ld-and-apply-multilayer-perceptron

March 26, 2024

Name | Jomarie Dupaya **Section** | CPE32S3 **Date Performed**: | 2/3/2024 **Date Submitted**: | 2/3/2024 **Instructor**: | Engr. Roman M. Richard

Title: Banana Quality **Link To Origin of the Dataset:** https://www.kaggle.com/datasets/l3llff/banana/data **Link to Spreadsheet Dataset:**https://docs.google.com/spreadsheets/d/16FdIZE2I35vgcOtTnyA8mA-IMvdjWJH_VFErJhDr79E/edit?usp=sharing

Based on the found dataset the problem I am trying to solve is to know the quality of the banana based on the different features of the fruit.

On this assignment I will perform both of the MLP using SKlearn MLPClassifier and TensorFlow keras, for benchmarking.

#**MultiLayer Perceptron Using Sklearn**

#####**Loading, processing, and cleaning the data for the 1st model**

[554]:
```python
#Import and load the dataset
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.neural_network import MLPClassifier
banana = pd.read_csv("/content/banana_quality.csv")
print(banana.columns)
```

```
Index(['Size', 'Weight', 'Sweetness', 'Softness', 'HarvestTime', 'Ripeness',
       'Acidity', 'Quality'],
      dtype='object')
```

[555]:
```python
banana.head()
```

[555]:
```
       Size    Weight  Sweetness  Softness  HarvestTime  Ripeness   Acidity  \
0 -1.924968  0.468078   3.077832 -1.472177     0.294799  2.435570  0.271290
1 -2.409751  0.486870   0.346921 -2.495099    -0.892213  2.067549  0.307325
2 -0.357607  1.483176   1.568452 -2.645145    -0.647267  3.090643  1.427322
3 -0.868524  1.566201   1.889605 -1.273761    -1.006278  1.873001  0.477862
4  0.651825  1.319199  -0.022459 -1.209709    -1.430692  1.078345  2.812442

  Quality
0    Good
```

```
1    Good
2    Good
3    Good
4    Good
```

[556]:
```
banana.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Size         8000 non-null   float64
 1   Weight       8000 non-null   float64
 2   Sweetness    8000 non-null   float64
 3   Softness     8000 non-null   float64
 4   HarvestTime  8000 non-null   float64
 5   Ripeness     8000 non-null   float64
 6   Acidity      8000 non-null   float64
 7   Quality      8000 non-null   object
dtypes: float64(7), object(1)
memory usage: 500.1+ KB
```

[557]:
```
from sklearn import preprocessing

# Creating labelEncoder
le = preprocessing.LabelEncoder()

# Converting string labels into numbers.
banana['Quality']=le.fit_transform(banana['Quality'])
```

[558]:
```
banana.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Size         8000 non-null   float64
 1   Weight       8000 non-null   float64
 2   Sweetness    8000 non-null   float64
 3   Softness     8000 non-null   float64
 4   HarvestTime  8000 non-null   float64
 5   Ripeness     8000 non-null   float64
 6   Acidity      8000 non-null   float64
 7   Quality      8000 non-null   int64
dtypes: float64(7), int64(1)
```

memory usage: 500.1 KB

[559]:
```
banana.describe()
```

[559]:
```
              Size       Weight    Sweetness     Softness  HarvestTime  \
count  8000.000000  8000.000000  8000.000000  8000.000000  8000.000000
mean     -0.747802    -0.761019    -0.770224    -0.014441    -0.751288
std       2.136023     2.015934     1.948455     2.065216     1.996661
min      -7.998074    -8.283002    -6.434022    -6.959320    -7.570008
25%      -2.277651    -2.223574    -2.107329    -1.590458    -2.120659
50%      -0.897514    -0.868659    -1.020673     0.202644    -0.934192
75%       0.654216     0.775491     0.311048     1.547120     0.507326
max       7.970800     5.679692     7.539374     8.241555     6.293280

          Ripeness      Acidity      Quality
count  8000.000000  8000.000000  8000.000000
mean      0.781098     0.008725     0.500750
std       2.114289     2.293467     0.500031
min      -7.423155    -8.226977     0.000000
25%      -0.574226    -1.629450     0.000000
50%       0.964952     0.098735     1.000000
75%       2.261650     1.682063     1.000000
max       7.249034     7.411633     1.000000
```

Remarks: The data is checked, verified, and columns filled with data that will be used for the model.

#####**Implementing Feature Importance**

[ ]:
```
#Separate features and target variable
X = banana.drop(['Acidity', 'Quality','Sweetness', 'Size'], axis=1)
y = banana['Quality']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

#Feature importance
#Train a Random Forest classifier
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)

#Get feature importances
feature_importances = rf.feature_importances_

#Create DataFrame to display feature importances
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance':␣
 ↪feature_importances})
```

```
feature_importance_df = feature_importance_df.sort_values(by='Importance',␣
↪ascending=False)

print("Feature importance:")
print(feature_importance_df)
```

```
Feature importance:
      Feature  Importance
2  HarvestTime    0.292866
0       Weight    0.261915
1     Softness    0.226521
3     Ripeness    0.218698
```

Remarks: Using random forest classifier to get the feature importance of the dataset to limit the data that will used to connect with the target variable which will can improve the modelling.

#####**Removing Outliers**

[ ]:
```
#Remove Outliers
Q1 = X.quantile(0.25)
Q3 = X.quantile(0.75)
IQR = Q3 - Q1
threshold = 1.5 * IQR
outliers = (X < (Q1 - threshold)) | (X > (Q3 + threshold))
cleaned_df = X[~outliers.any(axis=1)]

#2nd Outlier removal
Q1_2 = cleaned_df.quantile(0.25)
Q3_2 = cleaned_df.quantile(0.75)
IQR_2 = Q3_2 - Q1_2
threshold_2 = 1.5 * IQR_2
outliers_2 = (cleaned_df < (Q1_2 - threshold_2)) | (cleaned_df > (Q3_2 +␣
↪threshold_2))
cleaned_df2 = cleaned_df[~outliers_2.any(axis=1)]
```

[ ]:
```
cleaned_df2.describe()
```

[ ]:

	Weight	Softness	HarvestTime	Ripeness
count	7859.000000	7859.000000	7859.000000	7859.000000
mean	-0.742800	-0.028938	-0.780937	0.823255
std	2.009928	2.058572	1.937883	2.050002
min	-6.609340	-6.290912	-6.002344	-4.749013
25%	-2.215542	-1.618966	-2.127844	-0.534235
50%	-0.848095	0.179909	-0.957602	0.986723
75%	0.796848	1.543462	0.463344	2.276937
max	5.184198	6.124278	4.333418	6.490461

Remarks: Processing and removing some of the outliers, to lessen the difference of the 75% and maximum count of the dataframe to improve the performance and reliability of data.

#####Standardizing the Data

```
[ ]: #Standardized the data
scaler = StandardScaler()
scaleddf = scaler.fit_transform(cleaned_df2)
pd.DataFrame(scaleddf, columns=cleaned_df2.columns).describe()
```

[ ]:

	Weight	Softness	HarvestTime	Ripeness
count	7.859000e+03	7859.000000	7.859000e+03	7.859000e+03
mean	-5.786326e-17	0.000000	8.679489e-17	-5.786326e-17
std	1.000064e+00	1.000064	1.000064e+00	1.000064e+00
min	-2.918967e+00	-3.042095	-2.694559e+00	-2.718350e+00
25%	-7.327807e-01	-0.772443	-6.950850e-01	-6.622321e-01
50%	-5.239083e-02	0.101459	-9.116975e-02	7.974532e-02
75%	7.660702e-01	0.763879	6.421232e-01	7.091577e-01
max	2.949049e+00	2.989260	2.639314e+00	2.764664e+00

Remarks: Cleaning and processing more of the data to bring all of the important features to a common scale, which will lessen the dominance of the features during model training.

#####Normalizing the Data

```
[ ]: #Normalized the data
min_max_scaler = MinMaxScaler()
normalized_X = min_max_scaler.fit_transform(scaleddf)
pd.DataFrame(normalized_X, columns=cleaned_df2.columns).describe()
```

[ ]:

	Weight	Softness	HarvestTime	Ripeness
count	7859.000000	7859.000000	7859.000000	7859.000000
mean	0.497437	0.504380	0.505179	0.495777
std	0.170426	0.165811	0.187493	0.182393
min	0.000000	0.000000	0.000000	0.000000
25%	0.372560	0.376309	0.374863	0.374998
50%	0.488509	0.521202	0.488086	0.510321
75%	0.627987	0.631031	0.625565	0.625114
max	1.000000	1.000000	1.000000	1.000000

```
[ ]: print("Shape of normalized_X:", normalized_X.shape)
print("Shape of y:", y.shape)
y = y[:normalized_X.shape[0]]
```

```
Shape of normalized_X: (7859, 4)
Shape of y: (7859,)
```

Remarks: Normalizing the data to further shrink the scale of features on almost the same level which is to ensure every feature is proportional to the model training process.

#####Creating and Training the MLPClassifier Model

```
[ ]: #Split the cleaned and scaled/normalized data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(normalized_X, y,␣
 ↪test_size=0.3, random_state=42)

#Create MLPClassifier model
clf = MLPClassifier(hidden_layer_sizes=(12, 10), max_iter=300, verbose=True)
```

Remarks: After processing and cleaning the data,it is now used to split and create the model using the MLPClassifier from SKlearn to determine if the model can learn to distinguish target quality using the features that are input to the model.

```
[ ]: # Train the model
clf.fit(X_train, y_train)
```

```
Iteration 1, loss = 0.68784842
Iteration 2, loss = 0.68442915
Iteration 3, loss = 0.68064108
Iteration 4, loss = 0.67612452
Iteration 5, loss = 0.67021507
Iteration 6, loss = 0.66200898
Iteration 7, loss = 0.64875267
Iteration 8, loss = 0.62732553
Iteration 9, loss = 0.60464666
Iteration 10, loss = 0.58439580
Iteration 11, loss = 0.56536137
Iteration 12, loss = 0.54798694
Iteration 13, loss = 0.53286170
Iteration 14, loss = 0.51949268
Iteration 15, loss = 0.50895628
Iteration 16, loss = 0.49720132
Iteration 17, loss = 0.48790579
Iteration 18, loss = 0.47970848
Iteration 19, loss = 0.47210711
Iteration 20, loss = 0.46602550
Iteration 21, loss = 0.45984001
Iteration 22, loss = 0.45484465
Iteration 23, loss = 0.45090468
Iteration 24, loss = 0.44777250
Iteration 25, loss = 0.44423884
Iteration 26, loss = 0.44222026
Iteration 27, loss = 0.44018198
Iteration 28, loss = 0.43833434
Iteration 29, loss = 0.43756500
Iteration 30, loss = 0.43554702
Iteration 31, loss = 0.43601799
Iteration 32, loss = 0.43373416
Iteration 33, loss = 0.43364188
Iteration 34, loss = 0.43301609
```

```
Iteration 35, loss = 0.43359337
Iteration 36, loss = 0.43232541
Iteration 37, loss = 0.43143767
Iteration 38, loss = 0.43107498
Iteration 39, loss = 0.43129368
Iteration 40, loss = 0.43007507
Iteration 41, loss = 0.43067266
Iteration 42, loss = 0.43065942
Iteration 43, loss = 0.42943990
Iteration 44, loss = 0.42929806
Iteration 45, loss = 0.42922546
Iteration 46, loss = 0.42873942
Iteration 47, loss = 0.42917096
Iteration 48, loss = 0.42700979
Iteration 49, loss = 0.42564960
Iteration 50, loss = 0.42420808
Iteration 51, loss = 0.42281263
Iteration 52, loss = 0.42110391
Iteration 53, loss = 0.41931842
Iteration 54, loss = 0.41733482
Iteration 55, loss = 0.41647149
Iteration 56, loss = 0.41331967
Iteration 57, loss = 0.41112892
Iteration 58, loss = 0.41048797
Iteration 59, loss = 0.40759523
Iteration 60, loss = 0.40500424
Iteration 61, loss = 0.40316241
Iteration 62, loss = 0.40078013
Iteration 63, loss = 0.39908566
Iteration 64, loss = 0.39650013
Iteration 65, loss = 0.39426509
Iteration 66, loss = 0.39247028
Iteration 67, loss = 0.39058562
Iteration 68, loss = 0.38841059
Iteration 69, loss = 0.38763445
Iteration 70, loss = 0.38497104
Iteration 71, loss = 0.38290412
Iteration 72, loss = 0.38235556
Iteration 73, loss = 0.37987142
Iteration 74, loss = 0.37976579
Iteration 75, loss = 0.37624256
Iteration 76, loss = 0.37468556
Iteration 77, loss = 0.37360977
Iteration 78, loss = 0.37213363
Iteration 79, loss = 0.37063465
Iteration 80, loss = 0.36912461
Iteration 81, loss = 0.36852910
Iteration 82, loss = 0.36785381
```

```
Iteration 83, loss = 0.36510039
Iteration 84, loss = 0.36396322
Iteration 85, loss = 0.36265580
Iteration 86, loss = 0.36139031
Iteration 87, loss = 0.36106396
Iteration 88, loss = 0.35982503
Iteration 89, loss = 0.35851139
Iteration 90, loss = 0.35718000
Iteration 91, loss = 0.35668410
Iteration 92, loss = 0.35583605
Iteration 93, loss = 0.35480381
Iteration 94, loss = 0.35441207
Iteration 95, loss = 0.35331236
Iteration 96, loss = 0.35203095
Iteration 97, loss = 0.35267694
Iteration 98, loss = 0.35053733
Iteration 99, loss = 0.35053508
Iteration 100, loss = 0.34913964
Iteration 101, loss = 0.34946256
Iteration 102, loss = 0.34810297
Iteration 103, loss = 0.34794287
Iteration 104, loss = 0.34697996
Iteration 105, loss = 0.34664263
Iteration 106, loss = 0.34584676
Iteration 107, loss = 0.34590629
Iteration 108, loss = 0.34464405
Iteration 109, loss = 0.34501039
Iteration 110, loss = 0.34423689
Iteration 111, loss = 0.34318609
Iteration 112, loss = 0.34326050
Iteration 113, loss = 0.34260084
Iteration 114, loss = 0.34163433
Iteration 115, loss = 0.34320158
Iteration 116, loss = 0.34144960
Iteration 117, loss = 0.34031994
Iteration 118, loss = 0.34111794
Iteration 119, loss = 0.34012015
Iteration 120, loss = 0.33946081
Iteration 121, loss = 0.33915845
Iteration 122, loss = 0.33921308
Iteration 123, loss = 0.33883320
Iteration 124, loss = 0.33814630
Iteration 125, loss = 0.33831010
Iteration 126, loss = 0.33790858
Iteration 127, loss = 0.33665527
Iteration 128, loss = 0.33708414
Iteration 129, loss = 0.33684795
Iteration 130, loss = 0.33638566
```

```
Iteration 131, loss = 0.33601348
Iteration 132, loss = 0.33613286
Iteration 133, loss = 0.33568910
Iteration 134, loss = 0.33539689
Iteration 135, loss = 0.33464698
Iteration 136, loss = 0.33420348
Iteration 137, loss = 0.33414854
Iteration 138, loss = 0.33392044
Iteration 139, loss = 0.33375722
Iteration 140, loss = 0.33390327
Iteration 141, loss = 0.33307690
Iteration 142, loss = 0.33429114
Iteration 143, loss = 0.33283524
Iteration 144, loss = 0.33248081
Iteration 145, loss = 0.33258001
Iteration 146, loss = 0.33209847
Iteration 147, loss = 0.33195002
Iteration 148, loss = 0.33437925
Iteration 149, loss = 0.33175457
Iteration 150, loss = 0.33203084
Iteration 151, loss = 0.33092195
Iteration 152, loss = 0.33236932
Iteration 153, loss = 0.33074630
Iteration 154, loss = 0.33071630
Iteration 155, loss = 0.33069609
Iteration 156, loss = 0.33205292
Iteration 157, loss = 0.33153726
Iteration 158, loss = 0.33020127
Iteration 159, loss = 0.32960556
Iteration 160, loss = 0.32976250
Iteration 161, loss = 0.32949684
Iteration 162, loss = 0.32854254
Iteration 163, loss = 0.32864994
Iteration 164, loss = 0.32849620
Iteration 165, loss = 0.32872333
Iteration 166, loss = 0.32796862
Iteration 167, loss = 0.32868359
Iteration 168, loss = 0.32723486
Iteration 169, loss = 0.32810848
Iteration 170, loss = 0.32722936
Iteration 171, loss = 0.32694740
Iteration 172, loss = 0.32628417
Iteration 173, loss = 0.32691805
Iteration 174, loss = 0.32716576
Iteration 175, loss = 0.32557722
Iteration 176, loss = 0.32544991
Iteration 177, loss = 0.32541187
Iteration 178, loss = 0.32569913
```

```
Iteration 179, loss = 0.32497722
Iteration 180, loss = 0.32467608
Iteration 181, loss = 0.32399568
Iteration 182, loss = 0.32345920
Iteration 183, loss = 0.32472374
Iteration 184, loss = 0.32314476
Iteration 185, loss = 0.32263961
Iteration 186, loss = 0.32334152
Iteration 187, loss = 0.32228950
Iteration 188, loss = 0.32215252
Iteration 189, loss = 0.32186922
Iteration 190, loss = 0.32292629
Iteration 191, loss = 0.32179015
Iteration 192, loss = 0.32208707
Iteration 193, loss = 0.32069643
Iteration 194, loss = 0.32073017
Iteration 195, loss = 0.32008789
Iteration 196, loss = 0.31984777
Iteration 197, loss = 0.32175395
Iteration 198, loss = 0.31897484
Iteration 199, loss = 0.31884130
Iteration 200, loss = 0.31824743
Iteration 201, loss = 0.31851436
Iteration 202, loss = 0.31879209
Iteration 203, loss = 0.31772172
Iteration 204, loss = 0.31788910
Iteration 205, loss = 0.31742688
Iteration 206, loss = 0.31677229
Iteration 207, loss = 0.31800071
Iteration 208, loss = 0.31619218
Iteration 209, loss = 0.31592884
Iteration 210, loss = 0.31560421
Iteration 211, loss = 0.31553998
Iteration 212, loss = 0.31543414
Iteration 213, loss = 0.31515021
Iteration 214, loss = 0.31481881
Iteration 215, loss = 0.31478423
Iteration 216, loss = 0.31480533
Iteration 217, loss = 0.31386195
Iteration 218, loss = 0.31359239
Iteration 219, loss = 0.31320579
Iteration 220, loss = 0.31395694
Iteration 221, loss = 0.31350982
Iteration 222, loss = 0.31269138
Iteration 223, loss = 0.31236252
Iteration 224, loss = 0.31146817
Iteration 225, loss = 0.31175537
Iteration 226, loss = 0.31183964
```

```
Iteration 227, loss = 0.31117781
Iteration 228, loss = 0.31096004
Iteration 229, loss = 0.31087803
Iteration 230, loss = 0.31081723
Iteration 231, loss = 0.31006414
Iteration 232, loss = 0.31036612
Iteration 233, loss = 0.31270626
Iteration 234, loss = 0.31225699
Iteration 235, loss = 0.31180479
Iteration 236, loss = 0.30973036
Iteration 237, loss = 0.30897475
Iteration 238, loss = 0.30913600
Iteration 239, loss = 0.30965078
Iteration 240, loss = 0.30949752
Iteration 241, loss = 0.30861700
Iteration 242, loss = 0.30852095
Iteration 243, loss = 0.30785590
Iteration 244, loss = 0.30727303
Iteration 245, loss = 0.30735157
Iteration 246, loss = 0.30663902
Iteration 247, loss = 0.30678297
Iteration 248, loss = 0.30617974
Iteration 249, loss = 0.30662153
Iteration 250, loss = 0.30677646
Iteration 251, loss = 0.30578781
Iteration 252, loss = 0.30553012
Iteration 253, loss = 0.30572820
Iteration 254, loss = 0.30650572
Iteration 255, loss = 0.30660103
Iteration 256, loss = 0.30472657
Iteration 257, loss = 0.30488801
Iteration 258, loss = 0.30455576
Iteration 259, loss = 0.30396633
Iteration 260, loss = 0.30414076
Iteration 261, loss = 0.30382351
Iteration 262, loss = 0.30387338
Iteration 263, loss = 0.30332348
Iteration 264, loss = 0.30332678
Iteration 265, loss = 0.30320142
Iteration 266, loss = 0.30354344
Iteration 267, loss = 0.30310687
Iteration 268, loss = 0.30262623
Iteration 269, loss = 0.30206681
Iteration 270, loss = 0.30201686
Iteration 271, loss = 0.30226319
Iteration 272, loss = 0.30182968
Iteration 273, loss = 0.30153914
Iteration 274, loss = 0.30120346
```

```
Iteration 275, loss = 0.30172840
Iteration 276, loss = 0.30158535
Iteration 277, loss = 0.30044616
Iteration 278, loss = 0.30054607
Iteration 279, loss = 0.30032967
Iteration 280, loss = 0.30086427
Iteration 281, loss = 0.29996635
Iteration 282, loss = 0.29984323
Iteration 283, loss = 0.29991163
Iteration 284, loss = 0.29975467
Iteration 285, loss = 0.30000480
Iteration 286, loss = 0.29886354
Iteration 287, loss = 0.29869975
Iteration 288, loss = 0.29955594
Iteration 289, loss = 0.29832923
Iteration 290, loss = 0.29863865
Iteration 291, loss = 0.29825773
Iteration 292, loss = 0.29825476
Iteration 293, loss = 0.29827645
Iteration 294, loss = 0.29760956
Iteration 295, loss = 0.29834532
Iteration 296, loss = 0.29800641
Iteration 297, loss = 0.29851850
Iteration 298, loss = 0.29717716
Iteration 299, loss = 0.29723516
Iteration 300, loss = 0.29752913
```

```
/usr/local/lib/python3.10/dist-
packages/sklearn/neural_network/_multilayer_perceptron.py:686:
ConvergenceWarning: Stochastic Optimizer: Maximum iterations (300) reached and
the optimization hasn't converged yet.
  warnings.warn(
```

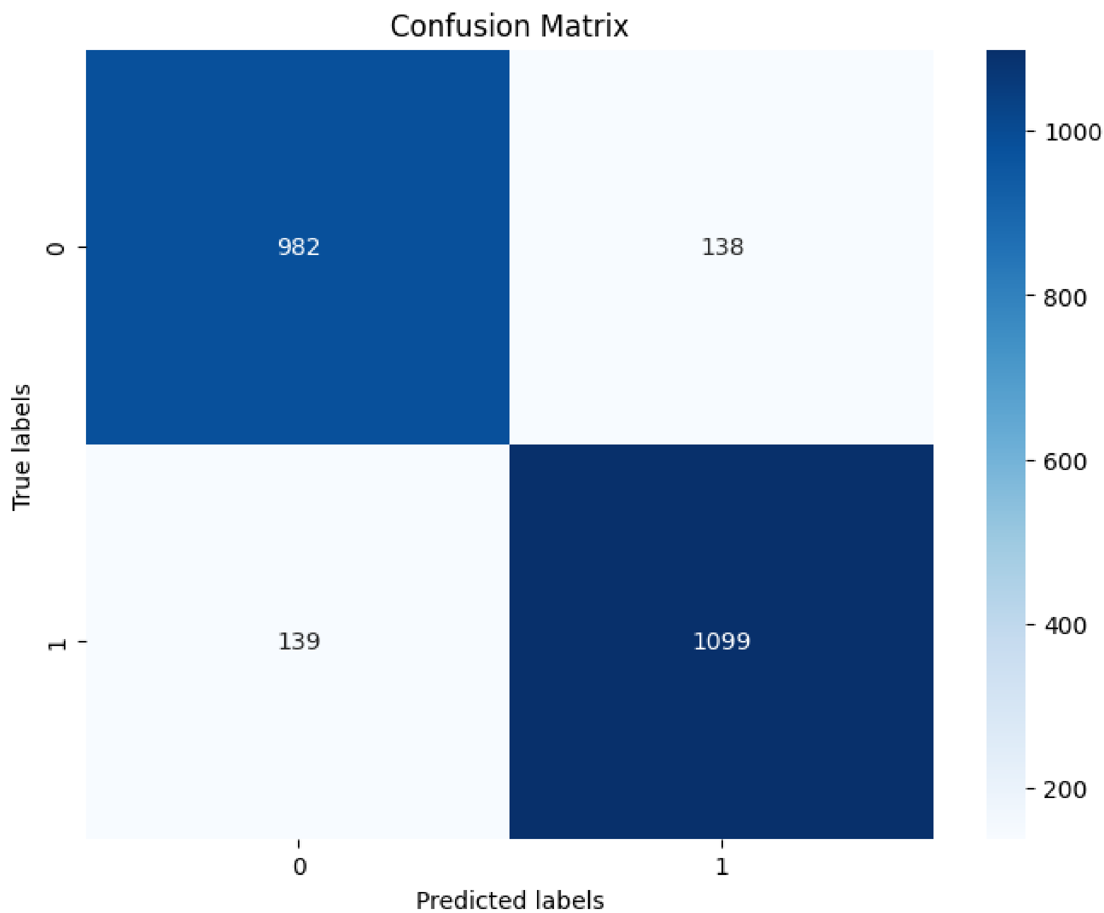
[ ]:
```
MLPClassifier(hidden_layer_sizes=(12, 10), max_iter=300, verbose=True)
```

#####**Evaluation and Validation of the Model(SKlearn)**

[ ]:
```python
from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Predict on the testing set
y_pred = clf.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
```

```python
# Generate classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='g', cmap='Blues')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('Confusion Matrix')
plt.show()
```

```
Accuracy: 0.8825275657336726

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.88      0.88      1120
           1       0.89      0.89      0.89      1238

    accuracy                           0.88      2358
   macro avg       0.88      0.88      0.88      2358
weighted avg       0.88      0.88      0.88      2358
```



Remarks: After processing the model. The model can learn overtime as more iterations in the model, decreases the loss to the point that it can no longer decrease and the model stops. however upon running the model the accuracy outputed a score of around 88%, while the output of the confusion matrix show that the model did well on predicting correctly and learning from its mistakes over time.

#MultiLayer Perceptron Using Tensorflow

#####Loading, processing, and cleaning the data for the 2nd model

[536]:
```
import tensorflow as tf
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

```
from tensorflow.keras.optimizers import Adam
import matplotlib.pyplot as plt
```

[518]:
```
data = pd.read_csv('/content/banana_quality.csv')
data.head(5)
```

[518]:
```
       Size    Weight  Sweetness  Softness  HarvestTime  Ripeness   Acidity  \
0 -1.924968  0.468078   3.077832 -1.472177     0.294799  2.435570  0.271290
1 -2.409751  0.486870   0.346921 -2.495099    -0.892213  2.067549  0.307325
2 -0.357607  1.483176   1.568452 -2.645145    -0.647267  3.090643  1.427322
3 -0.868524  1.566201   1.889605 -1.273761    -1.006278  1.873001  0.477862
4  0.651825  1.319199  -0.022459 -1.209709    -1.430692  1.078345  2.812442

  Quality
0    Good
1    Good
2    Good
3    Good
4    Good
```

[519]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Size         8000 non-null   float64
 1   Weight       8000 non-null   float64
 2   Sweetness    8000 non-null   float64
 3   Softness     8000 non-null   float64
 4   HarvestTime  8000 non-null   float64
 5   Ripeness     8000 non-null   float64
 6   Acidity      8000 non-null   float64
 7   Quality      8000 non-null   object
dtypes: float64(7), object(1)
memory usage: 500.1+ KB
```

[520]:
```
from sklearn import preprocessing

# Creating labelEncoder
le = preprocessing.LabelEncoder()

# Convert string label into numbers.
data['Quality']=le.fit_transform(data['Quality'])
```

[521]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8000 entries, 0 to 7999
Data columns (total 8 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Size         8000 non-null   float64
 1   Weight       8000 non-null   float64
 2   Sweetness    8000 non-null   float64
 3   Softness     8000 non-null   float64
 4   HarvestTime  8000 non-null   float64
 5   Ripeness     8000 non-null   float64
 6   Acidity      8000 non-null   float64
 7   Quality      8000 non-null   int64  
dtypes: float64(7), int64(1)
memory usage: 500.1 KB
```

#####**Implementing Feature Importance**

[522]:
```
# Separate features and target variable
X = data.drop(['Acidity', 'Quality', 'Sweetness', 'Size'], axis=1)
y = data['Quality']

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

# Feature importance
rf = RandomForestClassifier(n_estimators=100, random_state=42)
rf.fit(X_train, y_train)
feature_importances = rf.feature_importances_
feature_importance_df = pd.DataFrame({'Feature': X.columns, 'Importance':␣
 ↪feature_importances})
feature_importance_df = feature_importance_df.sort_values(by='Importance',␣
 ↪ascending=False)
print("Feature importance:")
print(feature_importance_df)
```

```
Feature importance:
       Feature  Importance
2  HarvestTime    0.290070
0       Weight    0.270018
3     Ripeness    0.221586
1     Softness    0.218326
```

#####**Removing Outliers**

[523]:
```
# Remove outliers
Q1 = X.quantile(0.25)
Q3 = X.quantile(0.75)
```

```
IQR = Q3 - Q1
threshold = 1.5 * IQR
outliers = (X < (Q1 - threshold)) | (X > (Q3 + threshold))
cleaned_df = X[~outliers.any(axis=1)]

Q1_2 = cleaned_df.quantile(0.25)
Q3_2 = cleaned_df.quantile(0.75)
IQR_2 = Q3_2 - Q1_2
threshold_2 = 1.5 * IQR_2
outliers_2 = (cleaned_df < (Q1_2 - threshold_2)) | (cleaned_df > (Q3_2 +␣
 ↪threshold_2))
cleaned_df2 = cleaned_df[~outliers_2.any(axis=1)]
cleaned_df2.describe()
```

[523]:

	Weight	Softness	HarvestTime	Ripeness
count	7859.000000	7859.000000	7859.000000	7859.000000
mean	-0.742800	-0.028938	-0.780937	0.823255
std	2.009928	2.058572	1.937883	2.050002
min	-6.609340	-6.290912	-6.002344	-4.749013
25%	-2.215542	-1.618966	-2.127844	-0.534235
50%	-0.848095	0.179909	-0.957602	0.986723
75%	0.796848	1.543462	0.463344	2.276937
max	5.184198	6.124278	4.333418	6.490461

#####Standardize features

[524]:
```
# Standardize the data
scaler = StandardScaler()
scaled_df = scaler.fit_transform(cleaned_df2)
pd.DataFrame(scaled_df, columns=cleaned_df2.columns).describe()
```

[524]:

	Weight	Softness	HarvestTime	Ripeness
count	7.859000e+03	7859.000000	7.859000e+03	7.859000e+03
mean	-5.786326e-17	0.000000	8.679489e-17	-5.786326e-17
std	1.000064e+00	1.000064	1.000064e+00	1.000064e+00
min	-2.918967e+00	-3.042095	-2.694559e+00	-2.718350e+00
25%	-7.327807e-01	-0.772443	-6.950850e-01	-6.622321e-01
50%	-5.239083e-02	0.101459	-9.116975e-02	7.974532e-02
75%	7.660702e-01	0.763879	6.421232e-01	7.091577e-01
max	2.949049e+00	2.989260	2.639314e+00	2.764664e+00

#####Normalizing the Data

[525]:
```
# Normalize the data
min_max_scaler = MinMaxScaler()
normalized_X = min_max_scaler.fit_transform(scaled_df)
pd.DataFrame(normalized_X, columns=cleaned_df2.columns).describe()
```

[525]:

	Weight	Softness	HarvestTime	Ripeness
count	7859.000000	7859.000000	7859.000000	7859.000000
mean	0.497437	0.504380	0.505179	0.495777
std	0.170426	0.165811	0.187493	0.182393
min	0.000000	0.000000	0.000000	0.000000
25%	0.372560	0.376309	0.374863	0.374998
50%	0.488509	0.521202	0.488086	0.510321
75%	0.627987	0.631031	0.625565	0.625114
max	1.000000	1.000000	1.000000	1.000000

Remarks: Simalar process as the SKlearn model, the data is first cleaned and processed for improved results, better understanding of the data, and making the data into smaller scale values.

[547]:
```
num_samples_to_keep = min(normalized_X.shape[0], y_train.shape[0])

normalized_X = normalized_X[:num_samples_to_keep]
y_train = y_train[:num_samples_to_keep]

print("Shapes after alignment:")
print("Shape of input data:", normalized_X.shape)
print("Shape of labels:", y_train.shape)
```

```
Shapes after alignment:
Shape of input data: (5600, 4)
Shape of labels: (5600,)
```

Remarks: The processed data in the previous repeated codes are for variable X or features only while y or the target variable is just alligning the processed value data of X variable.

#####**Creating and Training the Tensorflow Keras Model**

[548]:
```
# Creating the Model
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.5),  # Add dropout regularization
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])
```

[549]:
```
# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

[550]:
```
# Train the model
history = model.fit(X_train, y_train, epochs=50, batch_size=32,␣
 ↪validation_split=0.2)
```

```
Epoch 1/50
```

```
140/140 [==============================] - 1s 3ms/step - loss: 0.4305 -
accuracy: 0.8065 - val_loss: 0.2995 - val_accuracy: 0.8732
Epoch 2/50
140/140 [==============================] - 0s 2ms/step - loss: 0.3490 -
accuracy: 0.8554 - val_loss: 0.2747 - val_accuracy: 0.8848
Epoch 3/50
140/140 [==============================] - 0s 2ms/step - loss: 0.3208 -
accuracy: 0.8679 - val_loss: 0.2575 - val_accuracy: 0.8929
Epoch 4/50
140/140 [==============================] - 0s 3ms/step - loss: 0.3014 -
accuracy: 0.8779 - val_loss: 0.2539 - val_accuracy: 0.8938
Epoch 5/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2999 -
accuracy: 0.8833 - val_loss: 0.2492 - val_accuracy: 0.8938
Epoch 6/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2917 -
accuracy: 0.8850 - val_loss: 0.2389 - val_accuracy: 0.9071
Epoch 7/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2848 -
accuracy: 0.8879 - val_loss: 0.2327 - val_accuracy: 0.9125
Epoch 8/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2829 -
accuracy: 0.8935 - val_loss: 0.2255 - val_accuracy: 0.9107
Epoch 9/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2691 -
accuracy: 0.8982 - val_loss: 0.2246 - val_accuracy: 0.9179
Epoch 10/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2688 -
accuracy: 0.8915 - val_loss: 0.2181 - val_accuracy: 0.9214
Epoch 11/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2592 -
accuracy: 0.8989 - val_loss: 0.2167 - val_accuracy: 0.9170
Epoch 12/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2608 -
accuracy: 0.9067 - val_loss: 0.2158 - val_accuracy: 0.9205
Epoch 13/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2549 -
accuracy: 0.9047 - val_loss: 0.2116 - val_accuracy: 0.9268
Epoch 14/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2613 -
accuracy: 0.9002 - val_loss: 0.2118 - val_accuracy: 0.9259
Epoch 15/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2490 -
accuracy: 0.9045 - val_loss: 0.2130 - val_accuracy: 0.9232
Epoch 16/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2509 -
accuracy: 0.9054 - val_loss: 0.2172 - val_accuracy: 0.9214
Epoch 17/50
```

```
140/140 [==============================] - 0s 3ms/step - loss: 0.2474 -
accuracy: 0.9047 - val_loss: 0.2098 - val_accuracy: 0.9250
Epoch 18/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2499 -
accuracy: 0.9105 - val_loss: 0.2079 - val_accuracy: 0.9277
Epoch 19/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2491 -
accuracy: 0.9058 - val_loss: 0.2102 - val_accuracy: 0.9295
Epoch 20/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2419 -
accuracy: 0.9056 - val_loss: 0.2095 - val_accuracy: 0.9312
Epoch 21/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2462 -
accuracy: 0.9100 - val_loss: 0.2074 - val_accuracy: 0.9304
Epoch 22/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2527 -
accuracy: 0.9076 - val_loss: 0.2140 - val_accuracy: 0.9259
Epoch 23/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2442 -
accuracy: 0.9042 - val_loss: 0.2110 - val_accuracy: 0.9277
Epoch 24/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2433 -
accuracy: 0.9067 - val_loss: 0.2109 - val_accuracy: 0.9268
Epoch 25/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2376 -
accuracy: 0.9107 - val_loss: 0.2094 - val_accuracy: 0.9286
Epoch 26/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2405 -
accuracy: 0.9085 - val_loss: 0.2103 - val_accuracy: 0.9286
Epoch 27/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2374 -
accuracy: 0.9109 - val_loss: 0.2131 - val_accuracy: 0.9295
Epoch 28/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2375 -
accuracy: 0.9109 - val_loss: 0.2103 - val_accuracy: 0.9250
Epoch 29/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2400 -
accuracy: 0.9087 - val_loss: 0.2102 - val_accuracy: 0.9312
Epoch 30/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2325 -
accuracy: 0.9154 - val_loss: 0.2040 - val_accuracy: 0.9277
Epoch 31/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2362 -
accuracy: 0.9134 - val_loss: 0.2061 - val_accuracy: 0.9277
Epoch 32/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2384 -
accuracy: 0.9078 - val_loss: 0.2066 - val_accuracy: 0.9277
Epoch 33/50
```

```
140/140 [==============================] - 0s 2ms/step - loss: 0.2346 -
accuracy: 0.9100 - val_loss: 0.2185 - val_accuracy: 0.9232
Epoch 34/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2407 -
accuracy: 0.9069 - val_loss: 0.2090 - val_accuracy: 0.9259
Epoch 35/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2341 -
accuracy: 0.9158 - val_loss: 0.2066 - val_accuracy: 0.9277
Epoch 36/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2351 -
accuracy: 0.9107 - val_loss: 0.2118 - val_accuracy: 0.9268
Epoch 37/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2296 -
accuracy: 0.9158 - val_loss: 0.2077 - val_accuracy: 0.9277
Epoch 38/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2323 -
accuracy: 0.9103 - val_loss: 0.2125 - val_accuracy: 0.9259
Epoch 39/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2288 -
accuracy: 0.9114 - val_loss: 0.2048 - val_accuracy: 0.9295
Epoch 40/50
140/140 [==============================] - 1s 4ms/step - loss: 0.2349 -
accuracy: 0.9080 - val_loss: 0.2078 - val_accuracy: 0.9259
Epoch 41/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2286 -
accuracy: 0.9143 - val_loss: 0.2044 - val_accuracy: 0.9277
Epoch 42/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2275 -
accuracy: 0.9165 - val_loss: 0.2044 - val_accuracy: 0.9286
Epoch 43/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2270 -
accuracy: 0.9123 - val_loss: 0.2110 - val_accuracy: 0.9268
Epoch 44/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2334 -
accuracy: 0.9127 - val_loss: 0.2091 - val_accuracy: 0.9259
Epoch 45/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2313 -
accuracy: 0.9147 - val_loss: 0.2025 - val_accuracy: 0.9268
Epoch 46/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2318 -
accuracy: 0.9150 - val_loss: 0.2106 - val_accuracy: 0.9286
Epoch 47/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2250 -
accuracy: 0.9161 - val_loss: 0.2047 - val_accuracy: 0.9268
Epoch 48/50
140/140 [==============================] - 0s 3ms/step - loss: 0.2291 -
accuracy: 0.9132 - val_loss: 0.2080 - val_accuracy: 0.9286
Epoch 49/50
```

```
140/140 [==============================] - 0s 2ms/step - loss: 0.2309 -
accuracy: 0.9145 - val_loss: 0.2075 - val_accuracy: 0.9250
Epoch 50/50
140/140 [==============================] - 0s 2ms/step - loss: 0.2313 -
accuracy: 0.9136 - val_loss: 0.2074 - val_accuracy: 0.9259
```

Remarks: After creating and training the model I noticed that my output is still learning for in more iterations however the accuracy overtime will almost stop after a few more epochs as the accuracy value is going smaller increased overtime, however the model did still also learn overtime from the trained and test data.

#####**Evaluation and Validation of the Model(TensorFlow)**

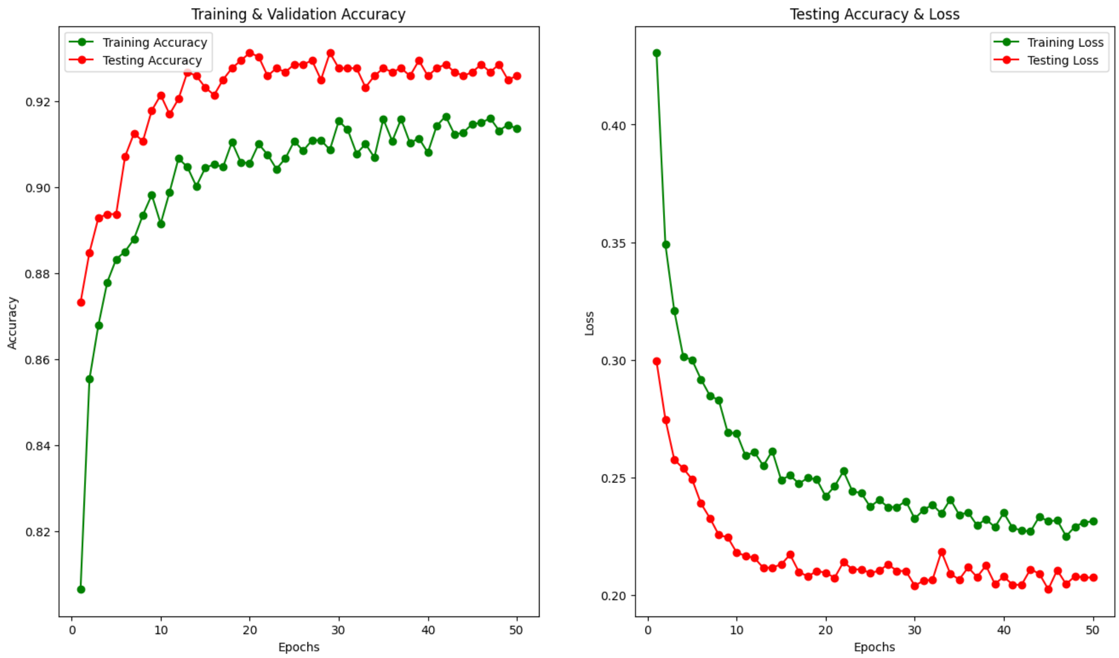
[551]:
```
# Evaluate the model on the test set
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss}')
print(f'Test Accuracy: {test_accuracy}')
```

```
75/75 [==============================] - 0s 2ms/step - loss: 0.2131 - accuracy:
0.9192
Test Loss: 0.21309509873390198
Test Accuracy: 0.9191666841506958
```

Remarks: After performing the cleaning, processing and creating the model the results are good as test accuracy shows around 91% while loss at around 21% although there still inconsistences if the model is rerun it either decreases of increases.

[552]:
```
# Plot training and validation metrics
epochs = range(1, len(history.history['accuracy']) + 1)
train_acc = history.history['accuracy']
train_loss = history.history['loss']
val_acc = history.history['val_accuracy']
val_loss = history.history['val_loss']

fig, ax = plt.subplots(1, 2)
fig.set_size_inches(16, 9)

ax[0].plot(epochs, train_acc, 'go-', label='Training Accuracy')
ax[0].plot(epochs, val_acc, 'ro-', label='Testing Accuracy')
ax[0].set_title('Training & Validation Accuracy')
ax[0].legend()
ax[0].set_xlabel("Epochs")
ax[0].set_ylabel("Accuracy")

ax[1].plot(epochs, train_loss, 'g-o', label='Training Loss')
ax[1].plot(epochs, val_loss, 'r-o', label='Testing Loss')
ax[1].set_title('Testing Accuracy & Loss')
ax[1].legend()
ax[1].set_xlabel("Epochs")
```

```
ax[1].set_ylabel("Loss")

plt.show()
```



Remarks: The plotter shows a good result and plots the loss and accuracy of the trained and test data over time, it shows that the plots are almost parallel with each other but there are stil gaps in between therefore improvement in decreasing the loss could still be an option, as different parameters are being change in the model to either improve or worsen the model.

#Summary, Conclusion, and Lesson Learned

In sumamry building and applying the multilayer perceptron, will help us understand more of machine learning, and be applied trough aspects of engineering and analysis, benchmarking both of the models (SKlearn and TensorFlow) I observed that good are pretty good algorithms in terms of integration SKlearn is I find easy to understand, compare to TensorFlow, however in terms of handling data TensorFlow shows greater accuracy over SKlearn, and I learned that by building and applying MLP will help me gain knowledge and skill to apply in technology that has data acquisition system and data engineering.