(0, 10.0, 1000)
evaluate_regularisation(X_train_std, Y_train_std, X_val_std, Y_val_std,␣
 ↪X_test_std, Y_test_std,
                        mu_train_y, sigma_train_y, alpha_list, 0)
```

```
Best R Squared = 0.7185647427447908
Best Adjusted = 0.5757042060670196
Best RMSE (val) = 0.1553748181777985
Best RMSE (test) = 0.15560311713881106
Best coefficients on the normalised model
Best slope = [[ 0.00654498]
 [-0.08261657]
 [ 0.08734514]
 [-0.05381302]
```

```
[ 0.00138797]
[-0.17480713]
[-0.06013663]
[ 0.04399552]
[ 0.06563322]
[ 0.06039095]
[ 0.00088043]
[ 0.07462407]
[ 0.05172256]
[-0.05027768]
[ 0.0145202 ]
[-0.10611063]
[ 0.06574661]
[ 0.0472038 ]
[-0.08570737]
[ 0.02329233]
[ 0.00902525]
[ 0.00711943]
[-0.00206784]
[-0.03520741]
[ 0.08767832]
[ 0.00606919]
[ 0.05248105]
[-0.02268259]
[-0.11045956]
[-0.04505044]
[ 0.03164664]
[ 0.03493553]
[-0.09916151]
[ 0.13181493]
[-0.07786563]
[-0.00953879]
[ 0.01238723]
[-0.02661636]
[-0.01863145]
[ 0.00841003]
[-0.00555766]
[-0.00972973]
[ 0.06706536]
[-0.09915312]
[-0.11831371]
[-0.07773286]
[-0.01980406]
[ 0.06102459]
[-0.09878858]
[ 0.05979634]
[ 0.11763644]
[-0.04987964]
```

```
[-0.04014837]
[-0.03892786]
[-0.03031703]
[ 0.08912991]
[-0.04927747]
[-0.03611297]
[ 0.03471484]
[ 0.04679283]
[ 0.01996326]
[ 0.0278842 ]
[ 0.05228118]
[ 0.01819383]
[ 0.1008962 ]
[-0.08024458]
[ 0.05195562]
[-0.0856257 ]
[ 0.12086827]
[ 0.02383889]
[-0.00466512]
[ 0.06543783]
[-0.04723035]
[-0.01181503]
[ 0.09259166]
[-0.05366814]
[ 0.00778681]
[-0.01068907]
[ 0.04954241]
[-0.01463279]
[-0.0371072 ]
[-0.04744734]
[-0.11969922]
[-0.00759524]
[ 0.06813818]
[ 0.02553026]
[ 0.11581675]
[-0.00942537]
[-0.04890958]
[ 0.02063894]
[ 0.04515878]
[ 0.04042541]
[-0.03863748]
[-0.01616871]
[ 0.00206239]
[ 0.04937343]
[-0.07059539]
[-0.07982754]
[-0.0185454 ]
[ 0.10968608]]
```

```
best_alpha = 10.0
```


RMSE vs Lambda
Training
Validation
RMSE
Lambda
Test Set Performance
Actual
Predicted
Q-Q Plot for Linear Regression
Sample Quantiles
Theoretical Quantiles
Residuals for training set
Residuals
Predicted
Trace PLot of Coefficients
Coefficient Value
Lambda

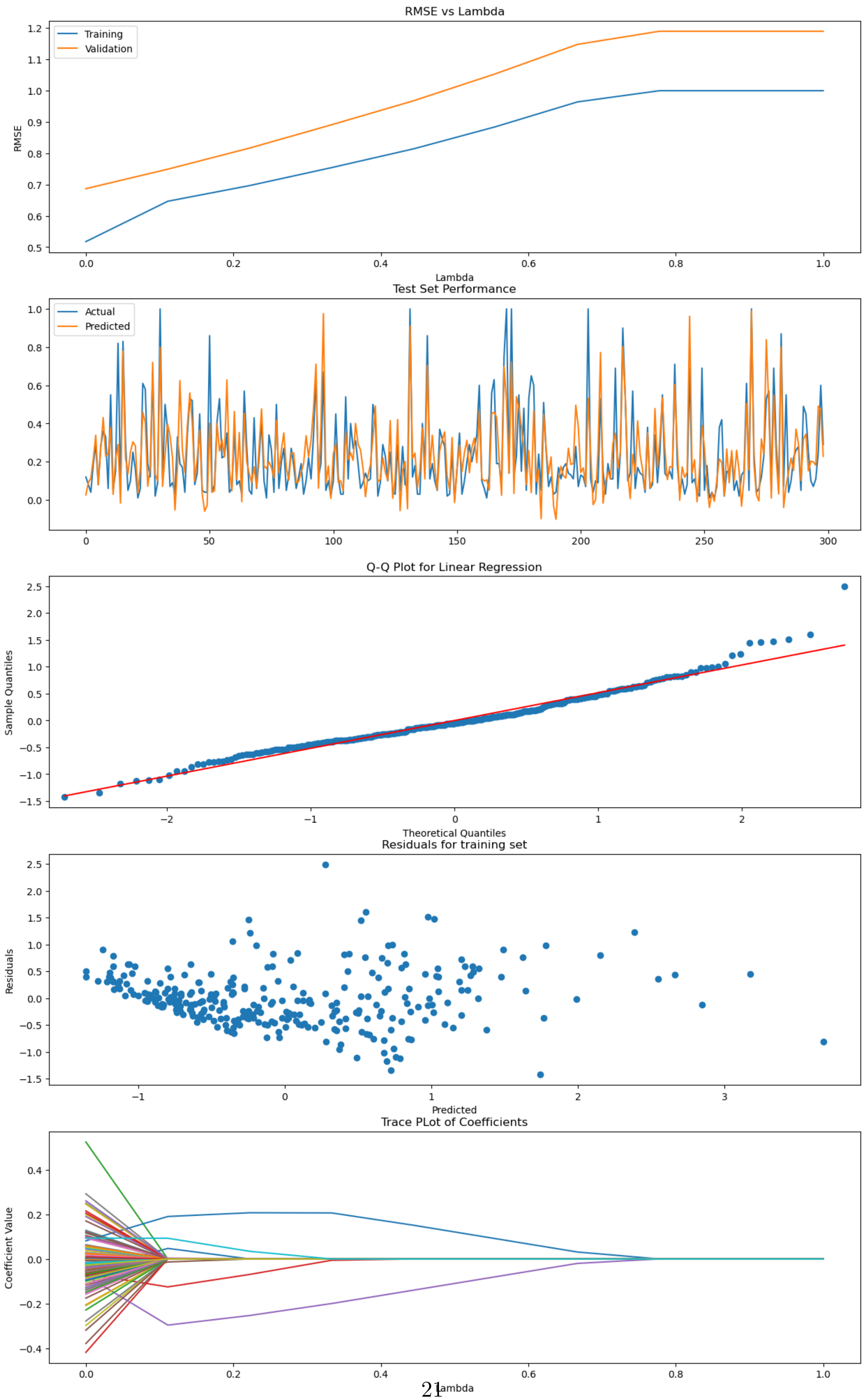
4 Lasso Regression

[79]:
```
alpha_list = np.linspace(0, 1, 10)
evaluate_regularisation(X_train_std, Y_train_std, X_val_std, Y_val_std,␣
↪X_test_std, Y_test_std,
                        mu_train_y, sigma_train_y, alpha_list, 1)
```

```
Best R Squared = 0.7320517436259425
Best Adjusted = 0.5960374002888575
Best RMSE (val) = 0.16616256537828702
Best RMSE (test) = 0.16882789195092898
Best coefficients on the normalised model
Best slope = [[ 0.0951327 ]
 [-0.20536406]
 [ 0.0158983 ]
 [-0.06581499]
 [-0.05764481]
 [-0.3790592 ]
 [-0.15208163]
 [ 0.29218796]
 [ 0.03748696]
 [ 0.12846291]
 [-0.05321039]
 [ 0.09672982]
 [ 0.1158247 ]
 [-0.13927802]
 [ 0.04493583]
 [-0.11683339]
 [-0.02114606]
 [-0.00303662]
 [-0.10965065]
 [ 0.00494789]
 [-0.02783488]
 [-0.05502541]
 [ 0.02587699]
 [-0.04134678]
 [ 0.11886312]
 [-0.00776218]
 [ 0.02960271]
 [ 0.06361052]
 [-0.29800255]
 [-0.07942747]
 [ 0.25015345]
 [ 0.24614915]
```

```
[-0.09951367]
[ 0.1050791 ]
[-0.13859496]
[-0.01563503]
[ 0.05206444]
[-0.12651914]
[-0.07338288]
[-0.02236428]
[-0.0581375 ]
[ 0.0215339 ]
[ 0.52461685]
[-0.41868846]
[-0.07428744]
[-0.07741337]
[ 0.00366627]
[ 0.12728814]
[-0.20872803]
[ 0.04936899]
[ 0.08138725]
[-0.05980208]
[-0.00161817]
[-0.07159912]
[-0.12805884]
[ 0.17036167]
[-0.158442  ]
[-0.00187816]
[ 0.19930234]
[ 0.02816141]
[-0.00181524]
[-0.00088959]
[-0.02557436]
[-0.06466426]
[ 0.19016943]
[-0.31912778]
[-0.09151668]
[-0.27818437]
[ 0.24783474]
[-0.04286527]
[ 0.00424273]
[ 0.02611288]
[-0.06787612]
[ 0.20610695]
[ 0.10131124]
[-0.04936255]
[ 0.01720174]
[-0.05760721]
[ 0.05371743]
[-0.01689523]
```

```
 [-0.0948886 ]
 [-0.04268313]
 [-0.22863811]
 [ 0.21533216]
 [ 0.26054733]
 [-0.17559524]
 [ 0.09286631]
 [-0.03487035]
 [-0.08540568]
 [-0.02228401]
 [ 0.06074456]
 [ 0.05956339]
 [-0.14859325]
 [ 0.00984575]
 [-0.04050163]
 [ 0.12134044]
 [-0.11318151]
 [-0.13590737]
 [-0.02741896]
 [ 0.09359384]]
best_alpha = 1.0
```


RMSE vs Lambda
Training
Validation
RMSE
Lambda
Test Set Performance
Actual
Predicted
Q-Q Plot for Linear Regression
Sample Quantiles
Theoretical Quantiles
Residuals for training set
Residuals
Predicted
Trace PLot of Coefficients
Coefficient Value
Lambda

```
[ ]:
```

```
[ ]:
```

CAB420_A1A_Q2_Template

April 18, 2023

1 CAB420 Assignment 1A Question 2: Template

Simon Denman (s.denman@qut.edu.au)

1.1 Overview

This notebook provides a brief template for CAB420 Assignment 1A, Question 2. It simply implements the data loading, and splitting the data into the predictors and response. You are to use the data splits defined here in your response.

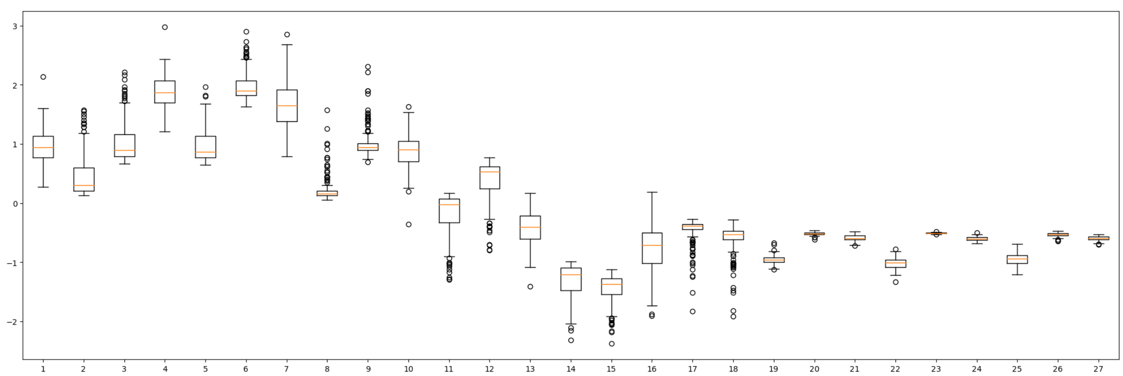
Note: File paths used in this template may need to change for your local machine. Please set these based on your local file system structure.

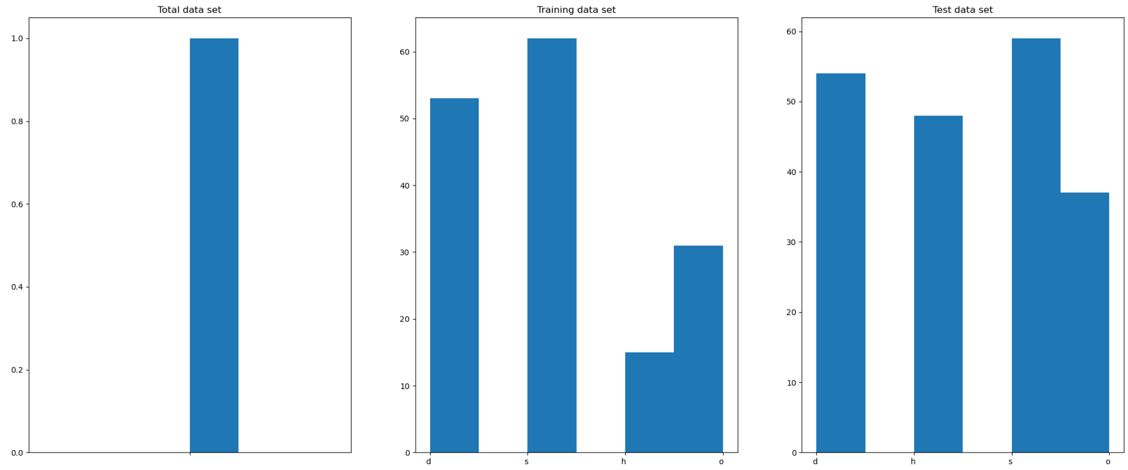
[2]:
```
import pandas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import re
import string
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.svm import SVC
from sklearn.multiclass import OneVsRestClassifier, OneVsOneClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import precision_score, recall_score, f1_score,␣
 ↪classification_report
from sklearn.model_selection import GridSearchCV
from scipy.stats import norm
from sklearn.datasets import load_digits
# load data
train = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q2/testing.csv')
val = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q2/training.csv')
test = pandas.read_csv('/home/n9894403/cab420/cab420 pracs/Q2/training.csv')

# pull out X and y data, convert to numpy
X_train = train.iloc[:,1:].to_numpy()
```

```
Y_train = train.iloc[:,0].to_numpy()
X_val = val.iloc[:,1:].to_numpy()
Y_val = val.iloc[:,0].to_numpy()
X_test = test.iloc[:,1:].to_numpy()
Y_test = test.iloc[:,0].to_numpy()
```

[7]:
```
fig = plt.figure(figsize=[25, 8])
ax = fig.add_subplot(1, 1, 1)
ax.boxplot(X_train);
X = ''
# now get the response variable by just getting the `quality` column
Y = ''
# having a look at class imbalance
fig = plt.figure(figsize=[25, 10])
ax = fig.add_subplot(1, 3, 1)
ax.hist(Y, 6)
ax.set_title('Total data set')
ax = fig.add_subplot(1, 3, 2)
ax.hist(Y_train, 6)
ax.set_title('Training data set')
ax = fig.add_subplot(1, 3, 3)
ax.hist(Y_test, 6)
ax.set_title('Test data set')
```

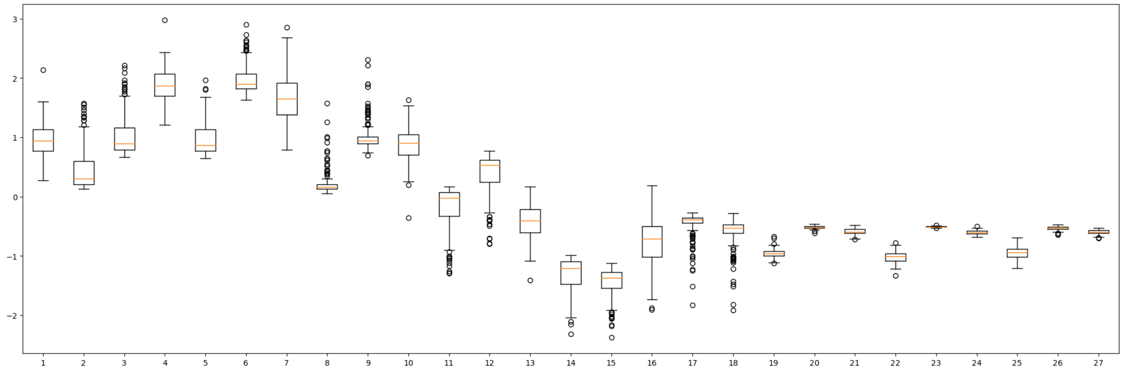
[7]: Text(0.5, 1.0, 'Test data set')





[6]:
```
mu = np.mean(X_train)
sigma = np.std(X_train)
X_train = (X_train - mu) / sigma;
X_test = (X_test - mu) / sigma;
```

[5]:
```
fig = plt.figure(figsize=[25, 8])
ax = fig.add_subplot(1, 1, 1)
ax.boxplot(X_train);
```



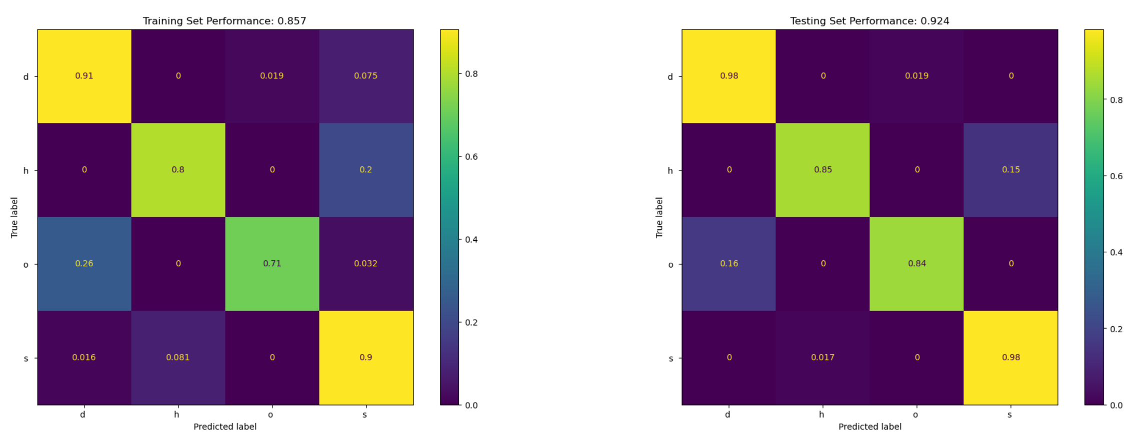
[9]:
```
def eval_model(model, X_train, Y_train, X_test, Y_test):
    fig = plt.figure(figsize=[25, 8])
    ax = fig.add_subplot(1, 2, 1)
    conf = ConfusionMatrixDisplay.from_estimator(model, X_train, Y_train,␣
 ↪normalize='true', ax=ax)
    conf.ax_.set_title('Training Set Performance: %1.3f' % (sum(model.
 ↪predict(X_train) == Y_train)/len(Y_train)));
    ax = fig.add_subplot(1, 2, 2)
```

```
    conf = ConfusionMatrixDisplay.from_estimator(model, X_test, Y_test, normalize='true', ax=ax)
    conf.ax_.set_title('Testing Set Performance: %1.3f' % (sum(model.predict(X_test) == Y_test)/len(Y_test)));
    print(classification_report(Y_test, model.predict(X_test)))
```

2 K Nearest Neighbours Classifier

[10]:
```
cknn = KNeighborsClassifier(n_neighbors=8)
cknn.fit(X_train, Y_train)
eval_model(cknn, X_train, Y_train, X_test, Y_test)
```

	precision	recall	f1-score	support
d	0.90	0.98	0.94	54
h	0.98	0.85	0.91	48
o	0.97	0.84	0.90	37
s	0.89	0.98	0.94	59
accuracy			0.92	198
macro avg	0.93	0.91	0.92	198
weighted avg	0.93	0.92	0.92	198



It's fair to say, this goes badly. We have massive class imbalance as seen above, so need to reduce the 'NumNeighbors' parameter to increase the chance of being able to get these rare classes right. If we have this too big, then by virtue of a lack of sample points, these rare classes will always be classified as something else simply becuase there are not enough points.

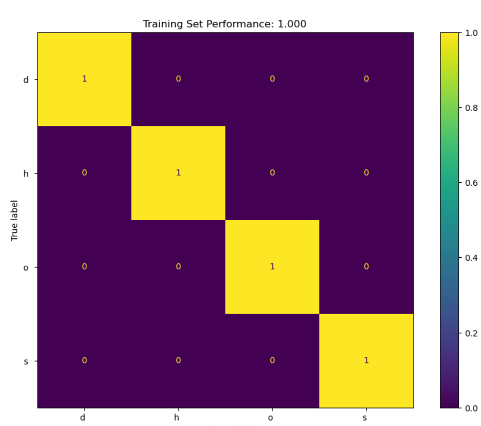
[36]:
```
values_of_k = [1, 2, 4, 8, 16, 32, 64, 128]
for k in values_of_k:
    cknn = KNeighborsClassifier(n_neighbors=k, weights='distance')
```

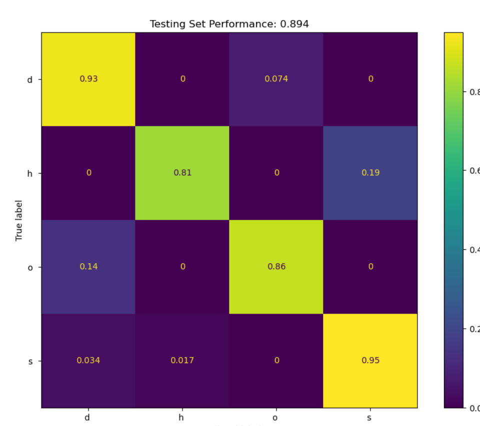
```
cknn.fit(X_train, Y_train)
eval_model(cknn, X_train, Y_train, X_test, Y_test)
```

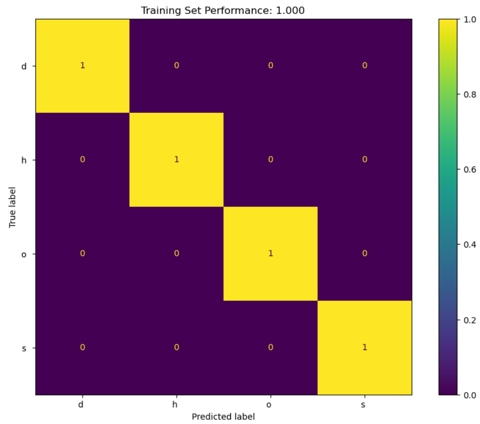
```
              precision    recall  f1-score   support

           d       0.88      0.93      0.90        54
           h       0.97      0.81      0.89        48
           o       0.89      0.86      0.88        37
           s       0.86      0.95      0.90        59

    accuracy                           0.89       198
   macro avg       0.90      0.89      0.89       198
weighted avg       0.90      0.89      0.89       198

              precision    recall  f1-score   support

           d       0.88      0.93      0.90        54
           h       0.97      0.81      0.89        48
           o       0.89      0.86      0.88        37
           s       0.86      0.95      0.90        59

    accuracy                           0.89       198
   macro avg       0.90      0.89      0.89       198
weighted avg       0.90      0.89      0.89       198

              precision    recall  f1-score   support

           d       0.91      0.96      0.94        54
           h       0.97      0.79      0.87        48
           o       0.94      0.86      0.90        37
           s       0.85      0.98      0.91        59

    accuracy                           0.91       198
   macro avg       0.92      0.90      0.91       198
weighted avg       0.92      0.91      0.91       198

              precision    recall  f1-score   support

           d       0.90      0.98      0.94        54
           h       0.98      0.83      0.90        48
           o       0.97      0.84      0.90        37
           s       0.88      0.98      0.93        59

    accuracy                           0.92       198
   macro avg       0.93      0.91      0.92       198
weighted avg       0.92      0.92      0.92       198

              precision    recall  f1-score   support
```

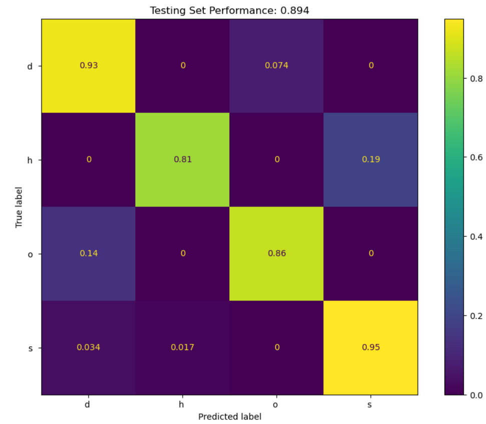
```
           d       0.88      0.94      0.91        54
           h       0.97      0.79      0.87        48
           o       0.94      0.81      0.87        37
           s       0.84      0.98      0.91        59

    accuracy                           0.89       198
   macro avg       0.91      0.88      0.89       198
weighted avg       0.90      0.89      0.89       198

              precision    recall  f1-score   support

           d       0.87      0.96      0.91        54
           h       0.97      0.69      0.80        48
           o       0.97      0.76      0.85        37
           s       0.77      0.98      0.87        59

    accuracy                           0.86       198
   macro avg       0.89      0.85      0.86       198
weighted avg       0.88      0.86      0.86       198

/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
```

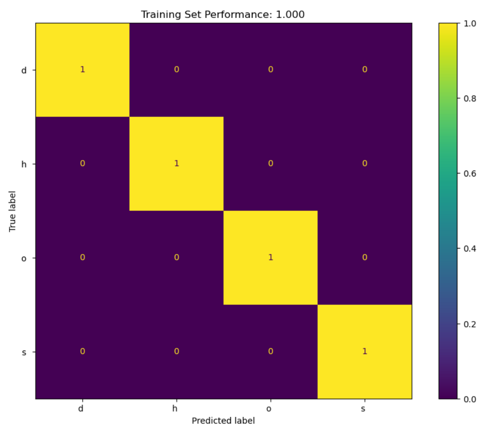
```
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           d       0.76      0.89      0.82        54
           h       0.00      0.00      0.00        48
           o       1.00      0.57      0.72        37
           s       0.52      1.00      0.68        59

    accuracy                           0.65       198
   macro avg       0.57      0.61      0.56       198
weighted avg       0.55      0.65      0.56       198

              precision    recall  f1-score   support

           d       0.64      0.85      0.73        54
           h       0.00      0.00      0.00        48
           o       1.00      0.22      0.36        37
           s       0.50      1.00      0.67        59

    accuracy                           0.57       198
   macro avg       0.53      0.52      0.44       198
weighted avg       0.51      0.57      0.46       198
```

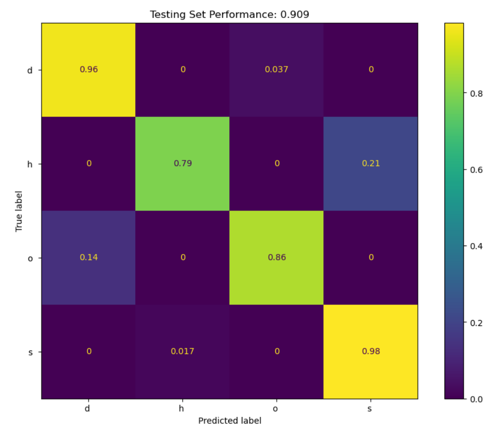


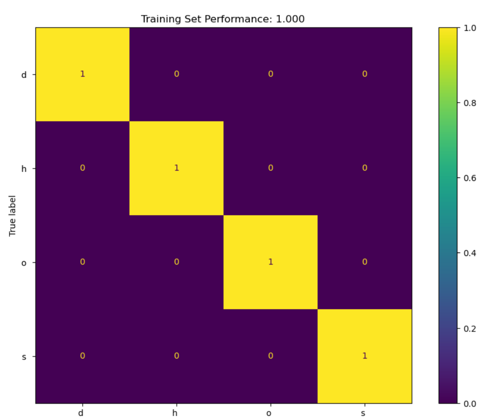


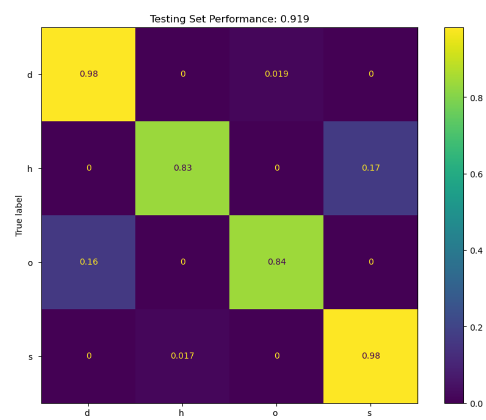

Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
0.0 0.2 0.4 0.6 0.8 1.0

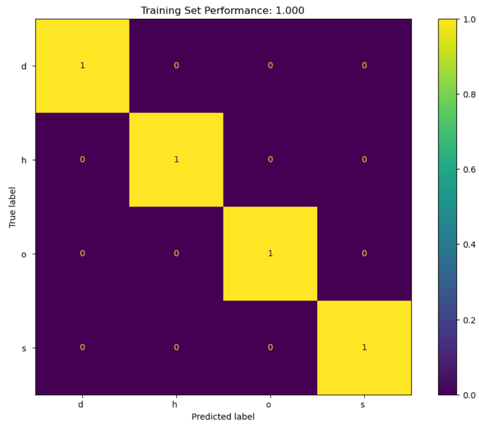

Testing Set Performance: 0.894
True label
Predicted label
d h o s
0.93 0 0.074 0
0 0.81 0 0.19
0.14 0 0.86 0
0.034 0.017 0 0.95
0.0 0.2 0.4 0.6 0.8


Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
0.0 0.2 0.4 0.6 0.8 1.0

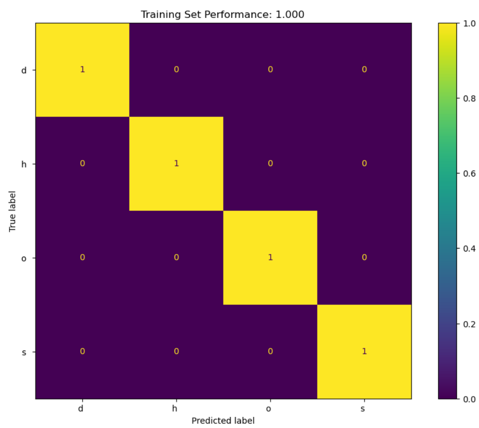

Testing Set Performance: 0.909
True label
Predicted label
d h o s
0.96 0 0.037 0
0 0.79 0 0.21
0.14 0 0.86 0
0 0.017 0 0.98
0.0 0.2 0.4 0.6 0.8

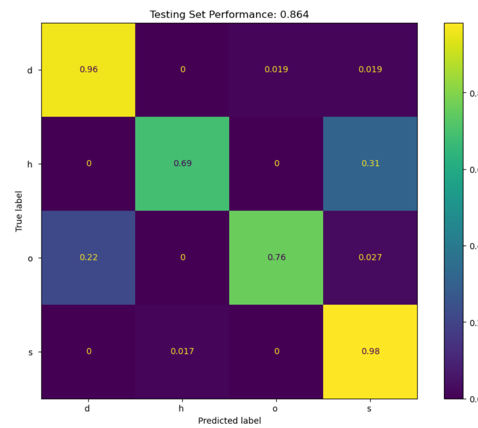

Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
0.0 0.2 0.4 0.6 0.8 1.0

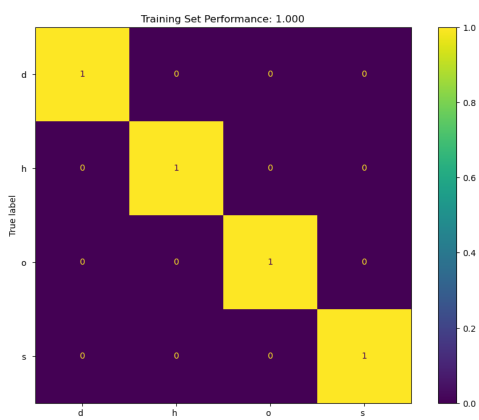

Testing Set Performance: 0.919
True label
Predicted label
d h o s
0.98 0 0.019 0
0 0.83 0 0.17
0.16 0 0.84 0
0 0.017 0 0.98
0.0 0.2 0.4 0.6 0.8

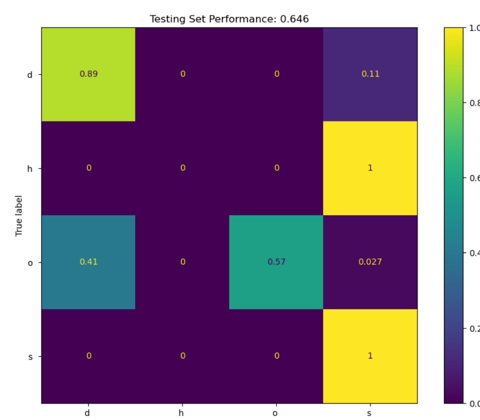

Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1

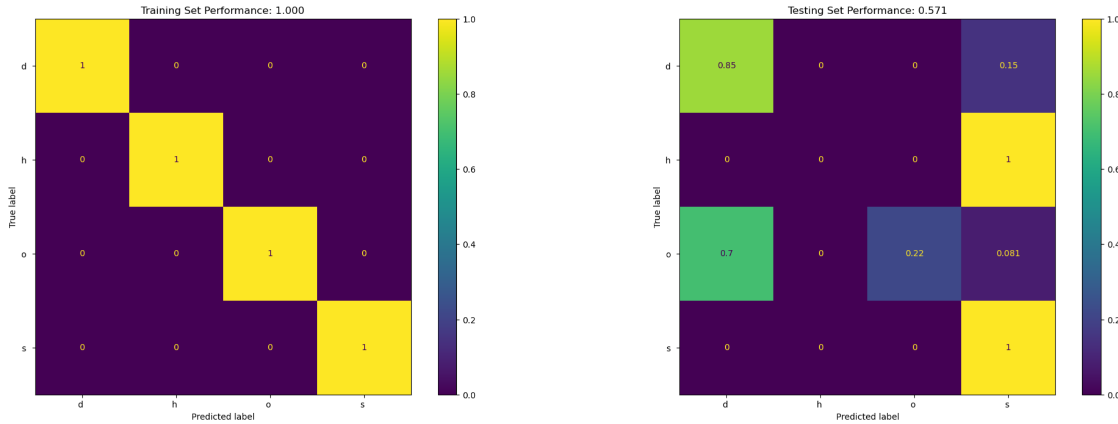

Testing Set Performance: 0.894
True label
Predicted label
d h o s
0.94 0 0.037 0.019
0 0.79 0 0.21
0.19 0 0.81 0
0 0.017 0 0.98


Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1


Testing Set Performance: 0.864
True label
Predicted label
d h o s
0.96 0 0.019 0.019
0 0.69 0 0.31
0.22 0 0.76 0.027
0 0.017 0 0.98


Training Set Performance: 1.000
True label
Predicted label
d h o s
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1


Testing Set Performance: 0.646
True label
Predicted label
d h o s
0.89 0 0 0.11
0 0 0 1
0.41 0 0.57 0.027
0 0 0 1




```
[ ]:
```

3 A Random Forest

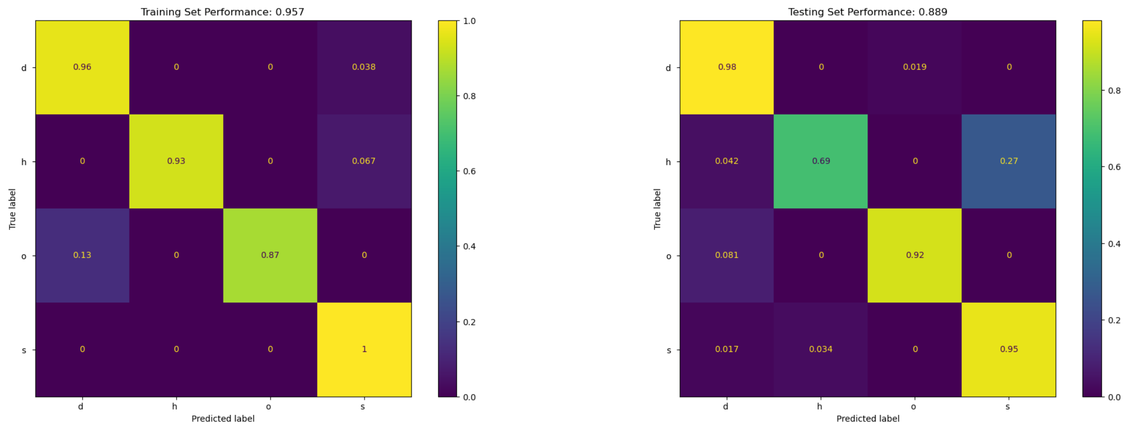
3.1 Before using Grid Search

```python
[15]: #helper function
def eval_model(model, X_train, Y_train, X_test, Y_test):
    fig = plt.figure(figsize=[25, 8])
    ax = fig.add_subplot(1, 2, 1)
    conf = ConfusionMatrixDisplay.from_estimator(model, X_train, Y_train,␣
 ↪normalize='true', ax=ax)
    conf.ax_.set_title('Training Set Performance: %1.3f' % (sum(model.
 ↪predict(X_train) == Y_train)/len(Y_train)));
    ax = fig.add_subplot(1, 2, 2)
    conf = ConfusionMatrixDisplay.from_estimator(model, X_test, Y_test,␣
 ↪normalize='true', ax=ax)
    conf.ax_.set_title('Testing Set Performance: %1.3f' % (sum(model.
 ↪predict(X_test) == Y_test)/len(Y_test)));
    print(classification_report(Y_test, model.predict(X_test)))
```

```python
[20]: # Simple model

rf = RandomForestClassifier(n_estimators=100, max_depth=4, random_state=0).
 ↪fit(X_train, Y_train)
eval_model(rf, X_train, Y_train, X_test, Y_test)
```

```
              precision    recall  f1-score   support

           d       0.90      0.98      0.94        54
           h       0.94      0.69      0.80        48
           o       0.97      0.92      0.94        37
```

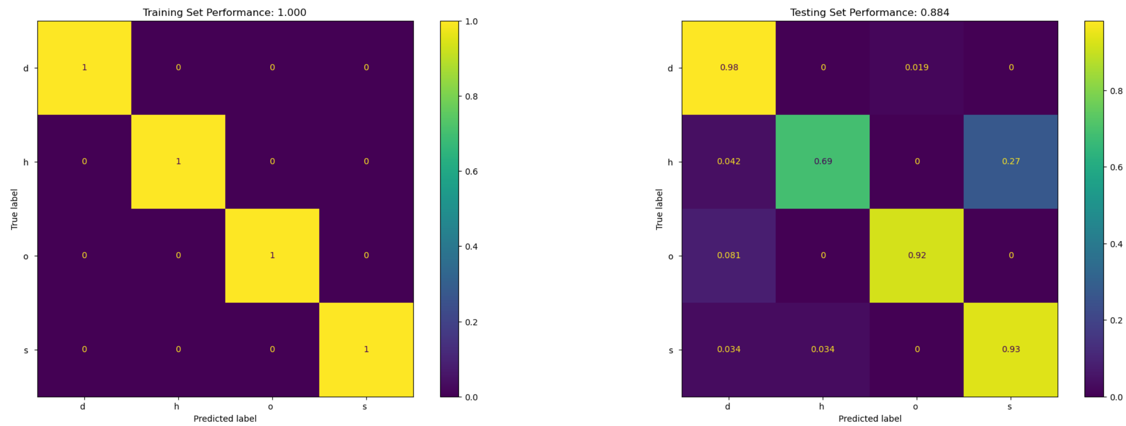
```
           s       0.81      0.95      0.87        59

    accuracy                           0.89       198
   macro avg       0.91      0.88      0.89       198
weighted avg       0.90      0.89      0.89       198
```



[17]:

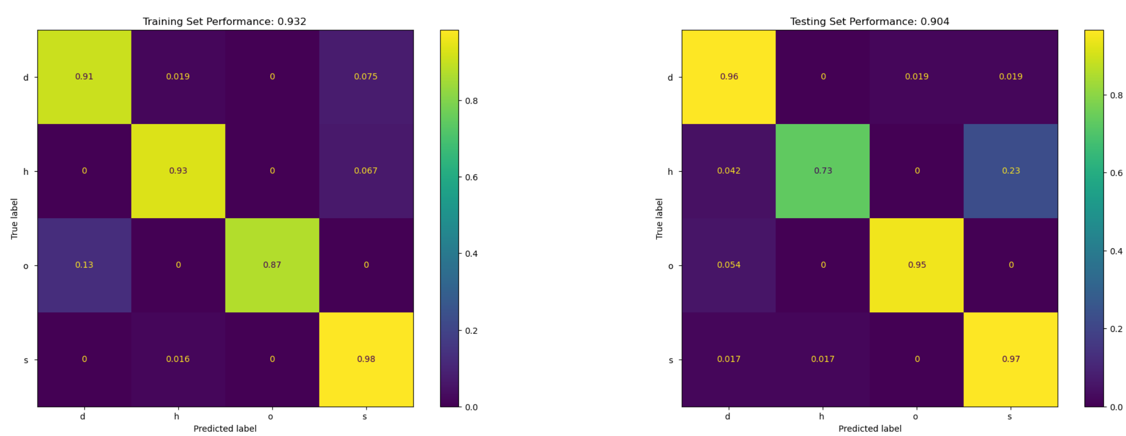
```
rf = RandomForestClassifier(n_estimators=100, max_depth=32, random_state=0).
 ↪fit(X_train, Y_train)
eval_model(rf, X_train, Y_train, X_test, Y_test)
```

```
              precision    recall  f1-score   support

           d       0.88      0.98      0.93        54
           h       0.94      0.69      0.80        48
           o       0.97      0.92      0.94        37
           s       0.81      0.93      0.87        59

    accuracy                           0.88       198
   macro avg       0.90      0.88      0.88       198
weighted avg       0.89      0.88      0.88       198
```




```
[18]: rf = RandomForestClassifier(n_estimators=100, max_depth=4, random_state=0,␣
       ↪class_weight='balanced_subsample').fit(X_train, Y_train)
      eval_model(rf, X_train, Y_train, X_test, Y_test)
```

```
              precision    recall  f1-score   support

           d       0.91      0.96      0.94        54
           h       0.97      0.73      0.83        48
           o       0.97      0.95      0.96        37
           s       0.83      0.97      0.89        59

    accuracy                           0.90       198
   macro avg       0.92      0.90      0.90       198
weighted avg       0.91      0.90      0.90       198
```



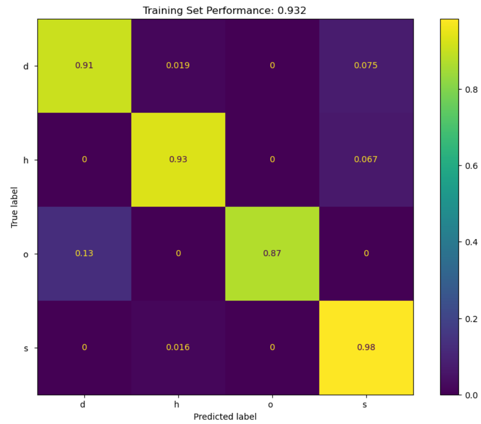
[ ]:
```
rf = RandomForestClassifier(n_estimators=150, max_depth=10, random_state=0,␣
↪class_weight='balanced_subsample').fit(X_train, Y_train)
eval_model(rf, X_train, Y_train, X_test, Y_test)
```

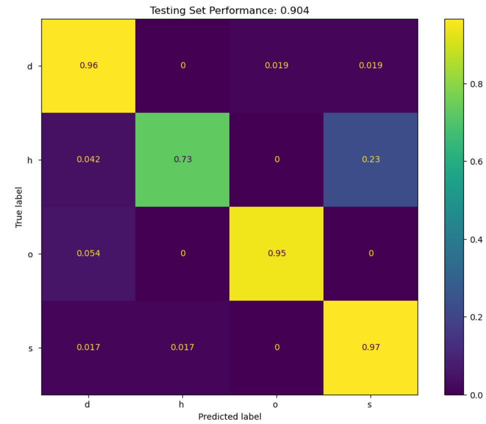
[11]:
```
# Define the parameter grid to search over
param_grid = {
    'n_estimators': [100, 150, 200],
    'max_depth': [1,2,3,4,5, 10, 15, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}

# Create the random forest classifier
rfc = RandomForestClassifier(random_state = 0)

# Perform the grid search without parallelization
grid_search = GridSearchCV(estimator = rfc, param_grid = param_grid,
                           cv = 3, n_jobs = -1)
grid_search.fit(X_train, Y_train)

# Print the best hyperparameters and corresponding score
print("Best hyperparameters:", grid_search.best_params_)
print("Best score:", grid_search.best_score_)
```

```
Best hyperparameters: {'max_depth': 4, 'min_samples_leaf': 1,
'min_samples_split': 2, 'n_estimators': 100}
Best score: 0.8700209643605871
```

[22]:
```
rf = RandomForestClassifier(n_estimators=100, max_depth=4, random_state=0,␣
↪class_weight='balanced_subsample').fit(X_train, Y_train)
eval_model(rf, X_train, Y_train, X_test, Y_test)
```

```
              precision    recall  f1-score   support

           d       0.91      0.96      0.94        54
           h       0.97      0.73      0.83        48
           o       0.97      0.95      0.96        37
           s       0.83      0.97      0.89        59

    accuracy                           0.90       198
   macro avg       0.92      0.90      0.90       198
weighted avg       0.91      0.90      0.90       198
```





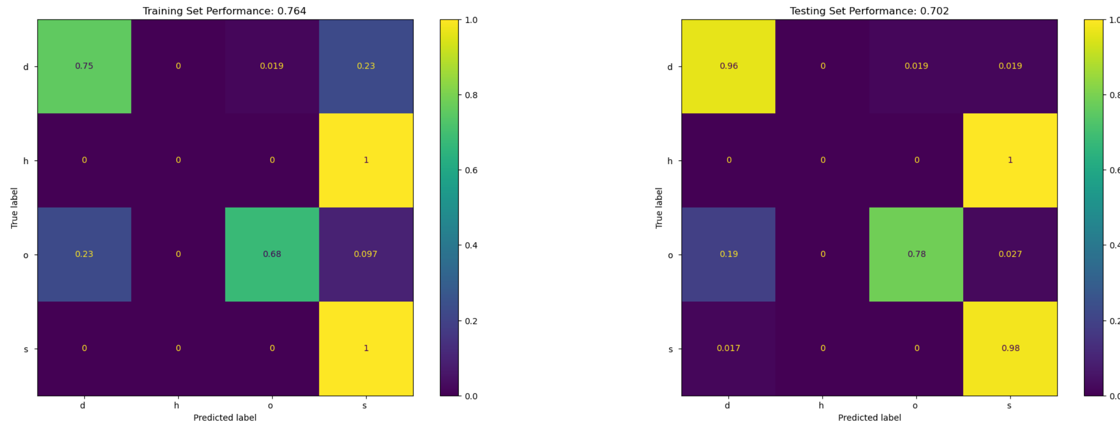
4 SVM

4.1 Before using Grid Search

[16]:
```
svm = SVC()
svm.fit(X_train, Y_train)
eval_model(svm, X_train, Y_train, X_test, Y_test)
```

```
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/opt/conda/lib/python3.10/site-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
              precision    recall  f1-score   support

           d       0.87      0.96      0.91        54
           h       0.00      0.00      0.00        48
           o       0.97      0.78      0.87        37
           s       0.54      0.98      0.69        59

    accuracy                           0.70       198
   macro avg       0.59      0.68      0.62       198
```

```
weighted avg       0.58      0.70      0.62       198
```



[12]:
```python
param_grid = [
  {'C': [0.1, 1, 10, 100], 'kernel': ['linear']},
  {'C': [0.1, 1, 10, 100], 'gamma': [0.1, 0.01, 0.001], 'kernel': ['rbf']},
  {'C': [0.1, 1, 10, 100], 'degree': [3, 4, 5], 'kernel': ['poly']},
 ]
svm = SVC(class_weight='balanced')
grid_search = GridSearchCV(svm, param_grid)
grid_search.fit(X_train, Y_train)
grid_search.cv_results_
```

[12]:
```
{'mean_fit_time': array([0.00255365, 0.00192237, 0.00210447, 0.00319586,
0.00327144,
        0.00341907, 0.00328131, 0.00214319, 0.00280757, 0.00281386,
        0.0018393 , 0.00205412, 0.00295577, 0.00173869, 0.00190101,
        0.00217338, 0.00254431, 0.00245509, 0.00244312, 0.00192933,
        0.00208459, 0.00194039, 0.00176697, 0.00199652, 0.00174699,
        0.00178952, 0.00189905, 0.00192962]),
 'std_fit_time': array([5.04428525e-04, 8.02416492e-05, 1.75586075e-04,
5.53846464e-04,
        1.11751608e-04, 1.15237799e-04, 1.02138965e-04, 1.28644921e-04,
        1.64495855e-04, 1.08085679e-04, 9.71735867e-05, 1.71281381e-04,
        3.32018964e-04, 2.15669952e-04, 8.11899247e-05, 2.06374902e-04,
        6.99513123e-05, 2.06668541e-04, 1.05236935e-04, 8.03241216e-05,
        2.32944034e-04, 1.44277293e-04, 1.41026264e-04, 5.97928623e-04,
        1.39393913e-04, 1.69289038e-04, 1.35861894e-04, 2.33769172e-04]),
 'mean_score_time': array([0.00077214, 0.00070996, 0.00067821, 0.0006763 ,
0.00089717,
        0.00093937, 0.00093231, 0.00072508, 0.00084496, 0.00080161,
        0.00075479, 0.00074515, 0.00083232, 0.00058765, 0.0007432 ,
```

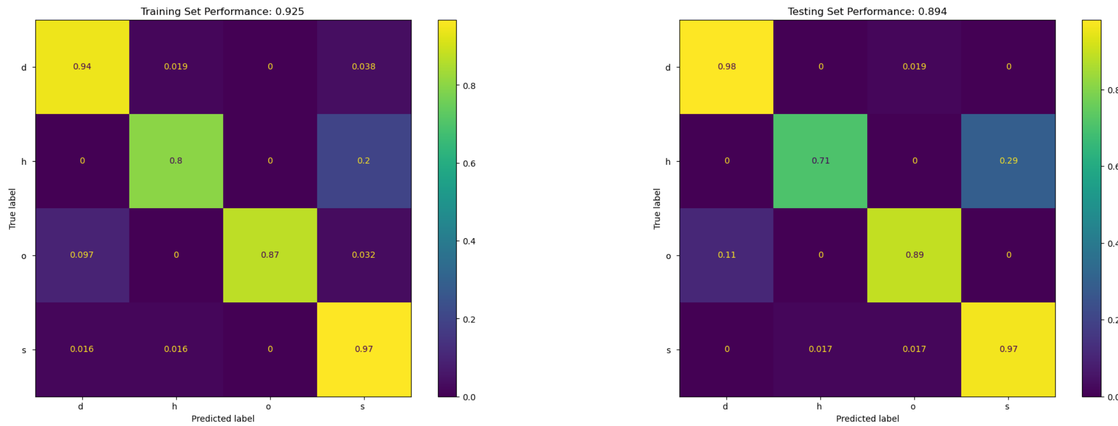
```
        0.00078759, 0.00072937, 0.00066767, 0.0007092 , 0.00067158,
        0.00059295, 0.00066786, 0.0006865 , 0.00058861, 0.00058331,
        0.00058641, 0.00060244, 0.00063205]),
 'std_score_time': array([6.28867770e-05, 1.79802388e-05, 3.88666455e-05,
2.03959256e-05,
        1.03472894e-04, 7.69853555e-05, 5.26216540e-05, 6.52276632e-05,
        5.96818424e-05, 2.57793885e-05, 1.31267995e-05, 5.76189392e-05,
        9.51966903e-05, 5.52265954e-05, 4.81500236e-05, 6.40778663e-05,
        3.85832868e-05, 4.25157487e-05, 2.42810800e-05, 1.10820218e-04,
        5.45089740e-05, 6.05687185e-05, 7.77461893e-05, 6.77077534e-05,
        2.48599134e-05, 8.01820646e-05, 4.33878320e-05, 3.61368957e-05]),
 'param_C': masked_array(data=[0.1, 1, 10, 100, 0.1, 0.1, 0.1, 1, 1, 1, 10, 10,
10,
                   100, 100, 100, 0.1, 0.1, 0.1, 1, 1, 1, 10, 10, 10, 100,
                   100, 100],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_kernel': masked_array(data=['linear', 'linear', 'linear', 'linear',
'rbf', 'rbf',
                   'rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf', 'rbf',
                   'rbf', 'rbf', 'poly', 'poly', 'poly', 'poly', 'poly',
                   'poly', 'poly', 'poly', 'poly', 'poly', 'poly', 'poly'],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_gamma': masked_array(data=[--, --, --, --, 0.1, 0.01, 0.001, 0.1, 0.01,
0.001,
                   0.1, 0.01, 0.001, 0.1, 0.01, 0.001, --, --, --, --, --,
                   --, --, --, --, --, --, --],
             mask=[ True,  True,  True,  True, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                    True,  True,  True,  True,  True,  True,  True,  True,
                    True,  True,  True,  True],
        fill_value='?',
             dtype=object),
 'param_degree': masked_array(data=[--, --, --, --, --, --, --, --, --, --, --,
--, --, --,
                   --, --, 3, 4, 5, 3, 4, 5, 3, 4, 5, 3, 4, 5],
             mask=[ True,  True,  True,  True,  True,  True,  True,  True,
                    True,  True,  True,  True,  True,  True,  True,  True,
```

```
                  False, False, False, False, False, False, False, False,
                  False, False, False, False],
       fill_value='?',
            dtype=object),
 'params': [{'C': 0.1, 'kernel': 'linear'},
  {'C': 1, 'kernel': 'linear'},
  {'C': 10, 'kernel': 'linear'},
  {'C': 100, 'kernel': 'linear'},
  {'C': 0.1, 'gamma': 0.1, 'kernel': 'rbf'},
  {'C': 0.1, 'gamma': 0.01, 'kernel': 'rbf'},
  {'C': 0.1, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 1, 'gamma': 0.1, 'kernel': 'rbf'},
  {'C': 1, 'gamma': 0.01, 'kernel': 'rbf'},
  {'C': 1, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 10, 'gamma': 0.1, 'kernel': 'rbf'},
  {'C': 10, 'gamma': 0.01, 'kernel': 'rbf'},
  {'C': 10, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 100, 'gamma': 0.1, 'kernel': 'rbf'},
  {'C': 100, 'gamma': 0.01, 'kernel': 'rbf'},
  {'C': 100, 'gamma': 0.001, 'kernel': 'rbf'},
  {'C': 0.1, 'degree': 3, 'kernel': 'poly'},
  {'C': 0.1, 'degree': 4, 'kernel': 'poly'},
  {'C': 0.1, 'degree': 5, 'kernel': 'poly'},
  {'C': 1, 'degree': 3, 'kernel': 'poly'},
  {'C': 1, 'degree': 4, 'kernel': 'poly'},
  {'C': 1, 'degree': 5, 'kernel': 'poly'},
  {'C': 10, 'degree': 3, 'kernel': 'poly'},
  {'C': 10, 'degree': 4, 'kernel': 'poly'},
  {'C': 10, 'degree': 5, 'kernel': 'poly'},
  {'C': 100, 'degree': 3, 'kernel': 'poly'},
  {'C': 100, 'degree': 4, 'kernel': 'poly'},
  {'C': 100, 'degree': 5, 'kernel': 'poly'}],
 'split0_test_score': array([0.72727273, 0.81818182, 0.84848485, 0.78787879,
0.54545455,
       0.33333333, 0.33333333, 0.78787879, 0.60606061, 0.36363636,
       0.81818182, 0.78787879, 0.60606061, 0.81818182, 0.81818182,
       0.78787879, 0.60606061, 0.63636364, 0.63636364, 0.72727273,
       0.75757576, 0.75757576, 0.81818182, 0.78787879, 0.78787879,
       0.81818182, 0.81818182, 0.81818182]),
 'split1_test_score': array([0.8125 , 0.8125 , 0.875  , 0.875  , 0.53125,
0.34375, 0.34375,
       0.78125, 0.65625, 0.34375, 0.8125 , 0.78125, 0.6875 , 0.84375,
       0.84375, 0.78125, 0.75   , 0.65625, 0.6875 , 0.84375, 0.84375,
       0.84375, 0.84375, 0.84375, 0.8125 , 0.875  , 0.875  , 0.875  ]),
 'split2_test_score': array([0.84375, 0.84375, 0.8125 , 0.8125 , 0.53125,
0.34375, 0.34375,
       0.84375, 0.8125 , 0.34375, 0.84375, 0.84375, 0.71875, 0.84375,
```

```
        0.84375, 0.875  , 0.6875 , 0.625  , 0.59375, 0.8125 , 0.78125,
        0.78125, 0.84375, 0.8125 , 0.78125, 0.84375, 0.84375, 0.84375]),
 'split3_test_score': array([0.8125 , 0.90625, 0.9375 , 0.90625, 0.5    , 0.3125
, 0.3125 ,
        0.90625, 0.75   , 0.1875 , 0.9375 , 0.875  , 0.75   , 0.9375 ,
        0.96875, 0.875  , 0.65625, 0.625  , 0.6875 , 0.78125, 0.8125 ,
        0.8125 , 0.90625, 0.9375 , 0.9375 , 0.96875, 0.96875, 0.96875]),
 'split4_test_score': array([0.8125 , 0.90625, 0.90625, 0.90625, 0.46875, 0.3125
, 0.3125 ,
        0.875  , 0.59375, 0.1875 , 0.90625, 0.875  , 0.59375, 0.90625,
        0.90625, 0.875  , 0.59375, 0.5    , 0.625  , 0.8125 , 0.8125 ,
        0.78125, 0.875  , 0.875  , 0.875  , 0.90625, 0.90625, 0.90625]),
 'mean_test_score': array([0.80170455, 0.85738636, 0.87594697, 0.85757576,
0.51534091,
        0.32916667, 0.32916667, 0.83882576, 0.68371212, 0.28522727,
        0.86363636, 0.83257576, 0.67121212, 0.86988636, 0.87613636,
        0.83882576, 0.65871212, 0.60852273, 0.64602273, 0.79545455,
        0.80151515, 0.79526515, 0.85738636, 0.85132576, 0.83882576,
        0.88238636, 0.88238636, 0.88238636]),
 'std_test_score': array([0.03913449, 0.04126268, 0.04356061, 0.04884969,
0.02763724,
        0.01412985, 0.01412985, 0.04855804, 0.08468472, 0.08012371,
        0.04969561, 0.0408819 , 0.06160808, 0.04457241, 0.0546637 ,
        0.04435378, 0.05685163, 0.05544942, 0.036629  , 0.03940577,
        0.02955152, 0.02987025, 0.03035389, 0.05210915, 0.05941176,
        0.05234062, 0.05234062, 0.05234062]),
 'rank_test_score': array([16,  9,  5,  8, 25, 26, 26, 12, 20, 28,  7, 15, 21,
6,  4, 12, 22,
        24, 23, 18, 17, 19,  9, 11, 12,  1,  1,  1], dtype=int32)}
```

[13]:
```
best_system = np.argmin(grid_search.cv_results_['rank_test_score'])
params = grid_search.cv_results_['params'][best_system]
print(params)
svm = SVC().set_params(**params)
svm.fit(X_train, Y_train)
eval_model(svm, X_train, Y_train, X_test, Y_test)
```

```
{'C': 100, 'degree': 3, 'kernel': 'poly'}
              precision    recall  f1-score   support

           d       0.93      0.98      0.95        54
           h       0.97      0.71      0.82        48
           o       0.94      0.89      0.92        37
           s       0.80      0.97      0.88        59

    accuracy                           0.89       198
   macro avg       0.91      0.89      0.89       198
weighted avg       0.90      0.89      0.89       198
```


Training Set Performance: 0.925
Testing Set Performance: 0.894
True label
Predicted label
d
h
o
s
0.94
0.019
0
0.038
0.8
0.2
0.097
0.87
0.032
0.016
0.97
0.98
0.71
0.29
0.11
0.89
0.017
0.0
0.2
0.4
0.6

[ ]:

CAB420_A1A_Q3_Template

April 18, 2023

1 CAB420 Assignment 1A Question 3: Template and Utilities Demo

Simon Denman (s.denman@qut.edu.au)

1.1 Overview

This notebook provides a quick demo and overview of the provided utility functions to help with Assignment 1A, Question 3.

It also implements the SVM that you are to compare against when responsing to the question.

1.2 Utility Functions

The following cell contains utility functions to: * Load the data * Vectorise the data * Plot images * Resize all images * Convert images to grayscale

These are provided to assist you in developing your solution.

[1]:
```
#
# Utility functions for CAB420, Assignment 1A, Q3
# Author: Simon Denman (s.denman@qut.edu.au)
#

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

from scipy.io import loadmat        # to load mat files
import matplotlib.pyplot as plt     # for plotting
import numpy as np                  # for reshaping, array manipulation
import cv2                          # for colour conversion
import tensorflow as tf             # for bulk image resize
from tensorflow.keras import layers


from tensorflow import keras

from tensorboard import notebook
from tensorflow.keras.utils import to_categorical
```

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import precision_score, recall_score, f1_score,␣
↪classification_report

import matplotlib.pyplot as plt
import seaborn as sns

import numpy
# Load data for Q3
#  train_path: path to training data mat file
#  test_path:  path to testing data mat file
#
#  returns:    arrays for training and testing X and Y data
#
def load_data(train_path, test_path):

    # load files
    train = loadmat(train_path)
    test = loadmat(test_path)

    # transpose, such that dimensions are (sample, width, height, channels),␣
↪and divide by 255.0
    train_X = np.transpose(train['train_X'], (3, 0, 1, 2)) / 255.0
    train_Y = train['train_Y']
    # change labels '10' to '0' for compatability with keras/tf. The label '10'␣
↪denotes the digit '0'
    train_Y[train_Y == 10] = 0
    train_Y = np.reshape(train_Y, -1)

    # transpose, such that dimensions are (sample, width, height, channels),␣
↪and divide by 255.0
    test_X = np.transpose(test['test_X'], (3, 0, 1, 2)) / 255.0
    test_Y = test['test_Y']
    # change labels '10' to '0' for compatability with keras/tf. The label '10'␣
↪denotes the digit '0'
    test_Y[test_Y == 10] = 0
    test_Y = np.reshape(test_Y, -1)

    # return loaded data
    return train_X, train_Y, test_X, test_Y

# vectorise an array of images, such that the shape is changed from {samples,␣
↪width, height, channels} to
# (samples, width * height * channels)
#   images: array of images to vectorise
```

```
#
#   returns: vectorised array of images
#
def vectorise(images):
    # use numpy's reshape to vectorise the data
    return np.reshape(images, [len(images), -1])

# Plot some images and their labels. Will plot the first 100 samples in a 10x10␣
↪grid
#  x: array of images, of shape (samples, width, height, channels)
#  y: labels of the images
#
def plot_images(x, y):
    fig = plt.figure(figsize=[15, 18])
    for i in range(100):
        ax = fig.add_subplot(10, 10, i + 1)
        ax.imshow(x[i,:])
        ax.set_title(y[i])
        ax.axis('off')

# Resize an array of images
#  images:   array of images, of shape (samples, width, height, channels)
#  new_size: tuple of the new size, (new_width, new_height)
#
#  returns:  resized array of images, (samples, new_width, new_height, channels)
#
def resize(images, new_size):
    # tensorflow has an image resize funtion that can do this in bulk
    # note the conversion back to numpy after the resize
    return tf.image.resize(images, new_size).numpy()

# Convert images to grayscale
#   images:  array of colour images to convert, of size (samples, width,␣
↪height, 3)
#
#   returns: array of converted images, of size (samples, width, height, 1)
#
def convert_to_grayscale(images):
    # storage for converted images
    gray = []
    # loop through images
    for i in range(len(images)):
        # convert each image using openCV
        gray.append(cv2.cvtColor(images[i,:], cv2.COLOR_BGR2GRAY))
    # pack converted list as an array and return
    return np.expand_dims(np.array(gray), axis = -1)
```

1.3 Utility Function Demonstration

The following presents a brief demonstration of the utility functions. These portions of code do not form part of the template, or solution, and could be commented out/removed.

1.3.1 Data Loading

Load the data, and visualise images.

[2]:
```
train_X, train_Y, test_X, test_Y = load_data('/home/n9894403/cab420/cab420␣
↪pracs/Q3/q3_train.mat', '/home/n9894403/cab420/cab420 pracs/Q3/q3_test.mat')

# check shape of data
print(train_X.shape)
print(train_Y.shape)
print(test_X.shape)
print(test_Y.shape)

# visualise images as a sanity check
plot_images(train_X, train_Y)
```

```
(1000, 32, 32, 3)
(1000,)
(10000, 32, 32, 3)
(10000,)
```

1.3.2 Vectorise Data

To train an SVM, each sample needs to be a vector rather than an image.

[3]:
```
train_vector_X = vectorise(train_X)
test_vector_X = vectorise(test_X)
print(train_vector_X.shape)
```

```
print(test_vector_X.shape)
```

```
(1000, 3072)
(10000, 3072)
```

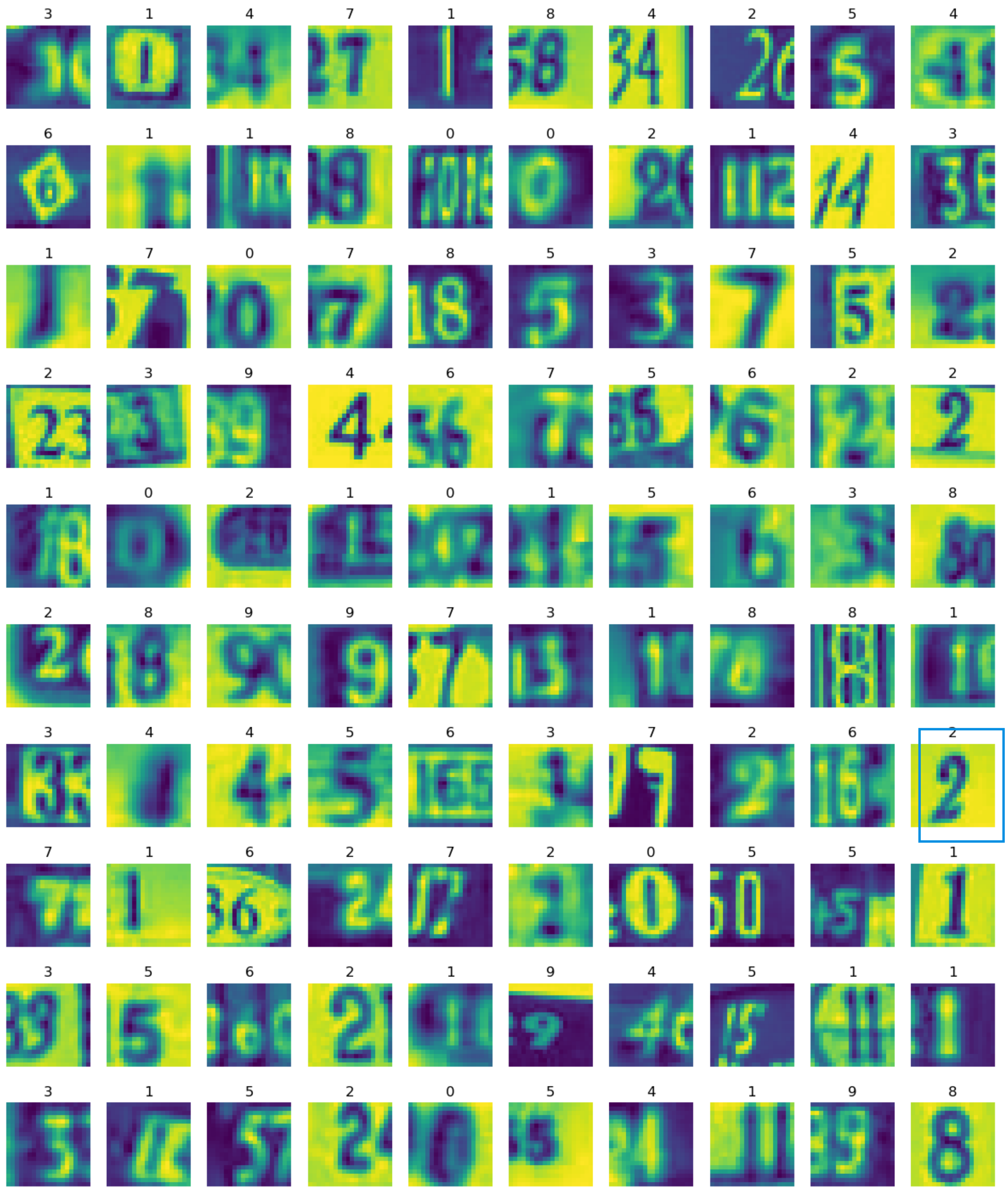
1.3.3 Conversion and Resizing

You may wish to either: * Resize images * Convert images to grayscale

Two functions are provided to do this, and can be used in combination as shown below.

[4]:
```
train_X_small = convert_to_grayscale(resize(train_X, (20, 20)))
print(train_X_small.shape)
plot_images(train_X_small, train_Y)
```

```
(1000, 20, 20, 1)
```



1.3.4 Measuring Time

There are a lot of ways to measure time in python. A simple one is to use `process_time` within the `time` package. This will simply measure the ellapsed process time in seconds. We can use ths to measure individual parts of our code as follows:

[5]:
```python
# import process_time
from time import process_time

# get a start time
time_1 = process_time()

# do some stuff, in this case we'll just load some data
train_X, train_Y, test_X, test_Y = load_data('/home/n9894403/cab420/cab420␣
↪pracs/Q3/q3_train.mat', '/home/n9894403/cab420/cab420 pracs/Q3/q3_test.mat')

# get the end time of our first lot of "stuff"
time_2 = process_time()

# do some other stuff
train_X_small = convert_to_grayscale(resize(train_X, (20, 20)))

# get the end time of our first lot of "stuff"
time_3 = process_time()

# the time it took to do "our stuff" is just the difference between the start␣
↪and end times
print('Time to load data:   %f seconds' % (time_2 - time_1))
print('Time to resize data: %f seconds' % (time_3 - time_2))
```

```
Time to load data:   0.358647 seconds
Time to resize data: 0.043993 seconds
```

1.4 Question 3 Template

The following provides a starting point for your solution. It trains the SVM that you are to compare your trained DCNNs against, and measures the time taken to train this SVM, and to perform inference with the train and test sets.

This does not measure the performance of the SVM - you will need to implement this as part of your solution.

[6]:
```python
from sklearn.svm import SVC
from time import process_time

# load data
train_X, train_Y, test_X, test_Y = load_data('/home/n9894403/cab420/cab420␣
↪pracs/Q3/q3_train.mat', '/home/n9894403/cab420/cab420 pracs/Q3/q3_test.mat')

# any resize, colour change, etc, would go here

# vectorise data
# If you do any resize, reshape, etc of the data prior to putting this into␣
↪your DCNN, change this code to
```

```
# vectorise that verison of the data. The same data should be used by all␣
↪models for a fair comparison; though
# you will only vectorise the data for the SVM (i.e. the DCNN will get the data␣
↪as images).
train_vector_X = vectorise(train_X)
test_vector_X = vectorise(test_X)

# train the SVM
svm_train_start = process_time()
svm = SVC(C = 1.0, kernel = 'linear').fit(train_vector_X, train_Y)
svm_train_end = process_time()
train_predictions = svm.predict(train_vector_X)
svm_train_pred_end = process_time()
test_predictions = svm.predict(test_vector_X)
svm_test_pred_end = process_time()

svm_train_time = svm_train_end - svm_train_start
svm_inference_train_time = svm_train_pred_end - svm_train_end
svm_inference_test_time = svm_test_pred_end - svm_train_pred_end
print('Training Time: %f\nInference Time (training set): %f\nInference Time␣
↪(testing set): %f' % \
      (svm_train_time, svm_inference_train_time, svm_inference_test_time))

# evaluate SVM


# develop, evaluate and compare DCNNs
```

```
Training Time: 3.220755
Inference Time (training set): 1.671767
Inference Time (testing set): 15.993651
```

1.5 VGG Model

[7]:
```
def CreateModel():

    inputs = keras.Input(shape=(32, 32, 3, ), name='img')

    # 3x3 conv block
    x = layers.Conv2D(filters=8, kernel_size=(3, 3), padding='same',␣
↪activation=None)(inputs)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(filters=8, kernel_size=(3, 3), padding='same',␣
↪activation=None)(x)
    # adding batch norm
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
```

```
    x = layers.MaxPool2D(pool_size=(2, 2))(x)

    # 3x3 conv block, increase filters
    x = layers.Conv2D(filters=16, kernel_size=(3, 3), padding='same',␣
↪activation=None)(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(filters=16, kernel_size=(3, 3), padding='same',␣
↪activation=None)(x)
    # adding batch norm
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(pool_size=(2, 2))(x)

    # 3x3 conv block, further increase filters
    x = layers.Conv2D(filters=32, kernel_size=(3, 3), padding='same',␣
↪activation=None)(x)
    x = layers.Activation('relu')(x)
    x = layers.Conv2D(filters=32, kernel_size=(3, 3), padding='same',␣
↪activation=None)(x)
    # adding batch norm
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)
    x = layers.MaxPool2D(pool_size=(2, 2))(x)

    # flatten layer
    x = layers.Flatten()(x)

    # dense layer, 256 neurons
    x = layers.Dense(512, activation=None)(x)
    # adding batch norm, note that I've changed the activation above to None to␣
↪place batch norm
    # before the activation function
    x = layers.BatchNormalization()(x)
    x = layers.Activation('relu')(x)

    # the output, one neuron for the cost, relu activation becuase the cost␣
↪must be positive
    outputs = layers.Dense(10, activation='softmax')(x)

    # build the model, and print a summary
    model_vgg = keras.Model(inputs=inputs, outputs=outputs, name='vgg')
    return model_vgg
```

[8]:
```
model_vgg_without_A = CreateModel()
model_vgg_without_A.summary()
```

```
Model: "vgg"
```

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 img (InputLayer)            [(None, 32, 32, 3)]       0

 conv2d (Conv2D)             (None, 32, 32, 8)         224

 activation (Activation)     (None, 32, 32, 8)         0

 conv2d_1 (Conv2D)           (None, 32, 32, 8)         584

 batch_normalization (BatchN  (None, 32, 32, 8)        32
 ormalization)

 activation_1 (Activation)   (None, 32, 32, 8)         0

 max_pooling2d (MaxPooling2D  (None, 16, 16, 8)        0
 )

 conv2d_2 (Conv2D)           (None, 16, 16, 16)        1168

 activation_2 (Activation)   (None, 16, 16, 16)        0

 conv2d_3 (Conv2D)           (None, 16, 16, 16)        2320

 batch_normalization_1 (Batc  (None, 16, 16, 16)       64
 hNormalization)

 activation_3 (Activation)   (None, 16, 16, 16)        0

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 16)         0
 2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 32)          4640

 activation_4 (Activation)   (None, 8, 8, 32)          0

 conv2d_5 (Conv2D)           (None, 8, 8, 32)          9248

 batch_normalization_2 (Batc  (None, 8, 8, 32)         128
 hNormalization)

 activation_5 (Activation)   (None, 8, 8, 32)          0

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 flatten (Flatten)           (None, 512)               0
```

```
 dense (Dense)               (None, 512)              262656

 batch_normalization_3 (Batc  (None, 512)             2048
 hNormalization)

 activation_6 (Activation)   (None, 512)              0

 dense_1 (Dense)             (None, 10)               5130

=================================================================
Total params: 288,242
Trainable params: 287,106
Non-trainable params: 1,136
_________________________________________________________________
```

[9]:

```
# convert the y-data to categoricals
import time
train_Y = to_categorical(train_Y, 10)
test_Y = to_categorical(test_Y, 10)

model_vgg_without_A.compile(
    # categorical cross entropy loss
    loss='categorical_crossentropy',
    # adam optimiser
    optimizer=keras.optimizers.Adam(),
    # compute the accuracy metric, in addition to the loss
    metrics=['accuracy'])
# train the model
# we'll capture the returned history object that will tell us about the␣
↪training performance

start_time = time.time()

history = model_vgg_without_A.fit(train_X, train_Y,
                batch_size=16,
                epochs=20,
                validation_data=(test_X, test_Y), verbose=True)

end_time = time.time()
training_time = end_time - start_time
print("Training time: ", training_time, " seconds")
```

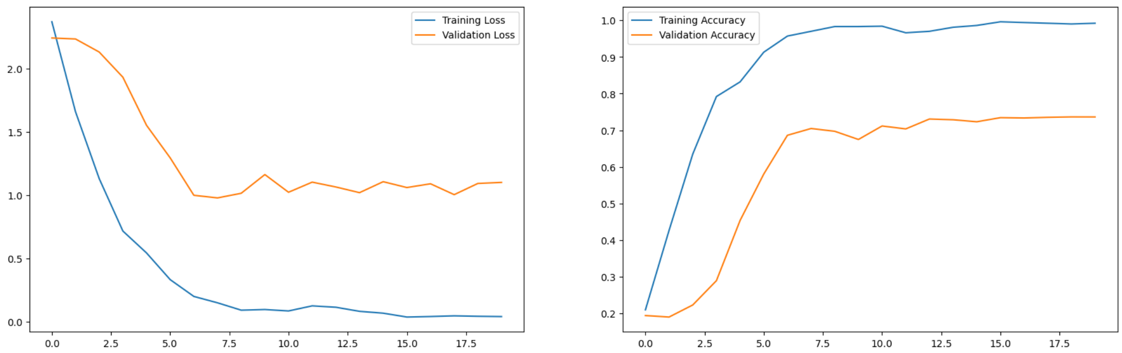
```
Epoch 1/20
63/63 [==============================] - 9s 43ms/step - loss: 2.3679 - accuracy:
0.2100 - val_loss: 2.2400 - val_accuracy: 0.1944
Epoch 2/20
63/63 [==============================] - 1s 24ms/step - loss: 1.6569 - accuracy:
```

```
0.4270 - val_loss: 2.2321 - val_accuracy: 0.1904
Epoch 3/20
63/63 [==============================] - 1s 23ms/step - loss: 1.1292 - accuracy:
0.6350 - val_loss: 2.1290 - val_accuracy: 0.2233
Epoch 4/20
63/63 [==============================] - 1s 24ms/step - loss: 0.7150 - accuracy:
0.7920 - val_loss: 1.9302 - val_accuracy: 0.2896
Epoch 5/20
63/63 [==============================] - 1s 24ms/step - loss: 0.5418 - accuracy:
0.8320 - val_loss: 1.5502 - val_accuracy: 0.4539
Epoch 6/20
63/63 [==============================] - 1s 24ms/step - loss: 0.3309 - accuracy:
0.9130 - val_loss: 1.2928 - val_accuracy: 0.5805
Epoch 7/20
63/63 [==============================] - 2s 24ms/step - loss: 0.1982 - accuracy:
0.9570 - val_loss: 0.9972 - val_accuracy: 0.6863
Epoch 8/20
63/63 [==============================] - 1s 23ms/step - loss: 0.1478 - accuracy:
0.9700 - val_loss: 0.9766 - val_accuracy: 0.7048
Epoch 9/20
63/63 [==============================] - 1s 24ms/step - loss: 0.0897 - accuracy:
0.9830 - val_loss: 1.0132 - val_accuracy: 0.6972
Epoch 10/20
63/63 [==============================] - 1s 23ms/step - loss: 0.0949 - accuracy:
0.9830 - val_loss: 1.1610 - val_accuracy: 0.6749
Epoch 11/20
63/63 [==============================] - 1s 23ms/step - loss: 0.0834 - accuracy:
0.9840 - val_loss: 1.0212 - val_accuracy: 0.7117
Epoch 12/20
63/63 [==============================] - 2s 24ms/step - loss: 0.1238 - accuracy:
0.9660 - val_loss: 1.1011 - val_accuracy: 0.7037
Epoch 13/20
63/63 [==============================] - 2s 24ms/step - loss: 0.1128 - accuracy:
0.9700 - val_loss: 1.0629 - val_accuracy: 0.7307
Epoch 14/20
63/63 [==============================] - 2s 25ms/step - loss: 0.0805 - accuracy:
0.9810 - val_loss: 1.0180 - val_accuracy: 0.7287
Epoch 15/20
63/63 [==============================] - 2s 24ms/step - loss: 0.0659 - accuracy:
0.9860 - val_loss: 1.1047 - val_accuracy: 0.7232
Epoch 16/20
63/63 [==============================] - 1s 23ms/step - loss: 0.0355 - accuracy:
0.9960 - val_loss: 1.0585 - val_accuracy: 0.7344
Epoch 17/20
63/63 [==============================] - 1s 24ms/step - loss: 0.0395 - accuracy:
0.9940 - val_loss: 1.0881 - val_accuracy: 0.7336
Epoch 18/20
63/63 [==============================] - 1s 23ms/step - loss: 0.0448 - accuracy:
```

```
0.9920 - val_loss: 1.0023 - val_accuracy: 0.7353
Epoch 19/20
63/63 [==============================] - 1s 24ms/step - loss: 0.0412 - accuracy:
0.9900 - val_loss: 1.0908 - val_accuracy: 0.7364
Epoch 20/20
63/63 [==============================] - 2s 25ms/step - loss: 0.0392 - accuracy:
0.9920 - val_loss: 1.0990 - val_accuracy: 0.7363
Training time:  37.05158185958862  seconds
```

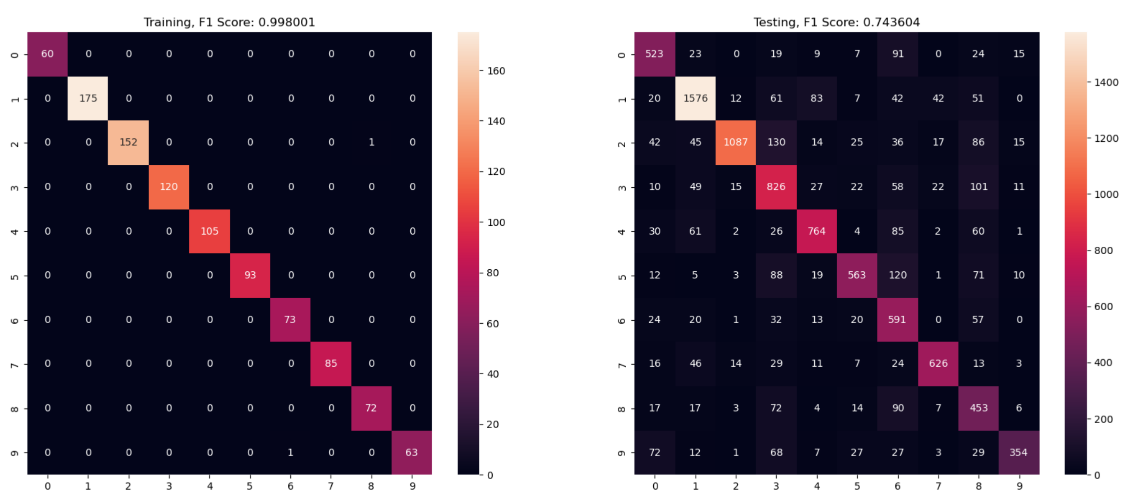
[10]:
```
fig = plt.figure(figsize=[20, 6])
ax = fig.add_subplot(1, 2, 1)
ax.plot(history.history['loss'], label="Training Loss")
ax.plot(history.history['val_loss'], label="Validation Loss")
ax.legend()

ax = fig.add_subplot(1, 2, 2)
ax.plot(history.history['accuracy'], label="Training Accuracy")
ax.plot(history.history['val_accuracy'], label="Validation Accuracy")
ax.legend()
```

[10]: <matplotlib.legend.Legend at 0x7f871b873c40>



[11]:
```
def eval_model(model, train, train_y, test, test_y):
    fig = plt.figure(figsize=[20, 8])

    ax = fig.add_subplot(1, 2, 1)
    # predict on the training set
    pred = model.predict(train, verbose=False);
    # get indexes for the predictions and ground truth, this is converting back
 ↪from a one-hot representation
    # to a single index
    indexes = tf.argmax(pred, axis=1)
    gt_idx = tf.argmax(train_y, axis=1)
```

```
    # plot the confusion matrix, I'm using tensorflow and seaborn here, but you could use
    # sklearn as well
    confusion_mtx = tf.math.confusion_matrix(gt_idx, indexes)
    sns.heatmap(confusion_mtx, xticklabels=range(10), yticklabels=range(10),
            annot=True, fmt='g', ax=ax)
    # set the title to the F1 scope
    ax.set_title('Training, F1 Score: %f' % f1_score(gt_idx, indexes, average='weighted'))

    # repeat visualisation for the test set
    ax = fig.add_subplot(1, 2, 2)
    pred = model.predict(test, verbose=False);
    indexes = tf.argmax(pred, axis=1)
    gt_idx = tf.argmax(test_y, axis=1)

    confusion_mtx = tf.math.confusion_matrix(gt_idx, indexes)
    sns.heatmap(confusion_mtx, xticklabels=range(10), yticklabels=range(10),
            annot=True, fmt='g', ax=ax)
    ax.set_title('Testing, F1 Score: %f' % f1_score(gt_idx, indexes, average='weighted'))

eval_model(model_vgg_without_A, train_X, train_Y, test_X, test_Y)
```



[21]:
```
##data augmentation

datagen = ImageDataGenerator(rotation_range=8,
                             zoom_range=[0.95, 1.05],
                             height_shift_range=0.10,
                             shear_range=0.15)
```

[13]:

```
batch = datagen.flow(train_X, train_Y, batch_size=100)
fig = plt.figure(figsize=[32, 32])
for i,img in enumerate(batch[0][0]):
    ax = fig.add_subplot(10, 10, i + 1)
    ax.imshow(img[:,:,:])
```

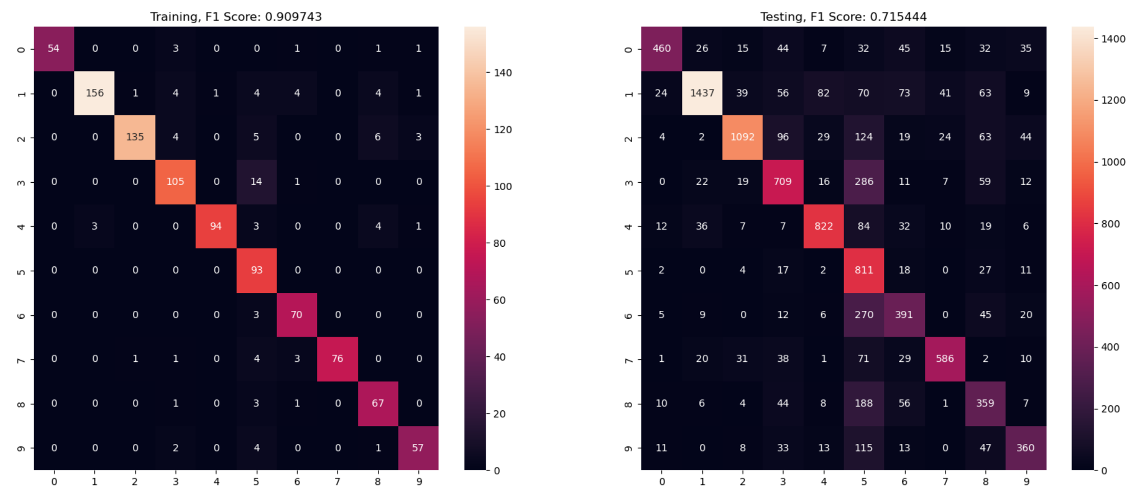
[22]:

```
model_vgg_with_A = CreateModel()
model_vgg_with_A.compile(
    # categorical cross entropy loss
    loss='categorical_crossentropy',
    # adam optimiser
    optimizer=keras.optimizers.Adam(),
    # compute the accuracy metric, in addition to the loss
```

```
    metrics=['accuracy'])
# train the model
# we'll capture the returned history object that will tell us about the␣
↪training performance

import time
start_time = time.time()
history = model_vgg_with_A.fit(datagen.flow(train_X, train_Y, batch_size=16),
                     steps_per_epoch=1000 // 16,
                    epochs=20,
                    validation_data=(test_X, test_Y), verbose=False)
end_time = time.time()
training_time = end_time - start_time
print("Training time: ", training_time, " seconds")
```

```
Training time:  29.4599711894989  seconds
```

[15]:
```
fig = plt.figure(figsize=[20, 6])
ax = fig.add_subplot(1, 2, 1)
ax.plot(history.history['loss'], label="Training Loss")
ax.plot(history.history['val_loss'], label="Validation Loss")
ax.legend()

ax = fig.add_subplot(1, 2, 2)
ax.plot(history.history['accuracy'], label="Training Accuracy")
ax.plot(history.history['val_accuracy'], label="Validation Accuracy")
ax.legend()

eval_model(model_vgg_with_A, train_X, train_Y, test_X, test_Y)
```



[16]:
```python
### time comparison

import time
def eval_model_nondl(model, X_train, Y_train, X_test, Y_test, verbose = False):

    train_pred_start = time.process_time()
    pred = model.predict(X_train)
    train_pred_end = time.process_time()

    if (verbose):
        fig = plt.figure(figsize=[25, 8])
        ax = fig.add_subplot(1, 2, 1)
        conf = ConfusionMatrixDisplay.from_estimator(model, X_train, Y_train,␣
↪normalize=None, xticks_rotation='vertical', ax=ax)

        conf.ax_.set_title('Training Set Performance: ' + str(sum(pred ==␣
↪Y_train)/len(Y_train)));

    test_pred_start = time.process_time()
    pred = model.predict(X_test)
    test_pred_end = time.process_time()
    test_acc = sum(pred == Y_test)/len(Y_test)

    if (verbose):
        ax = fig.add_subplot(1, 2, 2)
        conf = ConfusionMatrixDisplay.from_estimator(model, X_test, Y_test,␣
↪normalize=None, xticks_rotation='vertical', ax=ax)
        conf.ax_.set_title('Testing Set Performance: ' + str(test_acc));

        print(classification_report(y_test, pred))
```

```
    return (train_pred_end - train_pred_start), (test_pred_end -␣
 ↪test_pred_start), test_acc
def train_and_eval_nondl(model,x_train, y_train, x_test, y_test, verbose =␣
 ↪False):

    train_start = time.process_time()
    model.fit(x_train, y_train)
    train_end = time.process_time()
    train_time = train_end - train_start

    pred_train_time, pred_test_time, acc = eval_model_nondl(model, x_train,␣
 ↪y_train, x_test, y_test, verbose)

    return train_time, pred_train_time, pred_test_time, acc
```

[17]:
```
#train_and_eval_nondl(model_vgg_with_A, train_X, train_Y, test_X, test_Y,␣
 ↪verbose = False)
#(model_vgg_without_A, train_X, train_Y, test_X, test_Y
```

[18]:
```
#train_and_eval_nondl(model_vgg_without_A, train_X, train_Y, test_X, test_Y,␣
 ↪verbose = False)
```

[19]:
```
# your code for predicting on test set here

start_time = time.time()

test_predictions = model_vgg_without_A.predict(test_X)

end_time = time.time()
inference_time = end_time - start_time
print("Inference time: ", inference_time, " seconds")

test_acc = np.mean(np.argmax(test_predictions, axis=1) == np.argmax(test_Y,␣
 ↪axis=1))
print('Test set accuracy:', test_acc)



############
```

```
313/313 [==============================] - 1s 2ms/step
Inference time:  1.0527641773223877  seconds
Test set accuracy: 0.7363
```

[20]:
```
from sklearn.metrics import accuracy_score
test_generator = datagen.flow(test_X, test_Y, batch_size=32, shuffle=False)
```

```python
augmented_test_predictions = model_vgg_without_A.predict(test_generator)
augmented_test_accuracy = accuracy_score(np.argmax(augmented_test_predictions,
    axis=1), np.argmax(test_Y, axis=1))
print("Augmented Test Accuracy: ", augmented_test_accuracy)
```

```
313/313 [==============================] - 5s 15ms/step
Augmented Test Accuracy:  0.6361
```