Katholische Universität Eichstätt-Ingolstadt

"Campus Ingolstadt"

KATHOLISCHE UNIVERSITÄT
EICHSTÄTT–INGOLSTADT

Letter Image Recognition Data

Faustino Vazquez Gabino| 203961

Alessandro Sapia | 290742

Index

Abstract

Introduction

To get a better overview about the topic we should take a look at what letter recognition is actually about? What are some difficulties about this topic?.
Letter recognition is the ability to identify a letter that is being shown or to pick out a specific letter from among a data set containing different letters.
This topic finds its way in numerous domains, ranging from document analysis and text education to intelligent character input systems. At its core, letter recognition involves developing algorithms and models that can learn to distinguish and classify different letters based on their visual features and patterns. With the help of different features like horizontal straight or closed curve, letter recognition aims at identifying a set of letters. Machine learning models learn from vast datasets of labeled letters, where each letter image is associated with its corresponding class. By analyzing the data, those models can manage to learn the features that differentiate one letter from another.
Since letters can be so variable in their style and size it can be difficult for recognition models to handle such various writing variations.
A huge problem is that many letters share similarities (e.g. N, M ; F, P ; Q, O) which can be really impactful on the accuracy of a machine learning model.
In order of that we tested different Machine learning Models to compare and see which will perform the best.

Frame the Problem and Look at the Big Picture

The objective of this project is to analyze a file with a large amount of integer data which provides information about the magnitude of black and white rectangular pixels as characteristics of some of the 26 letters of the English alphabet.

The information is based on 20 different letter backgrounds and each was randomly placed in a file of 20,000 entries. Each entry has 16 attributes which are characterized by a magnitude between 0 and 15.

Within the information from the file, it was recommended to train on the first 16000 items and then use the resulting model to predict the letter category for the remaining 4000

The performance of the code will be measured by the accuracy and precision rate as well as the F1_Score. We are going to compare the results of the different models by the end of this project. Our goal is to reach at least 90% in every metric given.

The problem will be solved with the learnings and information learned during each Lecture of the course, plus additional research and insights from team members.

The members decided to formulate two questions that they would like to be answered when solving the problem, the questions are:

- **How well can one predict the letter based on the other data?**

Features

The Dataset has 17 attributes with the following characteristics:

	Name	Use	Type	% of missing

				values
1	lettr	horizontal position of box	(26 values from A to Z)	0
2	x-box	horizontal position of box	integer	0
3	y-box	vertical position of box	integer	0
4	width	width of box	integer	0
5	high	height of box	integer	0
6	onpix	total # on pixels	integer	0
7	x-bar	mean x of on pixels in box	integer	0
8	y-bar	mean y of on pixels in box	integer	0
9	x2bar	mean x variance	integer	0
10	y2bar	mean y variance	integer	0
11	xybar	mean x y correlation	integer	0
12	x2ybr	mean of x * x * y	integer	0
13	Xy2br	mean of x * y * y	integer	0
14	x-ege	mean edge count left to right	integer	0
15	xegvy	correlation of x-ege with y	integer	0
16	y-ege	mean edge count bottom to top	integer	0
17	yegvx	correlation of y-ege with x	integer	0

Table 1.0 Characteristics of the attributes

It is known that there are 0 missing values thanks to the method *'isnull'*

Important libraries

```
#Importing the first libraries
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import seaborn as sns # creation of univariate and bivariate plots
import matplotlib.pyplot as plt # comprehensive data visualization
```

These three important libraries are really important, seaborn and matplotlib are good for plotting graphs that will help us to take a look at the features and how they behave in the data set from our code. The pandas library will help us with file data processing.

```
# Column names
column_names = [
    'letter', 'x-box', 'y-box', 'width', 'high', 'onpix', 'x-bar',
'y-bar',
    'x2bar', 'y2bar', 'xybar', 'x2ybr', 'xy2br', 'x-ege', 'xegvy',
'y-ege', 'yegvx'
]
##########################################################################
###
# Convert the data to a format you can easily manipulate (without changing
the data itself), e.g. a Pandas DataFrame
data = pd.read_csv('letter-recognition.data', delimiter=',',
names=column_names)
data.to_csv("letter-recognition.csv",index= False)
```

With these lines of code, it has loaded the Dataset from the 'letter-recognition.data' file into a pandas DataFrame, a format we can easily manipulate named *'data'* with the appropriate column names, and we export the data into a csv file for better manipulation.

```
# Looking at the distribution of the different features from the data
print(data.info()) # Basic Information about the data
print("\nSample (10)\n",data.head(10)) # View a sample data (100)
print("\nStatistics information\n",data.describe()) # Statistic
information about the data
```

Sample

letter	x-box	y-box	width	high	onpix	x-bar	y-bar	x2bar	y2bar	xybar	x2ybr	xy2br	x-ege	xegvy	y-ege	yegvx
T	2	8	3	5	1	8	13	0	6	6	10	8	0	8	0	8
I	5	12	3	7	2	10	5	5	4	13	3	9	2	8	4	10
D	4	11	6	8	6	10	6	2	6	10	3	7	3	7	3	9
N	7	11	6	6	3	5	9	4	6	4	4	10	6	10	2	8
G	2	1	3	1	1	8	6	6	6	6	5	9	1	7	5	10
S	4	11	5	8	3	8	8	6	9	5	6	6	0	8	9	7
B	4	2	5	4	4	8	7	6	6	7	6	6	2	8	7	10
A	1	1	3	2	1	8	2	2	2	8	2	8	1	6	2	7
J	2	2	4	4	2	10	6	2	6	12	4	8	1	6	1	7
M	11	15	13	9	7	13	2	6	2	12	1	9	8	1	1	8

Table 2.0 First 10 samples from the Data Set

Sample																
letter	x-box	y-box	width	high	onpix	x-bar	y-bar	x2bar	y2bar	xybar	x2ybr	xy2br	x-ege	xegvy	y-ege	yegvx
T	2	8	3	5	1	8	13	0	6	6	10	8	0	8	0	8
I	5	12	3	7	2	10	5	5	4	13	3	9	2	8	4	10
D	4	11	6	8	6	10	6	2	6	10	3	7	3	7	3	9
N	7	11	6	6	3	5	9	4	6	4	4	10	6	10	2	8
G	2	1	3	1	1	8	6	6	6	6	5	9	1	7	5	10
S	4	11	5	8	3	8	8	6	9	5	6	6	0	8	9	7
B	4	2	5	4	4	8	7	6	6	7	6	6	2	8	7	10
A	1	1	3	2	1	8	2	2	2	8	2	8	1	6	2	7
J	2	2	4	4	2	10	6	2	6	12	4	8	1	6	1	7
M	11	15	13	9	7	13	2	6	2	12	1	9	8	1	1	8

Table 3.0 Statistics values from the Data Set

General Information				
#	Column	Non-Null	Null Count	Dtype
0	letter	20000	non-null	object
1	x-box	20000	non-null	int64
2	y-box	20000	non-null	int64
3	width	20000	non-null	int64
4	high	20000	non-null	int64
5	onpix	20000	non-null	int64
6	x-bar	20000	non-null	int64
7	y-bar	20000	non-null	int64
8	x2bar	20000	non-null	int64
9	y2bar	20000	non-null	int64
10	xybar	20000	non-null	int64
11	x2ybr	20000	non-null	int64
12	xy2br	20000	non-null	int64
13	x-ege	20000	non-null	int64
14	xegvy	20000	non-null	int64
15	y-ege	20000	non-null	int64
16	yegvx	20000	non-null	int64

Table 4.0 Basic Information about the Data Set

With this lines of code we will print out important information about the features, using the .info() method that provides basic information about the DataFrame, the .data() head will print out the first 10 samples from the DataFrame and finally the .describe() method will print out the statistics information about the data.

After analyzing these basic tables we can see we are working with numerical features with a magnitude from 0 to 15 and we are working with supervised data because we have a target variable (letter) and input features.

As it said we are working with a pandas DataFrame, we got in total 20,000 entries and 17 columns as described in the *Table 1.0* the first column is always the target variable while the rest are always the features from each letter.

Visualization

```
#Sorted data
sorted_data = data.sort_values('letter')
##########################
# Count of each letter(sorted)
print("\n",sorted_data['letter'].value_counts().sort_index())
##########################
# Visualization
sns.countplot( x = sorted_data['letter'])
plt.xlabel('Letter')
plt.ylabel('Count')
plt.title('Count of Letters Sorted')
plt.show()
data.hist(bins=40,figsize=(12, 8)) #
plt.show()
```

First we sorted the data to have a better visualization with each letter of the data, then we printed the count of each letter. We use the countplot() method from the seaborn library to have a look at each letter and its count.
And finally we print a histogram of the counts of the magnitude for each feature, with the statistical information and the histogram we can do a small analysis about the data.

Image 1.0 Counted graph from the numbers of times for each class (letter)

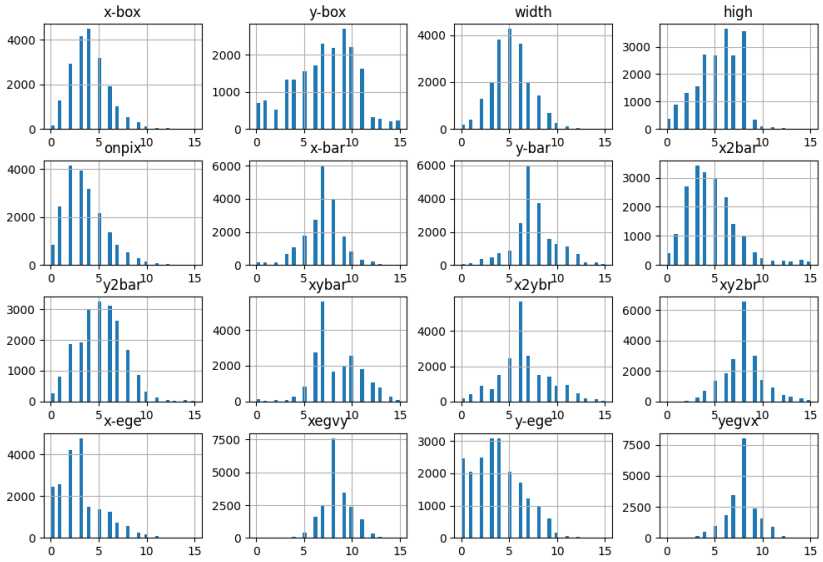

Image 2.0 Histogram of the counts of features

Here we can see the magnitude of the letters in relation to the amount of letters for each feature. We notice that y-box for example has a much broader spectrum than yegwx.

Splitting and training the data

```
from sklearn.model_selection import train_test_split

# Importing data
data = pd.read_csv('letter-recognition.csv') # Importing the DataFrame
X = data.drop("letter",axis= 1)
y = data['letter'] # Taking the letter column as our target value

# Training the first 12,000 entries and validating and testing the rest
8000
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4,
random_state=42, stratify=y)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp,
test_size=0.5, random_state=42, stratify=y_temp)
```

First we import the DataFrame and set to the y variable as the target variable the column “letter” from the DataFrame.

Using the train_test_split() from the sklearn library we trained the first 12,000 entries and let the rest 8,000 for validation and testing, we use the validation subset to check how the models behave and prevent overfitting, we used 42 as random_state because it ensures reproducibility, facilitates result comparison, and avoids potential biases introduced by random processes in machine learning algorithms and finally we use the stratify method even though our classes our classes are not really imbalanced we considered it is a great idea to have a better perfomance

Data Analysis

```
#########################
import seaborn as sns # creation of univariate and bivariate plots
import matplotlib.pyplot as plt # comprehensive data visualization
#########################
# Importing data
from _2_training_testing import X_train
trained_data = X_train
#########################
# Plotting Histogram
trained_data.hist(figsize = (12,8))
#########################
# Let us see how the different features behave for the different letters
# This is not so easy to do in an interpretable way for all 26 letters
# at the same time, so we do it here only for 'A vs not A'.
sns.set(font_scale = 1)
fig, ax = plt.subplots(4, 4, figsize=(15, 15))

features = list(trained_data.columns)[1:]

data_plot = trained_data.copy()
data_plot['is_A'] = (data_plot['letter'] == 'A')

for name, a in zip(features, ax.ravel()):
  hist = sns.histplot(data=data_plot,
                      x=name,
                      stat='percent',
                      common_norm=False,
                      bins=16,
                      shrink=0.7,
                      discrete=True,
                      hue="is_A",
                      multiple='dodge',
                      ax = a)
plt.show()
```

Plotting the features to analyze the data for the letter A.

```
# Plotting Box Plots
plt.figure(figsize=(10, 6))
sns.boxplot(data=trained_data.iloc[:, 1:])
plt.title('Box Plot of Numerical Features')
plt.xlabel('Features')
plt.ylabel('Values')
plt.xticks(rotation=45)
plt.show()
# Plotting Correlation Heatmap
plt.figure(figsize=(15, 10))
corr_matrix = trained_data.iloc[:, 1:].corr()
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

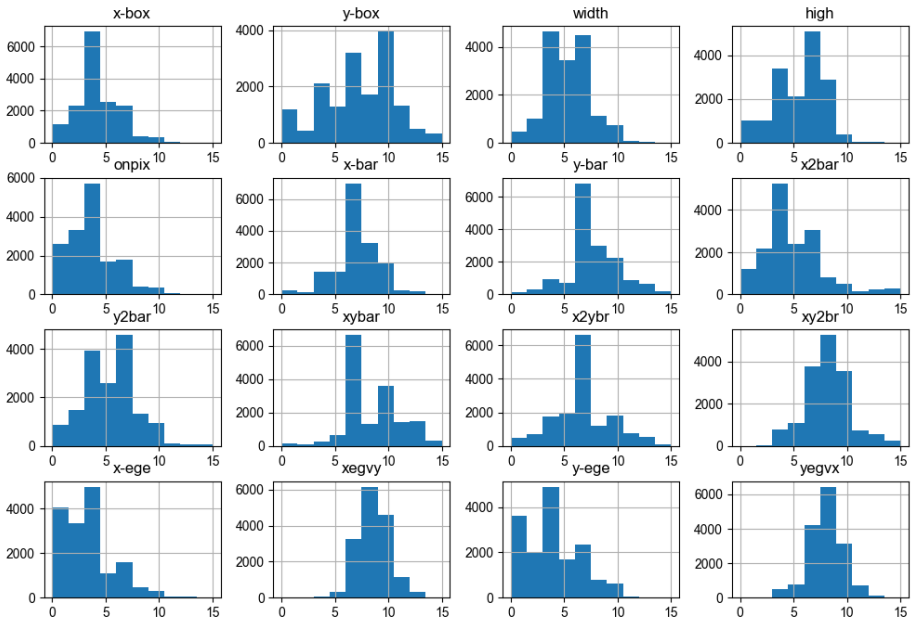

Image 3.0 Histogram of the counts of features

Analyzing the different features in aspect of magnitude and amount of letters to get a better view about the impact, which the features have on the data set, as we can see there are some features where the data is oriented in different means points.

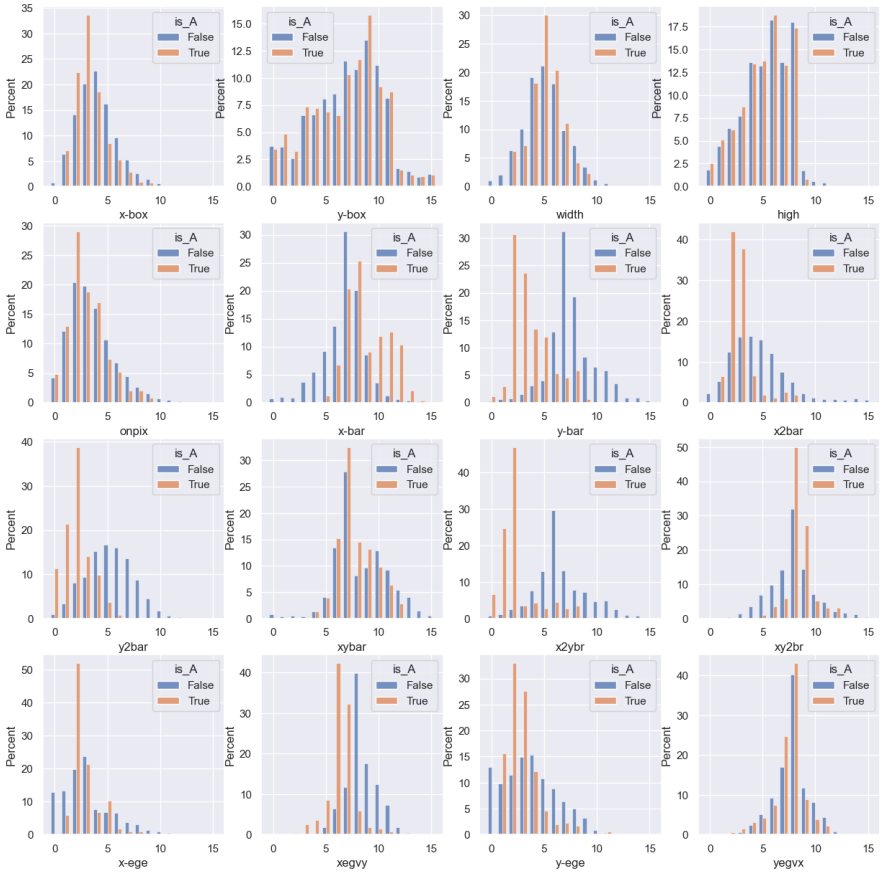

Image 4.0 Histogram behaviors features (A vs not A)

This diagram shows the contrast of is_A and isnot_A for all the features
We analyzed every letter starting from A - Z.
We will continue with the given features and see by the end how accurate our models predicted the letters.

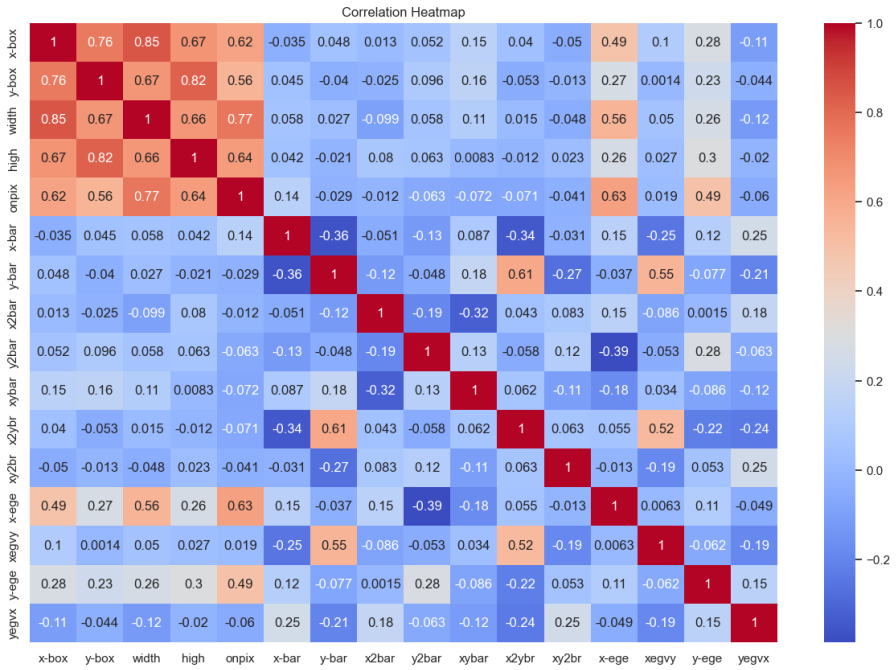

Image 5.0 Correlation HeatMap features

It turned out, there is a high correlation between some features in the upper left corner, specifically on the on-pix, high, width, y-box and x-box features.

Missing values:	
Feature	**MV**
x-box	0
y-box	0
width	0
high	0
onpix	0
x-bar	0
y-bar	0
x2bar	0
y2bar	0
xybar	0
x2ybr	0
xy2br	0
x-ege	0
xegvy	0
y-ege	0
yegvx	0

Table 5.0 Missing values

Fortunately our dataset has 0 missing values, saving us time and resources to have a better analysis, improve our models performance.

Preparing Data

```
from sklearn.preprocessing import StandardScaler

# Standardizing features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
X_test = scaler.transform(X_test)

X_train = pd.DataFrame(X_train)
X_val = pd.DataFrame(X_val)
X_test = pd.DataFrame(X_test)

# Exporting the data into csv files
X_train.to_csv("final_letter_recognition_X_train.csv",index = False)
X_test.to_csv("final_letter_recognition_X_test.csv",index = False)
X_val.to_csv("final_letter_recognition_X_val.csv",index = False)

y_train.to_csv("final_letter_recognition_y_train.csv",index = False)
y_test.to_csv("final_letter_recognition_y_test.csv",index = False)
y_val.to_csv("final_letter_recognition_y_val.csv",index = False)
```

StandardScalar removes the mean and scales each feature/variable to unit variance.
Standardizing the features in order to measure them in a common unit.
To get further with our work we need to export our given data into csv files.

Models Selection

To see different outcomes we tested a few ML-Models. In order to do that we need to understand properly how these models work. In the following we describe which models we used, why they are useful for our project.

```
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix

from sklearn.model_selection import cross_val_score

#Reshape into a one-dimensional array, ensuring compatibility with the
classification algorithms
y_train = y_train.values.ravel()
y_val = y_val.values.ravel()
```

To measure our results we imported different metrics like the accuracy_score to measure how many times the model was correct overall, precision_score to measure how good our model is at predicting a specific category, the F1_Score which combines precision and recall to provide a balanced assessment of a model performance and confusion matrix because we are working with a multiclass problem it easier to analyze and visualize the performance of the models.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
```

We import Pipeline to ensure that data processing and modeling are performed correctly. Pipeline summarizes several consecutive data processes. StandardScaler in order to scale each feature and GridSearchCV to find the optimal parameters for our models to get the best results possible.

Random Forest

In Random Forest, the "forest" consists of a collection of decision trees. Each decision tree is constructed by randomly selecting a subset of features from the dataset and creating a tree based on these features.
But why do we need Random Forest if we already have decision trees that we can work with?
Decision trees are highly sensitive to the training data which could result in high variance, different as in Random Forest. It is a collection of multiple random decision trees and it is much less sensitive to the training data.

Random Forest Model visualized:

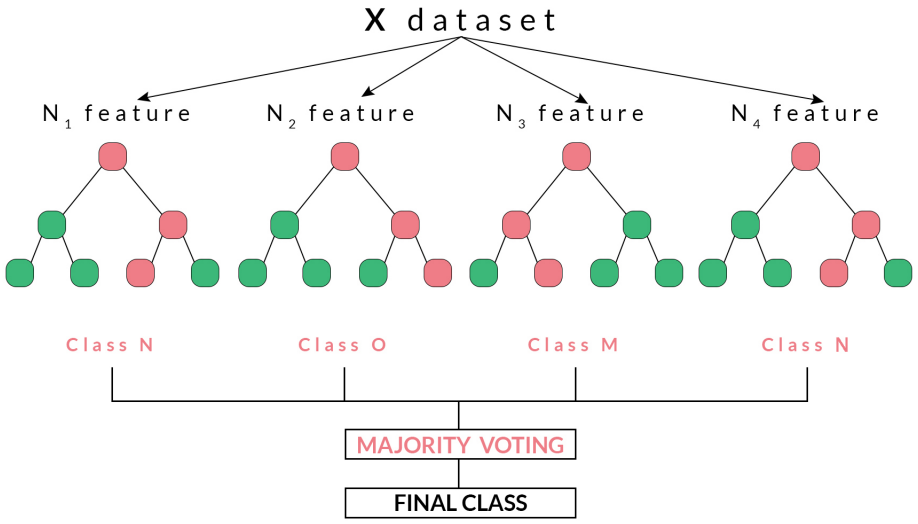

Image 6.0 Example Random Forest

This model was chosen because is really effectively working with categorical problems and numerical features, it is also good to combine multiple decisions trees, which results in better predictive robustness and performance,

Code

```
# Pipeline for Random Forest
rf_pipeline = Pipeline([
    ('classifier', RandomForestClassifier())
])

# Define hyperparameters and their search space
param_grid_rf = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [None, 10, 20, 30],
    'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2, 4],
    'classifier__max_features': ['auto', 'sqrt', 'log2']
}
```

We created a pipeline for the Random Forest model and imported the RandomForestClassifier(). With param_grid_rf we defined the hyperparameters, n_estimators we try to give as much possible to have a better performance, the max depth were between 0 and 30 to find the best generalization, the min samples split to capture fine details and max features to look for the best split.

```
# GridSearchCV object with cross-validation
rf_grid_search = GridSearchCV(rf_pipeline, param_grid_rf, cv=5, n_jobs=-1,
scoring='f1_micro')
rf_grid_search.fit(X_train, y_train)

# Best estimator and parameters
best_rf_model = rf_grid_search.best_estimator_
print("Best hyperparameters: ", rf_grid_search.best_params_)

y_pred_val_rf = best_rf_model.predict(X_val)
y_pred_train_rf = best_rf_model.predict(X_train)

train_f1_rf = f1_score(y_train, y_pred_train_rf, average='micro')
train_accuracy_rf = accuracy_score(y_train, y_pred_train_rf)
train_precision_rf = precision_score(y_train, y_pred_train_rf,
average='micro')
```

```
print("\nTrained scores (Random Forest):")
print("F1-Score: ", train_f1_rf)
print("Accuracy: ", train_accuracy_rf)
print("Precision: ", train_precision_rf)

val_f1_rf = f1_score(y_val, y_pred_val_rf, average='micro')
val_accuracy_rf = accuracy_score(y_val, y_pred_val_rf)
val_precision_rf = precision_score(y_val, y_pred_val_rf, average='micro')
print("\nValidation scores (Random Forest):")
print("F1-Score: ", val_f1_rf)
print("Accuracy: ", val_accuracy_rf)
print("Precision: ", val_precision_rf)
```

Finally we use a GridsearchCV with 5 folds cross validations to explore various combinations from the hyperparameters and we set the scoring in f1_micro
After running the code we see that the best hyperparameters are a max_depth in 0, max_features on log_2, a min_samples_leaf of 1, a min_samples_split in 2 and a n_estimators in 200.

```
# Function to plot the confusion matrix with letters
def plot_confusion_matrix_with_labels(confusion_matrix, class_labels,
title):
   plt.figure(figsize=(12, 10))  # Adjust the figure size as needed
   sns.heatmap(confusion_matrix, annot=True, fmt="", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
   plt.xlabel('Predicted')
   plt.ylabel('Actual')
   plt.title(title)
class_labels = [chr(65 + i) for i in range(26)]  # A=65, B=66, ..., Z=90

# Confusion matrix for Random Forest (Validation)
confusion_matrix_val_rf = confusion_matrix(y_val, y_pred_val_rf)
plot_confusion_matrix_with_labels(confusion_matrix_val_rf, class_labels,
"Confusion Matrix (Random Forest - Validation)")
```

With plot_confusion_matrix_with_labels() we created a function which plots a confusion matrix with letters.

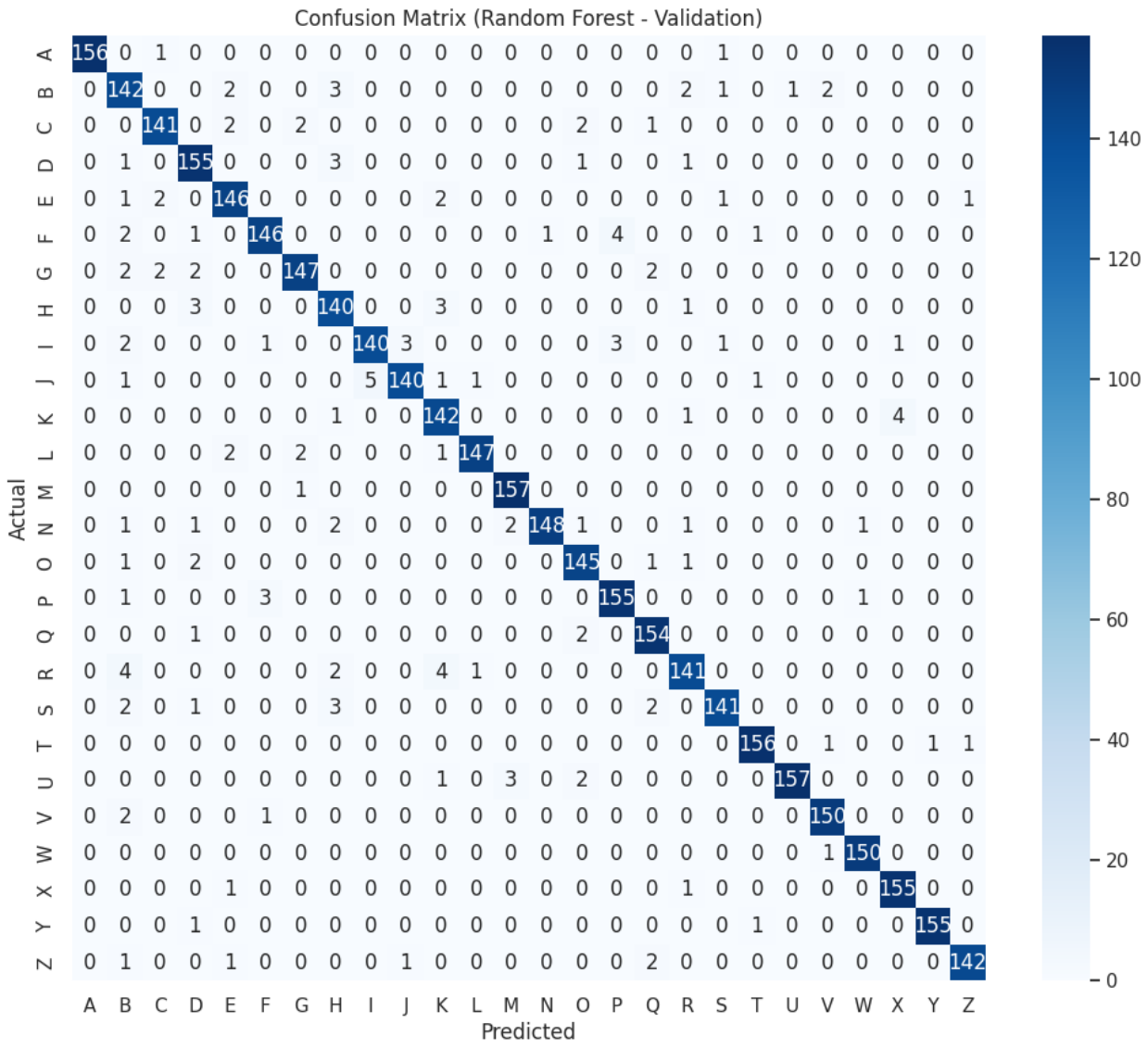

Image 7.0 Confusion Matrix (Random Forest Model)

We can see in the confusion matrix even though we have a great performance in the model resulting in one of the best models for the data set.

Trained scores (Random Forest):
F1-Score: 1.0
Accuracy: 1.0
Precision: 1.0

Validation scores (Random Forest):
F1-Score: 0.962
Accuracy: 0.962

Precision: 0.962

We can see that this models achieves a perfect scores in the metrics so this means the model has learned to fit the training data really good, the 0.962 scores for the validation set can tell us that we have a good generalization, but we can be dealing with a small possibility of overfitting because even though it's not a big difference between the scores there is a possibility.

Support Vector Machine (SVM)

SVM tries to find the smallest distance between data points and concurrently wants the decision boundary to be as large as possible. In the sense it maximizes the distance in each category, so called "margin". The margin describes the distance between the hyperplane and the closest point from either set. Points, which fall exactly on the margin, are called supporting vectors. To find the optimal hyperplane SVM requires a training set or a set of points which are already labeled with the correct category

Linear Model: **Nonlinear Model:**

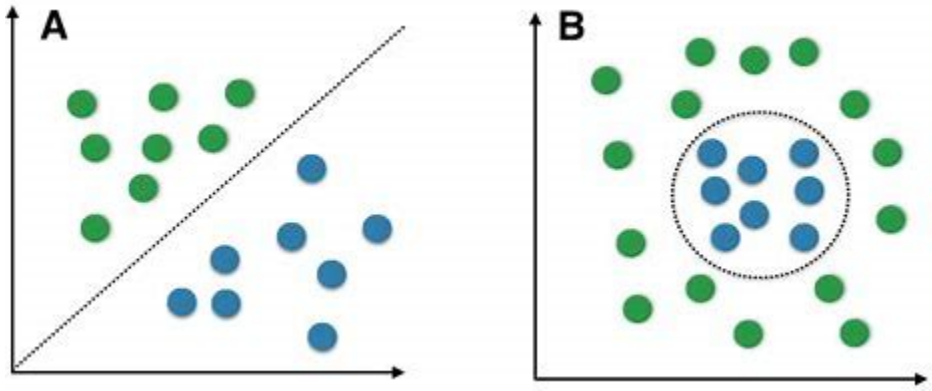

Image 8.0 SVM example

In many applications the points can not be separated by a hyperplane. A common workaround is to find a separating hyperplane in the higher dimensional space and project it back to the original space. This process can be done with Kernel functions, which transform nonlinear spaces into linear spaces so data can still be classified.

Code

```
# Pipeline for SVM
svm_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', SVC())
])
```

```
# Hyperparameters and their search space
param_grid_svm = {
    'classifier__C': [0.1, 1.0, 10.0],
    'classifier__kernel': ['linear', 'rbf'],
    'classifier__gamma': ['scale', 'auto', 0.1, 1.0]
}

# GridSearchCV object with cross-validation
svm_grid_search = GridSearchCV(svm_pipeline, param_grid_svm, cv=5,
n_jobs=-1, scoring='f1_micro')
svm_grid_search.fit(X_train, y_train)
```

We created a pipeline for the SVM model by setting the scale in the StandardScaler and the classifier with our SVM model . For the hyperparameters we decided to set,a regularization strength(C) between 0.1 and 10.0, the kernel between linear and rbf and the gamma we set between a scale, auto or even 0.1 or 1.
Finally we use a GridsearchCV with 5 folds cross validations to explore various combinations from the hyperparameters and we set the scoring in f1_micro

```
# Best estimator and parameters
best_svm_model = svm_grid_search.best_estimator_
print("Best hyperparameters: ", svm_grid_search.best_params_)

y_pred_val_svm = best_svm_model.predict(X_val)
y_pred_train_svm = best_svm_model.predict(X_train)

val_f1_svm = f1_score(y_val, y_pred_val_svm, average='micro')
val_accuracy_svm = accuracy_score(y_val, y_pred_val_svm)
val_precision_svm = precision_score(y_val, y_pred_val_svm,
average='micro')
print("\nValidation scores (Support Vector Machine ):")
print("F1-Score: ", val_f1_svm)
print("Accuracy: ", val_accuracy_svm)
print("Precision: ", val_precision_svm)

train_f1_svm = f1_score(y_train, y_pred_train_svm, average='micro')
train_accuracy_svm = accuracy_score(y_train, y_pred_train_svm)
train_precision_svm = precision_score(y_train, y_pred_train_svm,
average='micro')
```

```
print("\nTrained scores (Support Vector Machine ):")
print("F1-Score: ", train_f1_svm)
print("Accuracy: ", train_accuracy_svm)
print("Precision: ", train_precision_svm)
```

After running the code we see that the best hyperparameters are a regularization strength in 10.0, the classifier gamma in 0.1 and the kernel rbf.

```
# Function to plot the confusion matrix with letters
def plot_confusion_matrix_with_labels(confusion_matrix, class_labels,
title):
    plt.figure(figsize=(12, 10))  # Adjust the figure size as needed
    sns.heatmap(confusion_matrix, annot=True, fmt="", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(title)
class_labels = [chr(65 + i) for i in range(26)]  # A=65, B=66, ..., Z=90
confusion_matrix_val_svm = confusion_matrix(y_val, y_pred_val_svm)

# Confusion matrix for Support Vector Machine (Validation)
plot_confusion_matrix_with_labels(confusion_matrix_val_svm, class_labels,
"Confusion Matrix (Support Vector Machine  - Validation)")
```

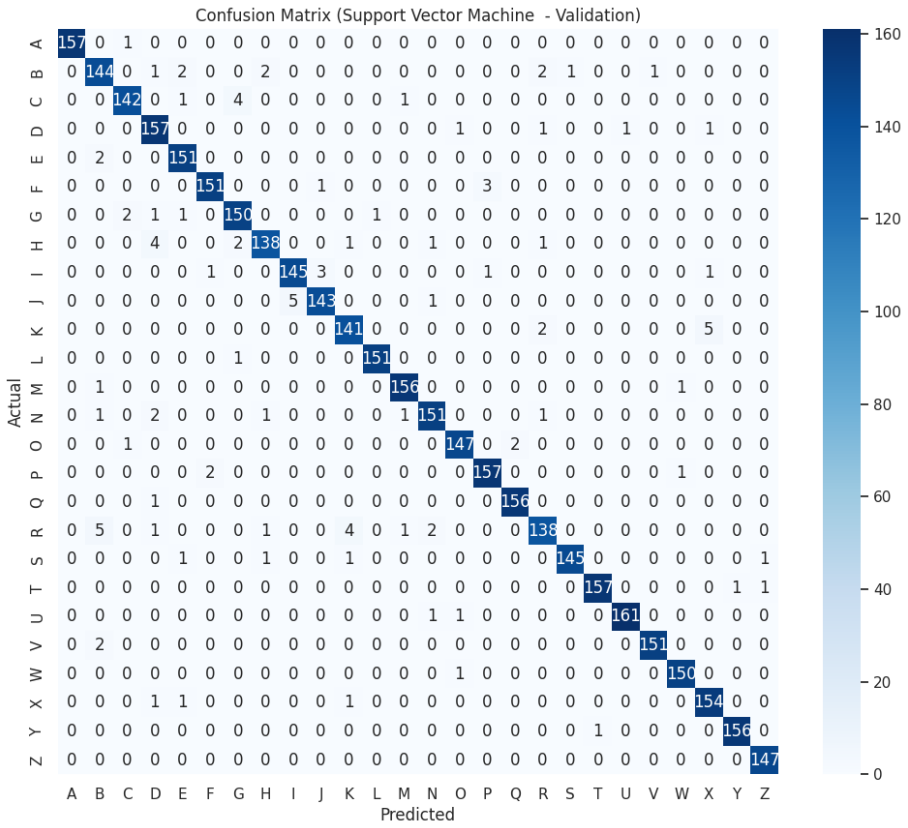

Image 9.0 Confusion Matrix (Support Vector Machine)

We can see in the confusion matrix even that we have a really great performance with the validation subset, the best model until now.

Validation scores (Support Vector Machine):
F1-Score: 0.974
Accuracy: 0.974
Precision: 0.974

Trained scores (Support Vector Machine):
F1-Score: 0.9971666666666666
Accuracy: 0.9971666666666666
Precision: 0.9971666666666666

After analyzing the scores we can see that we have the strongest performance for this model for validation scores and almost perfect for trained scores, so this indicates this model have a good generalization for unseen data and doesn't seem to be an indicator of overfitting, which means we are looking for a good balanced model with a good precision.

K Nearest Neighbor

This Model looks at the "closest area" of the new data and checks what class of objects are there. A given parameter K determines how many closest neighbors the model is supposed to look at. Then the point is assigned to the class most common among its K nearest neighbor.

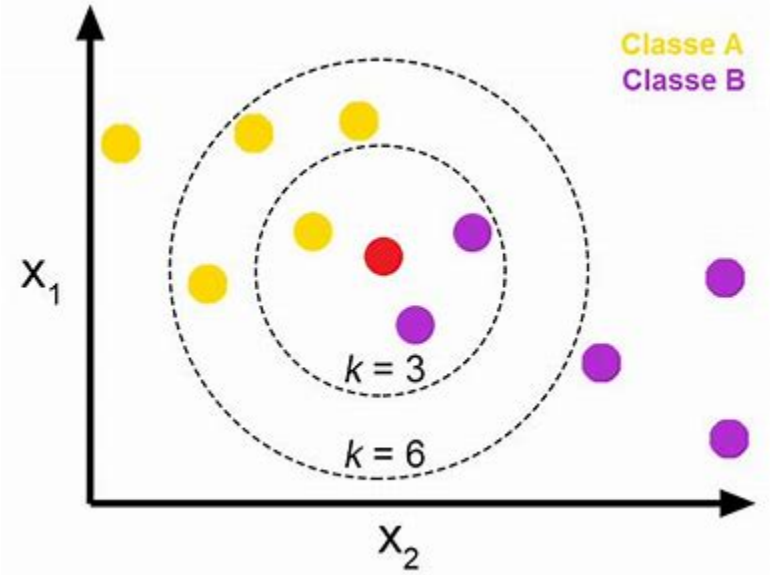

Image 10.0 K Nearest Neighbor Example

Code

```
# Pipeline for K-Nearest Neighbors
knn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', KNeighborsClassifier())
])

# Hyperparameters and their search space
param_grid_knn = {
    'classifier__n_neighbors': [2,3,4,5],
    'classifier__weights': ['uniform', 'distance'],
    'classifier__algorithm': ['brute', 'kd_tree'],
    'classifier__p': [1, 2]
}

# GridSearchCV object with cross-validation and fitting to the trained
data
knn_grid_search = GridSearchCV(knn_pipeline, param_grid_knn, cv=5,
n_jobs=-1, scoring='f1_micro')
knn_grid_search.fit(X_train, y_train)
```

We created a pipeline for the KMM model by setting the scale in the StandardScaler and the classifier with our KN Classifier model . For the hyperparameters we decided to set the numbers of neighbors between 2 and 5, a weight even uniform or distance, an algorithm brute or kd_tree and a classifier p between 1 and 2.

Finally we use a GridsearchCV with 5 folds cross validations to explore various combinations from the hyperparameters and we set the scoring in f1_micro

```
# Best estimator and parameters
best_knn_model = knn_grid_search.best_estimator_
print("Best hyperparameters: ", knn_grid_search.best_params_)

y_pred_val_knn = best_knn_model.predict(X_val)
y_pred_train_knn = best_knn_model.predict(X_train)

val_f1 = f1_score(y_val, y_pred_val_knn, average= 'micro')
val_accuracy = accuracy_score(y_val, y_pred_val_knn)
val_precision = precision_score(y_val, y_pred_val_knn, average= 'micro')
```

```
print("\nValidation scores:\nF1-Score: ", val_f1, "\nAccuracy: ",
val_accuracy, "\nPrecision: ", val_accuracy)

train_f1 = f1_score(y_train, y_pred_train_knn, average= 'micro')
train_accuracy = accuracy_score(y_train, y_pred_train_knn)
train_precision = precision_score(y_train, y_pred_train_knn, average=
'micro')
print("\nTrained scores:\nF1-Score: ", train_f1, "\nAccuracy: ",
train_accuracy, "\nPrecision: ", train_precision)
```

After running the code we see that the best hyperparameters are a brute algorithm, 4 neighbors and distance weight.

```
# Function to plot the confusion matrix with letters
def plot_confusion_matrix_with_labels(confusion_matrix, class_labels,
title):
    plt.figure(figsize=(12, 10))  # Adjust the figure size as needed
    sns.heatmap(confusion_matrix, annot=True, fmt="", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.title(title)
class_labels = [chr(65 + i) for i in range(26)]  # A=65, B=66, ..., Z=90
confusion_matrix_val_knn = confusion_matrix(y_val, y_pred_val_knn)

# Confusion matrix for K-Nearest Neighbors (Validation)
plot_confusion_matrix_with_labels(confusion_matrix_val_knn, class_labels,
"Confusion Matrix (K-Nearest Neighbors - Validation)")
```

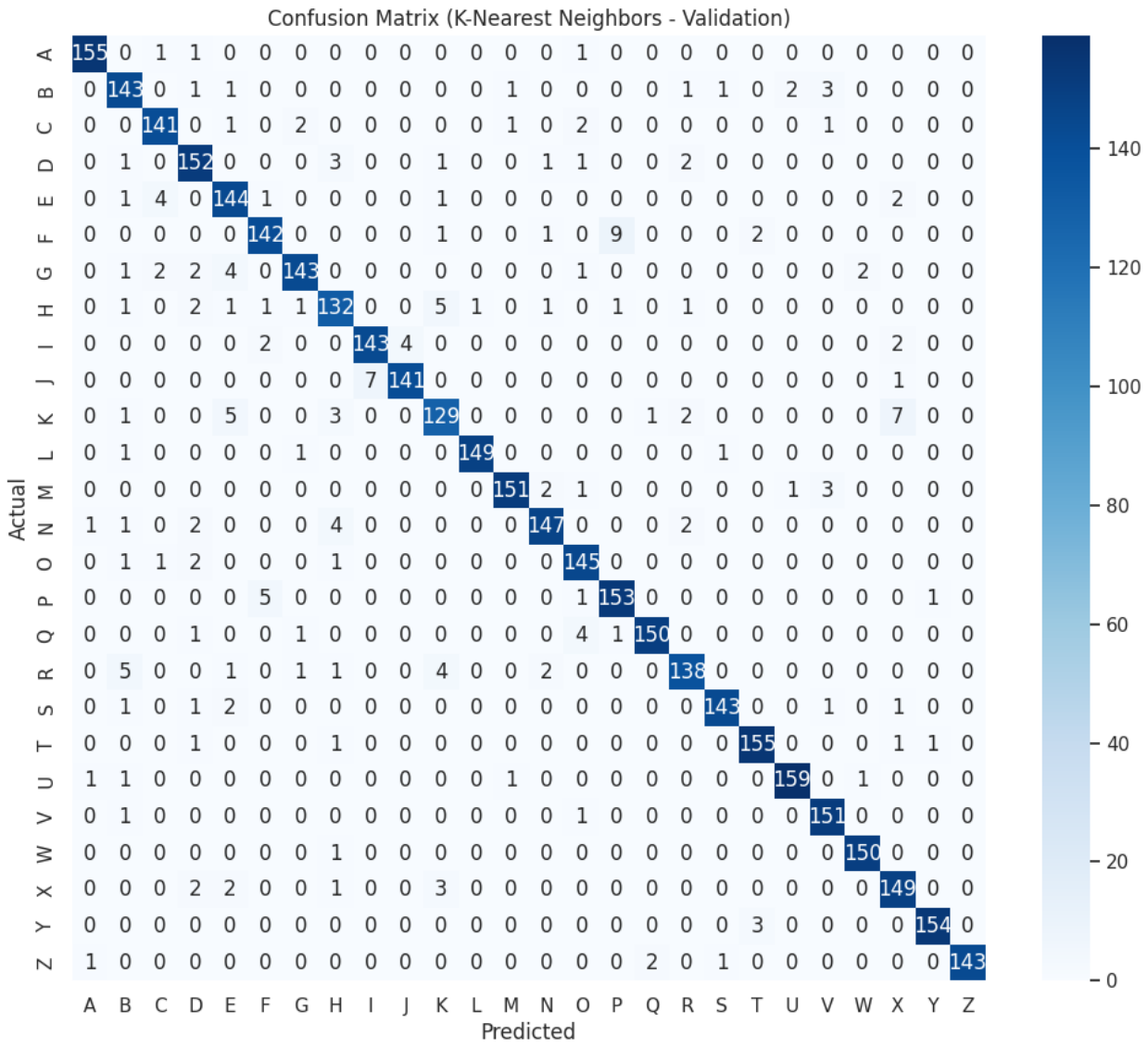

Image 11.0 K Nearest Neighbor Confusion Matrix

We can see in the confusion matrix even though we have a good performance it's not the best.

Validation scores:
F1-Score: 0.9505
Accuracy: 0.9505
Precision: 0.9505

Trained scores:
F1-Score: 1.0
Accuracy: 1.0
Precision: 1.0

We see we have a high training score even perfect so the model learned to fit the trained data and has a great generalization getting a 0.9505 score however we have a slight drop between both scores which means we can be dealing with an overfitting.

Neural Networks

Neural Networks are made up with layers of neurons, which are the core processing units of the network. The first layer is the "input layer" and receives the input. The "Output layer" predicts our final output. In between are "hidden layers", which perform most of the computations required by our network. Neurons of one layer are connected to neurons of the next layer through channels and are assigned to a numerical number called "weight".

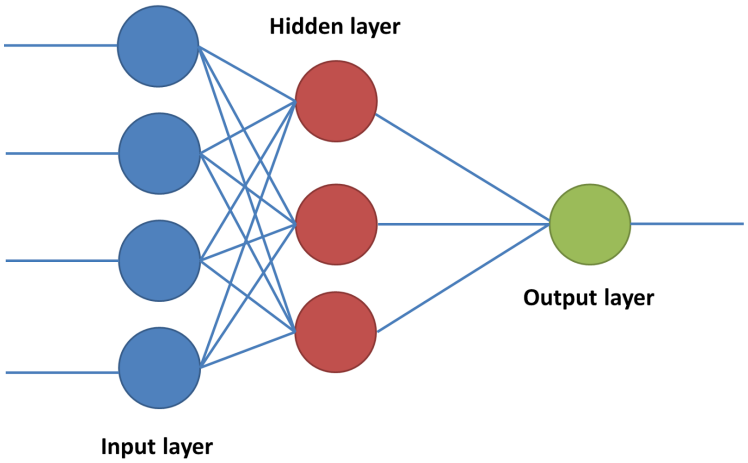

Image 12.0 Neural Networks Example

Code

```
# Pipeline for Neural Network
nn_pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', MLPClassifier())
])

# Hyperparameters and their search space for Neural Network
param_grid_nn = {
    'classifier__hidden_layer_sizes': [(50, 50), (100, 100), (50, 100,
50)],
    'classifier__activation': ['relu', 'tanh'],
    'classifier__alpha': [0.0001, 0.001, 0.01],
    'classifier__max_iter': [200, 300, 400],
}
```

We create a pipeline and set the search space for each hyperparameter in our Neural Network.

```
# GridSearchCV object with cross-validation and fitting to the trained
data
nn_grid_search = GridSearchCV(nn_pipeline, param_grid_nn, cv=5, n_jobs=-1,
scoring='f1_micro')
nn_grid_search.fit(X_train, y_train)

# Best estimator and parameters
best_nn_model = nn_grid_search.best_estimator_
print("Best hyperparameters: ", nn_grid_search.best_params_)

y_pred_val_nn = best_nn_model.predict(X_val)
y_pred_train_nn = best_nn_model.predict(X_train)

train_f1_nn = f1_score(y_train, y_pred_train_nn, average='micro')
train_accuracy_nn = accuracy_score(y_train, y_pred_train_nn)
train_precision_nn = precision_score(y_train, y_pred_train_nn,
average='micro')
print("\nTrained scores (Neural Network):")
print("F1-Score: ", train_f1_nn)
print("Accuracy: ", train_accuracy_nn)
print("Precision: ", train_precision_nn)

val_f1_nn = f1_score(y_val, y_pred_val_nn, average='micro')
val_accuracy_nn = accuracy_score(y_val, y_pred_val_nn)
val_precision_nn = precision_score(y_val, y_pred_val_nn, average='micro')
print("\nValidation scores (Neural Network):")
print("F1-Score: ", val_f1_nn)
print("Accuracy: ", val_accuracy_nn)
print("Precision: ", val_precision_nn)
```

With the help of rf_grid_search with 5 folds cross validations to explore various combinations from the hyperparameters and we set the scoring in f1_micro

```
# Function to plot the confusion matrix with letters
def plot_confusion_matrix_with_labels(confusion_matrix, class_labels,
title):
   plt.figure(figsize=(12, 10))  # Adjust the figure size as needed
   sns.heatmap(confusion_matrix, annot=True, fmt="", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
   plt.xlabel('Predicted')
   plt.ylabel('Actual')
   plt.title(title)
class_labels = [chr(65 + i) for i in range(26)]  # A=65, B=66, ..., Z=90
confusion_matrix_val_nn = confusion_matrix(y_val, y_pred_val_nn)

# Confusion matrix for Neural Network (Validation)
plot_confusion_matrix_with_labels(confusion_matrix_val_nn, class_labels,
"Confusion Matrix (Neural Network - Validation)")
```

With plot_confusion_matrix_with_labels() we created a function which plots a confusion matrix with letters.

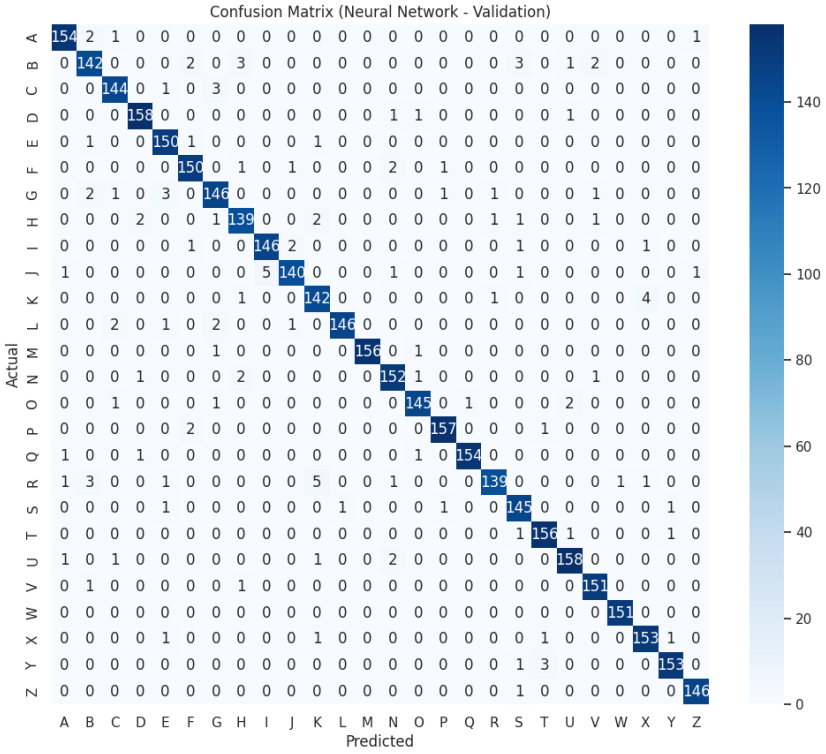

Image 13.0 Neural Networks Confusion Matrix

We can see in the confusion matrix we have a good performance which means we have a great generalization and its a balanced model.

Trained scores (Neural Network):
F1-Score: 1.0
Accuracy: 1.0
Precision: 1.0

Validation scores (Neural Network):
F1-Score: 0.96825
Accuracy: 0.96825
Precision: 0.96825

We see that we have a perfect training data set, and a great generalization because we got almost a 0.97 score for the validation set which can infer we have an overfitting . However, we consider this model to be well balanced because both scores are not the highest but with a balanced model.

Logistic Regression

Logistic Regression is a statistical technique that models the probability in a dataset given one or more variables. It starts with input data of any numeric type, but the output variable will be binary and be either 0 or 1, which leads to a boolean False or True Statement. With the help of an s-shaped function this model predicts if something is True or False and gives a final answer about the given data

Code

The code sets up a logistic regression model using a pipeline, optimizes its hyperparameters via GridSearchCV, and evaluates its performance on both the validation and training datasets. It also includes a function to visualize the confusion matrix with letters as labels.

```
# Pipeline for Logistic Regression
logistic_pipeline = Pipeline([
  ('scaler', StandardScaler()),
  ('classifier', LogisticRegression(multi_class= "multinomial"))
])

# Hyperparameters and their search space
param_grid_logistic = {
  'classifier__C': [0.001, 0.01, 0.1, 1.0, 10.0,50.0],  # Include smaller values for finer tuning
  'classifier__penalty': ['l2'],
  'classifier__class_weight': [None, 'balanced'],
  'classifier__max_iter': [100, 1000, 2000]  # Include different maximum iterations
}
# GridSearchCV object with cross-validation
logistic_grid_search = GridSearchCV(logistic_pipeline, param_grid_logistic, cv=5,
n_jobs=-1, scoring='f1_micro')
logistic_grid_search.fit(X_train, y_train)
```

We create a pipeline using Standard Scaler as a scaler, and applying the logistic regression model with multinomial regression for multiclass classification.
We set a grid search for hyperparameters with the parameters we considered important, For example, regularization strength(C) from 0.001 to 50.0, a penalty was chosen L2 because the L1 had problems with the data set, a class weight and different maximum iterations values. Finally we use a GridsearchCV with 5 folds cross validations to explore various combinations from the hyperparameters and we set the scoring in f1_micro

```
# Best estimator and parameters
best_logistic_model = logistic_grid_search.best_estimator_
print("Best hyperparameters: ", logistic_grid_search.best_params_)

y_pred_val_logistic = best_logistic_model.predict(X_val)
y_pred_train_logistic = best_logistic_model.predict(X_train)

val_f1 = f1_score(y_val, y_pred_val_logistic, average= 'micro')
val_accuracy = accuracy_score(y_val, y_pred_val_logistic)
val_precision = precision_score(y_val, y_pred_val_logistic, average= 'micro')
print("\nValidation scores:\nF1-Score: ", val_f1, "\nAccuracy: ", val_accuracy, "\nPrecision:
", val_accuracy)

train_f1 = f1_score(y_train, y_pred_train_logistic, average= 'micro')
train_accuracy = accuracy_score(y_train, y_pred_train_logistic)
train_precision = precision_score(y_train, y_pred_train_logistic, average= 'micro')
print("\nTrained scores:\nF1-Score: ", train_f1, "\nAccuracy: ", train_accuracy,
"\nPrecision: ", train_precision)
```

After running the code we see that the best hyperparameters are a regularization strength ('C') of 50.0, 'penalty' set to 'l2', 'class_weight' as None, and a maximum of 1000 iterations.

```
# Function to plot the confusion matrix with letters
def plot_confusion_matrix_with_labels(confusion_matrix, class_labels,
title):
   plt.figure(figsize=(12, 10))  # Adjust the figure size as needed
   sns.heatmap(confusion_matrix, annot=True, fmt="", cmap="Blues",
xticklabels=class_labels, yticklabels=class_labels)
   plt.xlabel('Predicted')
   plt.ylabel('Actual')
   plt.title(title)
class_labels = [chr(65 + i) for i in range(26)]  # A=65, B=66, ..., Z=90
confusion_matrix_val_logistic = confusion_matrix(y_val,
y_pred_val_logistic)

# Confusion matrix for Logistic Regression (Validation)
plot_confusion_matrix_with_labels(confusion_matrix_val_logistic,
class_labels, "Confusion Matrix (Logistic Regression - Validation)")
```

These line of code will give us a confusion matrix of the model which looks like this:

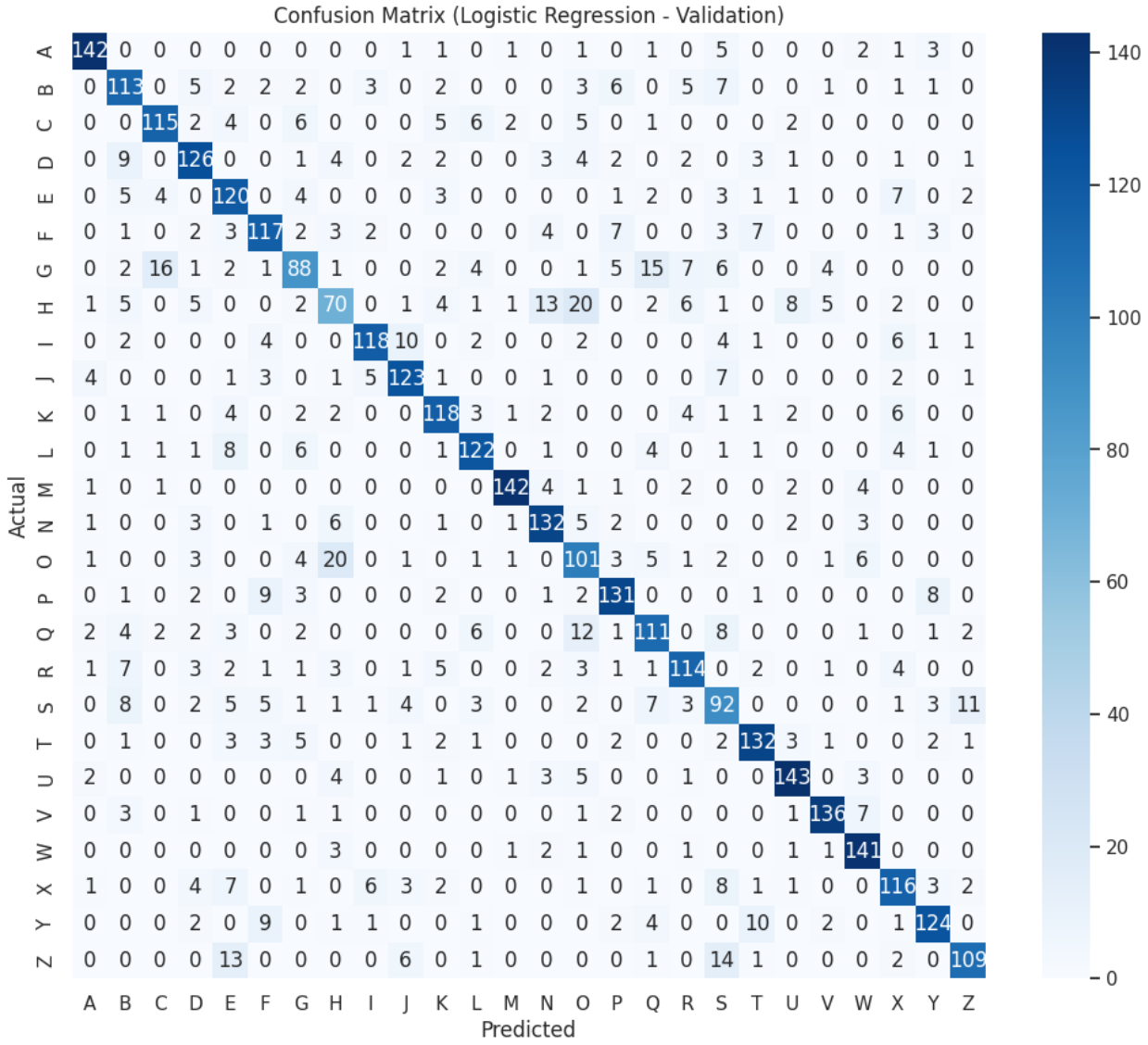

Image 14.0 Logistic Regression Confusion Matrix

We can see in the confusion matrix even though we have a reasonable performance the model is not the best suitable for this data set.

Finally we got these scores for the trained and validation subset.

Trained scores:
F1-Score: 0.7800833333333334
Accuracy: 0.7800833333333334
Precision: 0.78008333333333334

Validation scores:
F1-Score: 0.774
Accuracy: 0.774
Precision: 0.774

The performance is not the best compared to the others however we considered that we have a reasonable performance on both scores and they are really similar this indicates we have consistency which means there's no overfitting or underfitting.

Model Comparison

For model comparison we use the accuracy and precision score as well as the F1_Score in order to get a final overview of the different performances. We tried to choose the best parameters we could find for every model

Model	Precision	Accuracy	F1-Score
Random Forest	**0.962**	**0.962**	0.962
Support Vector Machine	0.974	0.974	0.974
K-Nearest Neighbor	**0.9505**	**0.9505**	0.9505
Neural Network	**0.96825**	**0.96825**	0.96825
Logistic Regression	**0.7800833**	**0.7800833**	0.7800833

We can clearly see the best model for our data set is Support Vector Machine with a score of 0.974, was the highest scores for the data validation and also was the most balanced model which means it can predict better unseen data and has lowest possibility to have overfitting.

Summary

In order to get the best results we tried different models to compare these in the end. Throughout our work we noticed that it's really important a previous analysis from the data in order to know which graphs, features engineerings and methods fits better with the problem we faced different problems specially with the hyperparameter, deciding which parameters fitted better with our data set and problem, taking many time to run it but every time knowing more and more how does our data set worked.

Different models have different capabilities and limitations. Achieving a high accuracy often requires a combination of appropriate data, optimal models and sufficient resources for training. Also different variations of letters leads to a difficult correct prediction of the letter.

During the realization of this project we worked with differents models, parameters, metrics not only using them but learning deeper and also using many graphs and features engineering in order to have a better performance, this was good to keep learning how to treat the data and which models and graphs are the better depending on the type of problems and types of features.

In conclusion, our best performing model was Support Vector Machine, which got a total percentage of 97.4%. We are very happy with the percentages we reached and models we created.

References

UCI Machine Learning Repository. (2019). Uci.edu.
https://archive.ics.uci.edu/dataset/59/letter+recognition
https://datascience.stackexchange.com

https://aigeekprogrammer.com
https://kaggle.com/datasets

lettr: This attribute represents a capital letter and can take one of 26 values from A to Z.

x-box: This attribute represents the horizontal position of a box and is an integer value.

y-box: This attribute represents the vertical position of a box and is an integer value.

width: This attribute represents the width of the box and is an integer value.

high: This attribute represents the height of the box and is an integer value.

onpix: This attribute represents the total number of "on" (i.e., active) pixels and is an integer value.

x-bar: This attribute represents the mean x-coordinate of "on" pixels within the box and is an integer value.

y-bar: This attribute represents the mean y-coordinate of "on" pixels within the box and is an integer value.

x2bar: This attribute represents the mean x-variance and is an integer value.

y2bar: This attribute represents the mean y-variance and is an integer value.

xybar: This attribute represents the mean correlation between x and y and is an integer value.

x2ybr: This attribute represents the mean of x * x * y and is an integer value.

xy2br: This attribute represents the mean of x * y * y and is an integer value.

x-ege: This attribute represents the mean edge count from left to right and is an integer value.

xegvy: This attribute represents the correlation of x-ege with y and is an integer value.

y-ege: This attribute represents the mean edge count from bottom to top and is an integer value.

yegvx: This attribute represents the correlation of y-ege with x and is an integer value.