December 14, 2021

1 Letter Recognition Using Machine Learning Models (Irvyn Hall)

The main objective of this paper is to predict letters based on statistic values of unique stimuli through machine learning models.

I will do this through exploring my data, stating the findings from the exploration, data preprocessing, training the model using an 80/20 training split and by through one (random forest) or many different modelling algorithms, improving on my current models and finally by presenting my model and stating its reproducibility. The reason i'm using a random forest model is that it exibits averaging, the decision from the random forest model is an average of all the predictions made.

2 1 Research & Dataset Exploration

There are many systems in our life that need to predict letters and words. For example, the SQA are known to use a system that automatically scans student papers and produces a digital version of them, another example is an online software called "gyazo" they use a proprietary OCR system to allow users to search through their saved images. There is obviously a growing demand for systems to be able to identify text. Its abundantly clear that there is a market for text-letter prediction.

Now in real world systems there is more than just being able to "predict" the letter, you firstly need to first be able to correctly label the data, this is a whole problem of its own. This is not what my paper is going to explore as I'm focusing on processing the data that has already been labelled and is being used to create the prediction models.

3 2 Dataset

I obtained this dataset from the UCI repository link from in my references It was first downloaded on the 23rd of September The reason I chose this dataset was due to my interest in image classification/recognition. I have always been interested in how certain systems and applications are able to determine text from pictures or videos.

• 1. x-box horizontal position of box (f_1) • 2. y-box vertical position of box (f_2) • 3. width width of box (f_3) • 4. high height of box (f_4) • 5. onpix total # on pixels (f_5) • 6. x-bar mean x of on pixels in box (f_6) • 7. y-bar mean y of on pixels in box (f_7) • 8. x2bar mean x variance (f_8) • 9. y2bar mean y variance (f_9) • 10. xybar mean x y correlation (f_10) • 11. x2ybrmeanofx*x*y(f_11) • 12. xy2brmeanofx*y*y(f_12) • 13. x-ege mean edge count left to right (f_13) • 14. xegvy correlation

of x-ege with y (f_14) • 15. y-ege mean edge count bottom to top (f_15) • 16. yegvx correlation of y-ege with x (f_16) • 17. Lettr Letter the values represent (Label)

As we know from our own writing some letters in the English alphabet are similar and it must be investigated how this can affect data extraction and model accuracy. For instance, the letter "M" and "N" are very similar visually, this means even if you get an accurate way to extract letters there is still another problem to tackle - correctly labelling the data. The dataset I have used is a combination of statistical attributes so it is able to give my model features that can allow for easier classification between these visually similar letters.

As a result of the dataset being pre-processed our findings can misrepresent how accurate the application of the model into the real-world will be, this is due to the following unknown variables: 1. The distortion method used. 2. The types of 20 different fonts used (are they all one font family just 20 different variations?) 3. The data was rounded after the values were extracted - extracting the mean and correlation would create floating point values but as you can see the dataset has been rounded into real numbers.

With the above points being made, I still believe that the model can provide an accurate way to predict letters based off the statistical attributes provided and as such it can still be an accurate model.

4 3 Related Work & Dataset Exploration

[93]:
```
# importing the libarries needed
import numpy as np
import pandas as pd
import seaborn as sns # for plotting
import matplotlib.pyplot as plt # for further data exploration
from collections import Counter
```

[94]:
```
# importing the data file and printing off a couple records to see the data␣␣
 ↪topology
df = pd.read_csv('./Data/letterRecog.csv')
```

First lets see how many rows and columns are in the dataset.

[95]:
```
# print the amount of rows and columns in the data frame
print(f'There are {df.shape[0]} rows, and {df.shape[1]} columns')
```

```
There are 20000 rows, and 17 columns
```

[96]:
```
# this will give us a more detailed analysis of the rows and columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20000 entries, 0 to 19999
Data columns (total 17 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Label   20000 non-null  object
```

```
 1   f_1     20000 non-null  int64
 2   f_2     20000 non-null  int64
 3   f_3     20000 non-null  int64
 4   f_4     20000 non-null  int64
 5   f_5     20000 non-null  int64
 6   f_6     20000 non-null  int64
 7   f_7     20000 non-null  int64
 8   f_8     20000 non-null  int64
 9   f_9     20000 non-null  int64
 10  f_10    20000 non-null  int64
 11  f_11    20000 non-null  int64
 12  f_12    20000 non-null  int64
 13  f_13    20000 non-null  int64
 14  f_14    20000 non-null  int64
 15  f_15    20000 non-null  int64
 16  f_16    20000 non-null  int64
dtypes: int64(16), object(1)
memory usage: 2.6+ MB
```

[97]:
```
# This will check for each item in the dataframe is there is a empty value it␣␣
 ↪will count it for each row
df.isnull().sum()
```

[97]:
```
Label    0
f_1      0
f_2      0
f_3      0
f_4      0
f_5      0
f_6      0
f_7      0
f_8      0
f_9      0
f_10     0
f_11     0
f_12     0
f_13     0
f_14     0
f_15     0
f_16     0
dtype: int64
```

This confirms that the dataset has no missing values, this makes my job easier as I don't have to employ missing-value methods. I will investigate the class distribution now to see if I must use any class distribution balancing methods.

Now I will print the count of each label and the percentage of the dataset that they represent, this is to check the class distribution - we have 20000 items if it was a perfectly balanced dataset 20 labels would be present 769 times with 6 of them being present 770 times

[98]:
```
# Counting how many of each label there are
count = (Counter(df.Label))
print(Counter(df.Label))
print("\n")
# Turn this into a percentage
alpha = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N',
         'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
# for each label calculate the percentage of the total dataset it represents #␣
 ↪tried to keep the time complexity as low as possible
for x in range (26):
    n_a= df[df.Label==alpha[x]].shape[0]
    p = (n_a/20000*100)
    print (alpha[x], +round(p,2),"%")
```

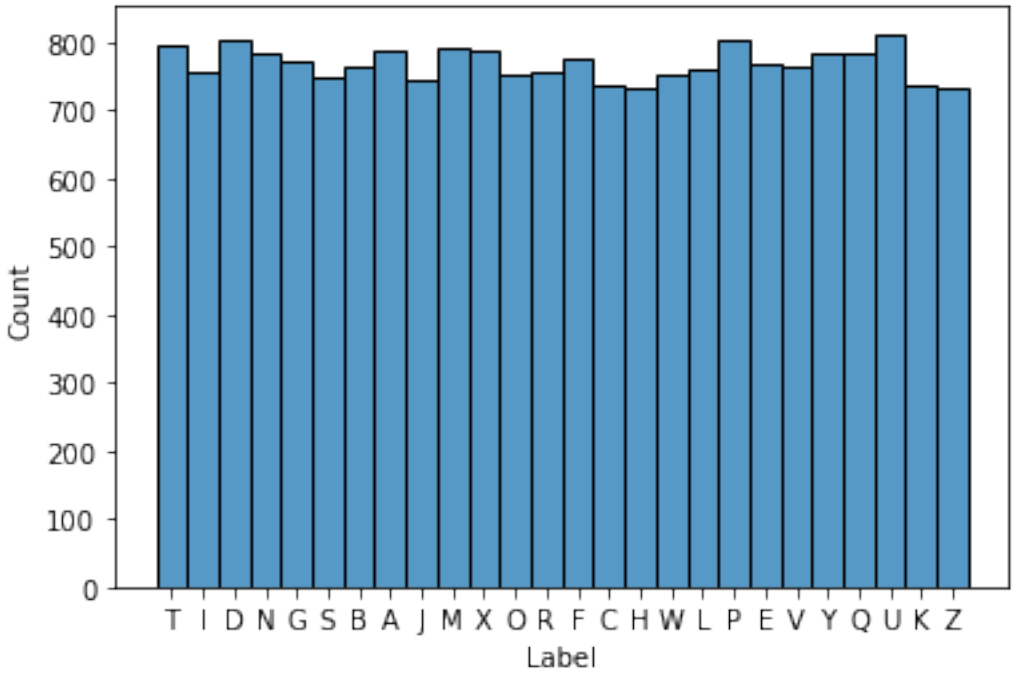
```
Counter({'U': 813, 'D': 805, 'P': 803, 'T': 796, 'M': 792, 'A': 789, 'X': 787,
'Y': 786, 'N': 783, 'Q': 783, 'F': 775, 'G': 773, 'E': 768, 'B': 766, 'V': 764,
'L': 761, 'R': 758, 'I': 755, 'O': 753, 'W': 752, 'S': 748, 'J': 747, 'K': 739,
'C': 736, 'H': 734, 'Z': 734})


A 3.94 %
B 3.83 %
C 3.68 %
D 4.03 %
E 3.84 %
F 3.88 %
G 3.86 %
H 3.67 %
I 3.77 %
J 3.74 %
K 3.69 %
L 3.81 %
M 3.96 %
N 3.91 %
O 3.77 %
P 4.01 %
Q 3.91 %
R 3.79 %
S 3.74 %
T 3.98 %
U 4.06 %
V 3.82 %
W 3.76 %
X 3.94 %
Y 3.93 %
Z 3.67 %
```

There is only a ± .5% difference in the most common and least common label, this would be

classified as a balanced dataset - given that the least common label is present 734 times.

[99]:
```
sns.histplot(df, x="Label", binwidth=3)
```

[99]:
```
<AxesSubplot:xlabel='Label', ylabel='Count'>
```



Lets see what the values of the features look like and how distributed they are, we can do this as they are statistical attributes.

[100]:
```
# gather some statistical data on the numerical values - note that the label␣
↪isn't included
df.describe()
```

[100]:

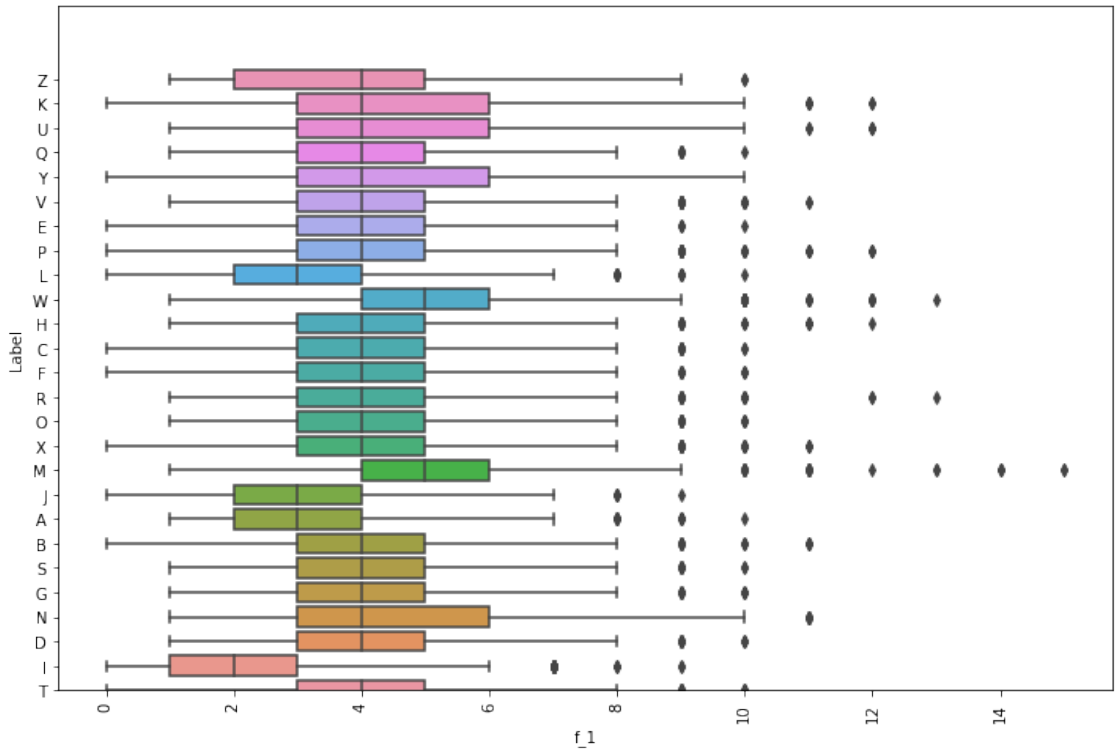
	f_1	f_2	f_3	f_4	f_5	\
count	20000.000000	20000.000000	20000.000000	20000.00000	20000.000000	
mean	4.023550	7.035500	5.121850	5.37245	3.505850	
std	1.913212	3.304555	2.014573	2.26139	2.190458	
min	0.000000	0.000000	0.000000	0.00000	0.000000	
25%	3.000000	5.000000	4.000000	4.00000	2.000000	
50%	4.000000	7.000000	5.000000	6.00000	3.000000	
75%	5.000000	9.000000	6.000000	7.00000	5.000000	
max	15.000000	15.000000	15.000000	15.00000	15.000000	

	f_6	f_7	f_8	f_9	f_10	\
count	20000.000000	20000.000000	20000.000000	20000.000000	20000.000000	

```
mean       6.897600      7.500450      4.628600      5.178650      8.282050
std        2.026035      2.325354      2.699968      2.380823      2.488475
min        0.000000      0.000000      0.000000      0.000000      0.000000
25%        6.000000      6.000000      3.000000      4.000000      7.000000
50%        7.000000      7.000000      4.000000      5.000000      8.000000
75%        8.000000      9.000000      6.000000      7.000000     10.000000
max       15.000000     15.000000     15.000000     15.000000     15.000000

              f_11          f_12          f_13          f_14          f_15  \
count  20000.00000  20000.000000  20000.000000  20000.000000  20000.000000
mean       6.45400      7.929000      3.046100      8.338850      3.691750
std        2.63107      2.080619      2.332541      1.546722      2.567073
min        0.00000      0.000000      0.000000      0.000000      0.000000
25%        5.00000      7.000000      1.000000      8.000000      2.000000
50%        6.00000      8.000000      3.000000      8.000000      3.000000
75%        8.00000      9.000000      4.000000      9.000000      5.000000
max       15.00000     15.000000     15.000000     15.000000     15.000000

              f_16
count  20000.00000
mean       7.80120
std        1.61747
min        0.00000
25%        7.00000
50%        8.00000
75%        9.00000
max       15.00000
```

We can see that some of the features are skewed differently, they have different means and deviation. We also can't really draw many correlations mentally from these numbers but there is a .corr() feature we can use to see which features are similar.

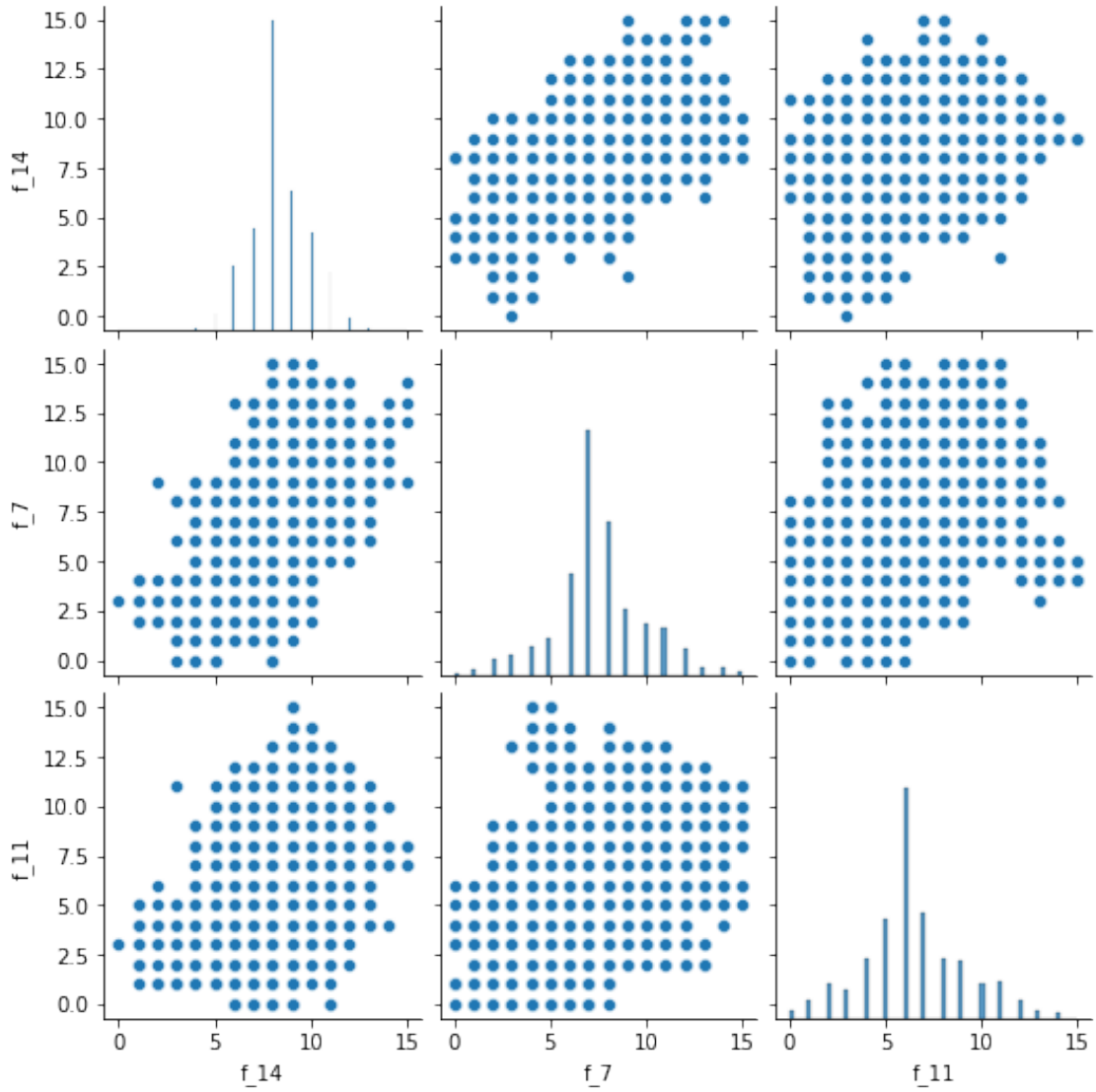
[101]:
```
# Box plot to see the distrubution of labels better for the given feature
var = 'f_1'
data = pd.concat([df['Label'], df[var]], axis=1)
f, ax = plt.subplots(figsize=(12, 8))
fig = sns.boxplot(x=var, y="Label", data=data)
fig.axis(ymin=0, ymax=28);
plt.xticks(rotation=90);
```



[102]:
```
corr_mat= df.corr()
corr_mat
#see all the values that correlate to feature 14
corr_mat['f_14'].sort_values(ascending=True)
```

[102]:
```
f_6    -0.253339
f_16   -0.187591
f_12   -0.184927
f_8    -0.084820
f_15   -0.064402
f_9    -0.052545
f_2    -0.001336
f_13    0.002849
f_5     0.017649
f_4     0.025359
f_10    0.029419
f_3     0.045658
f_1     0.098180
f_11    0.527239
f_7     0.555060
f_14    1.000000
Name: f_14, dtype: float64
```

[103]:
```
y_vals = ['f_14', 'f_7', 'f_11']
sns.pairplot(df[y_vals[:3]])
plt.show()
```



This shows that feature 7 and 11 are positively correlated to feature 14 - this is evident as they are similar attributes in the fact that they are related to pixel's on the Y axis - (means/correlations)

This makes it much easier to see correlation between the features.

5 4 Data Pre-Processing

Using the data from the above table we can see that in every feature the maximum value is 15. Typically, machine learning algorithms that are based around distance measuring (KNN, k-means,

SVM) do not perform well with features that have larger range than others. This can also be applied to other non-distance orientated models.

To mitigate these issues, I'm going to be dividing my features by 15 (the maximum in each column) but ideally, I could make a function to pass the values through. This is referred to as data normalisation, this is a very good practice especially when using a dataset with a varied statistical scale.

I will set the X and Y train and test sets and encode our labels as they are categorical values (A-Z eg. 0-25) NOTE - I know that the use of encoded labels isn't needed for random forest trees nor decision trees but it is however a good practice so I will utilise it for the sake of potentially using different models.

[104]:
```
# store the features and labels
X = df.iloc[:,1:]
y = df.iloc[:,0]
# Turn the X df into a numpy array
X = np.array(X)
# encode labels into integers
from sklearn.preprocessing import LabelEncoder
y_labels = y
# set Le to the LabelEncoding function
le = LabelEncoder()
# fit the df object Y into the le function
le.fit(y)
y_encoded = le.transform(y)
```

[105]:
```
from sklearn.model_selection import train_test_split
# if you want to train on the encoded labels you would use
# X_train,X_test,y_train,y_test=train_test_split(X,y_encoded, test_size = 0.
 ↪3,random_state=30)
# we wont for now as its not needed with RF
X_train,X_test,y_train,y_test=train_test_split(X,y, test_size = 0.
 ↪3,random_state=30)
```

[106]:
```
# normalise the data
X_train = X_train /15
X_test = X_test /15
```

6 5 Peer-Reviewed Paper

I'm reviewing the paper "Kannada Alphabets. . . .Random Forest Models" as I'm going to be build- ing a similar model using a similar dataset. My reasoning is that in my opinion there were some vital loopholes in the dataset used that ultimately affected the outcome of the models created – making the models inefficient at real world usages.

I will review the paper in the following order: 1. Methodology 2. Dataset 3. Improvements 4. Conclusion.

There are lots of classification papers on similar datasets using more complex systems however, these will not be reviewed and will only be referred to where it is deemed necessary.

1. The paper goes into deep detail about the mathematical rigour behind the models it creates, two of the models mentioned are decision tree and random forest trees. It rightfully mentions how decision trees can end up overfitting the data. The paper mentions this can be avoided through Pre- and Post- pruning of the data and then explains how and why these pruning methods are applied. These two models will be used in my paper as they are both very effective and accurate models for classification problems, this is proven in the paper as a confusion matrix is created to show the accuracy of the models.

2. The dataset is pre-processed through many different techniques, the paper also mentions how their "clean dataset" can help increase the performance of the model. This statement may be true however, in the real world the data that will be used with these algorithms is going to vary like I stated in the start of my paper - people will write differently so if the dataset is "clean" then, it will only perform well on other "clean" datasets. Ideally to get accurate results a dataset that is more varied and distorted would be used.

3. The paper also states that their results could be greatly increased from the use of a HOG. This will essentially return gradients from the letters; this can allow the model to be trained on statistical attributes of an image rather than the RAW-image – this can create a model more accurate. Another way the models will be more accurate is if the dataset used has variance in the data, my model does this by being pre-processed statistical attributes of a randomly distorted collection of 20 different fonts. The different fonts accounts for the different in handwriting, the random distortion creates a difference in digitalising the data and then the statistical attributes allow the model to be trained on more features rather than just X/Y pixel presence (In a way it is essentially what the HOG does)

4. To conclude, the mentioning of the use of HOGs to improve the model's accuracy has left me wanting to implement it into my own model so I can produce a more accurate model, I believe this is slightly achieved through my use of statistical attributes rather than pixel's.

5.0.1 References: • Dataset Source: https://archive.ics.uci.edu/ml/datasets/Letter+Recognition • PeerReviewedPaper:https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9315972 • Visualisation graphs: https://www.kaggle.com/pmarcelino/comprehensive-data- exploration-with-python Krori Dutta , KKD. et al., 2020. "Kannada Alphabets Recognition using Decision Tree and Random Forest Models". 3rd International Conference on Intelligent Sustainable Systems (ICISS), Volume 1, Page 1-2.

7 Modelling/ Classification

7.0.1 Training

[107]:
```
from sklearn.ensemble import RandomForestClassifier
# create an instance of RandomForestClassifier with the parameters in parenthesis
rf=RandomForestClassifier(n_estimators=100,)
# fit the model on training data
rf.fit(X_train,y_train)
```

[107]:
```
RandomForestClassifier()
```

7.0.2 Predictions

[108]:
```
# Predicted classes of the testing set
rf_predictions = rf.predict(X_test)
    # get probabilities of predictions
rf_probs = rf.predict_proba(X_test)
    # returns prob for the first item
rf_probs[:1]
```

[108]:
```
array([[0.98, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
        0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.01, 0.  , 0.  ,
        0.  , 0.  , 0.  , 0.  ]])
```

[109]:
```
def display_accuracy(y_test, rf_predictions):
    import numpy as np
    from sklearn.metrics import accuracy_score
    from sklearn.metrics import classification_report, confusion_matrix
    # use the accuracy score from the sklearn.metrics library
    accuracy = accuracy_score(y_test, rf_predictions)
    print(f'The overall accuracy of RF is {np.round(accuracy*100,2)}%')
    return accuracy
display_accuracy(y_test, rf_predictions)
```

```
The overall accuracy of RF is 95.82%
```

[109]:
```
0.9581666666666667
```

There are obvious ways to improve the algorithms accuracy, one would be to increase the number of estimators but there is an exponential drop off point and i believe around 50-100 is reasonable anything higher than that results in about a marginal increase in accuracy with a exponential increase in time.

I can also look into the use of other fine-tuning parameters available for random forest algorithms such as MT RY.

[110]:
```
# inversed_labels = le.inverse_transform(y_test)
```

[111]:
```
from sklearn.metrics import classification_report, confusion_matrix
def display_classification(y_test, predictions):
    # the purpose of this report is to correctly identify and classify letters␣
 ↪on characteristics
    # so we will perform a classification report to see how correct and accurate␣
 ↪we can do this
    print(classification_report(y_test,predictions))
display_classification(y_test, rf_predictions)
```

	precision	recall	f1-score	support
A	1.00	1.00	1.00	229
B	0.85	0.96	0.90	228
C	0.99	0.95	0.97	220
D	0.92	0.96	0.94	219
E	0.96	0.95	0.95	232
F	0.96	0.92	0.94	225
G	0.94	0.94	0.94	234
H	0.93	0.92	0.92	206
I	0.97	0.96	0.96	236
J	0.98	0.95	0.96	209
K	0.96	0.96	0.96	213
L	1.00	0.95	0.97	239
M	0.95	0.99	0.97	240
N	0.99	0.95	0.97	239
O	0.93	0.93	0.93	243
P	0.96	0.96	0.96	243
Q	0.94	0.95	0.94	228
R	0.94	0.94	0.94	230
S	0.96	0.97	0.97	220
T	0.97	0.99	0.98	234
U	0.98	0.96	0.97	235
V	0.97	0.95	0.96	228
W	0.97	0.98	0.97	250
X	0.96	0.97	0.97	237
Y	0.99	0.96	0.97	252
Z	1.00	0.97	0.98	231
accuracy			0.96	6000
macro avg	0.96	0.96	0.96	6000
weighted avg	0.96	0.96	0.96	6000

[112]:
```
# fucntion to make the wrongly predicted letters more readable
def prediction_results(y_test, predictions):
    results = pd.DataFrame({'Actual':y_test,'Predicted':predictions})
    wrongResults = {'Actual' : [],  'Wrongly_Predicted': [] }
    for i, row in results.iterrows():
        if (row['Actual']) != (row['Predicted']):
            # i have the prediction mistakes here i just need to make it easier␣
 ↪to read
            wrongResults['Actual'].append(row['Actual'])
            wrongResults['Wrongly_Predicted'].append(row['Predicted'])
            # this will show which letters were wrongly predicted and the actual␣
 ↪letters that were wrongly predicted
```

```
    print("Wrongly predicted letters were :")
    print(Counter(wrongResults['Wrongly_Predicted']))
    print("The actual letters that were wrongly predicted were :")
    print(Counter(wrongResults['Actual']))
prediction_results(y_test, rf_predictions)
```

```
Wrongly predicted letters were :
Counter({'B': 39, 'D': 19, 'O': 18, 'Q': 15, 'R': 15, 'H': 15, 'G': 13, 'M': 12,
'P': 11, 'X': 10, 'E': 10, 'F': 9, 'W': 8, 'I': 8, 'S': 8, 'K': 8, 'V': 6, 'T':
6, 'U': 5, 'J': 5, 'C': 3, 'Y': 3, 'N': 3, 'Z': 1, 'A': 1})
The actual letters that were wrongly predicted were :
Counter({'F': 17, 'H': 17, 'O': 16, 'G': 14, 'R': 13, 'Q': 12, 'E': 12, 'L': 12,
'C': 12, 'V': 11, 'N': 11, 'Y': 10, 'U': 10, 'J': 10, 'B': 10, 'I': 9, 'K': 9,
'D': 9, 'P': 9, 'Z': 7, 'S': 6, 'X': 6, 'W': 5, 'T': 2, 'M': 2})
```

I would like to initially comment on the letter H having one of the lowest recall scores, this means that out of all the correct predictions the model made the letter H is the least common to correctly predict.

My prediction with the letters M + N being hard to predict was inaccurate, however the basis on which that prediction stands on is true, the letters (Q,G,O) are commonly in the top half of the wrongly predicted letters. I believe this is because the statistical attributes of these 2 letters are similar due to them all being "circle-like" in structure.

There are many different techniques that could be used to avoid / fix this problem - the use of HOQ's like mentioned in the peer-reviewed paper could have some affect on helping to distinguish them but, if a feature was to be produced in which it could help predict all the Q,G and O's it would lead to a increase of ±1.215% in the models accuracy here is how: Out of the 4000 test values ± 180 of them are wrongly predicted(180/4000)*100 =4.5% .

± 50 of these are either the letter Q, G's or O's

Meaning if it were to properly predict the 50 Q, G's or O's it would improve the accuracy by around 1.215% GIVEN that the 50 incorrectly identified Q, G's and O's are around 27% of the incorrectly predicted labels which amounts to 27% of the 4.5% which is: 27/100th of 4.5% = 1.215%

Given the typical application of this algorithm - in a system where you will 1. recognise a letter 2. deduce statistical attributes from an image of a recognised letter 3. use this algorithm 4. export data into a document an accuracy of ±95% is pretty accurate given that the aiding of word-auto completion could probably aid this solution, however for the purpose of this paper I will still try a different algorithm and try to improve on it further.

8 Solution Improvement

[113]:
```
#  -
from imblearn.over_sampling import SMOTE
oversample = SMOTE()
X, y = oversample.fit_resample(X, y)
counter = Counter(y)
```

```
print(counter)
```

```
Counter({'T': 813, 'I': 813, 'D': 813, 'N': 813, 'G': 813, 'S': 813, 'B': 813,
'A': 813, 'J': 813, 'M': 813, 'X': 813, 'O': 813, 'R': 813, 'F': 813, 'C': 813,
'H': 813, 'W': 813, 'L': 813, 'P': 813, 'E': 813, 'V': 813, 'Y': 813, 'Q': 813,
'U': 813, 'K': 813, 'Z': 813})
```

[114]:
```
X_train,X_test,y_train,y_test=train_test_split(X,y, test_size = 0.
↪3,random_state=30)
```

[115]:
```
X_train = X_train /15
X_test = X_test /15
```

[116]:
```
# larger amount of trees
rf=RandomForestClassifier(n_estimators=500, n_jobs = -1,max_depth = 200)
# n_jobs allows for no limit on processors to be used
rf.fit(X_train,y_train)

# Predicted classes of the testing set
rf_predictions = rf.predict(X_test)
    # get probabilities of predictions
rf_probs = rf.predict_proba(X_test)
    # returns prob for the first item
rf_probs[:1]
```

[116]:
```
array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

[117]:
```
display_accuracy(y_test, rf_predictions)
```

```
The overall accuracy of RF is 96.64%
```

[117]:
```
0.966414380321665
```

[118]:
```
prediction_results(y_test, rf_predictions)
```

```
Wrongly predicted letters were :
Counter({'B': 30, 'R': 18, 'F': 16, 'E': 13, 'O': 13, 'G': 13, 'H': 12, 'Q': 12,
'S': 9, 'D': 9, 'K': 8, 'X': 7, 'M': 7, 'Y': 6, 'T': 6, 'I': 6, 'U': 6, 'C': 5,
'V': 4, 'J': 4, 'P': 3, 'W': 3, 'N': 3})
The actual letters that were wrongly predicted were :
Counter({'L': 15, 'K': 15, 'H': 15, 'G': 14, 'R': 14, 'V': 13, 'P': 13, 'N': 12,
'I': 10, 'C': 9, 'J': 9, 'E': 9, 'B': 9, 'Y': 8, 'Q': 8, 'F': 8, 'O': 8, 'S': 7,
'D': 4, 'W': 3, 'Z': 3, 'T': 2, 'X': 2, 'U': 2, 'M': 1})
```

[119]:
```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(max_depth = 200,random_state = 0)
clf.fit(X_train, y_train)
```

[119]:
```
DecisionTreeClassifier(max_depth=200, random_state=0)
```

[120]:
```
predictions = clf.predict(X_test)
```

[121]:
```
accuracy = clf.score(X_test, y_test)
print(f'The Model Accuracy is {round(accuracy*100,2)}%')
```

```
The Model Accuracy is 87.59%
```

The base descision tree model has an accuracy of around ± 87% - I could imploy methods to improve this model however due to the original model accuracy being low, and the general understanding that random forest tree's will normally always be more accurate - my improvements may only add ± 5% to the models accuracy which would still be inferior to the current improvement I have made with my rf model.

Final comments: Although my improved RF solution can generate upwards of 96.1% accuracy, there is still one improvement I would make. I would ideally cluster the dataset, this would results in the dataset being shaped similar to {A1, A2, B1, B2...} - this would allow for a more accurate model by allowing certain trees to be correctly made.