COMP 7703 Project

Diewen Yang
45951363

Contents

1. Data cleaning and Analysis

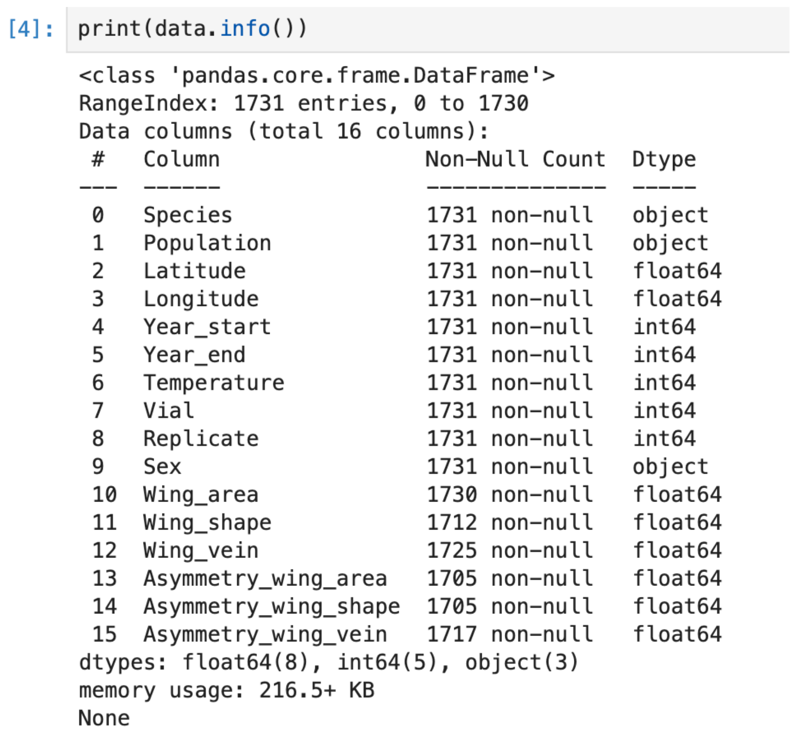
1.1 Dataset Description

The target of this project analysis is Document No. 84. In 1994, researchers collected biological information on five laboratory populations of two species of Drosophila, buzzatii and aldrichi. These populations were propagated in the laboratory at 25°C for five generations, and measurements were taken from the progeny of the fifth generation that were reared under three different temperature conditions (20°C, 25°C, and 30°C). This included wing area, wing shape, wing vein ratio, and fluctuating asymmetry (FA) in wing area, wing shape, and wing vein ratio for females and males, respectively.

There are 16 data feature columns in the file content, which can be roughly divided into four parts:

1) **Basic information**: species name, population name, geographical coordinates of sample collection, year of observation and gender
2) **Experimental conditions**: temperature during the experiment, Vial, and Replicate
3) **Wing morphological characteristics**: Wing area, Wing shape description, Characterization of veins in wings.
4) **Wing asymmetry measurements**: Asymmetry of wing area, Asymmetry measurement of wing shape, Asymmetry measurement of wing veins

Figure 1 shows the data type and number of features.



```
[4]: print(data.info())

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1731 entries, 0 to 1730
Data columns (total 16 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Species               1731 non-null   object 
 1   Population            1731 non-null   object 
 2   Latitude              1731 non-null   float64
 3   Longitude             1731 non-null   float64
 4   Year_start            1731 non-null   int64  
 5   Year_end              1731 non-null   int64  
 6   Temperature           1731 non-null   int64  
 7   Vial                  1731 non-null   int64  
 8   Replicate             1731 non-null   int64  
 9   Sex                   1731 non-null   object 
 10  Wing_area             1730 non-null   float64
 11  Wing_shape            1712 non-null   float64
 12  Wing_vein             1725 non-null   float64
 13  Asymmetry_wing_area   1705 non-null   float64
 14  Asymmetry_wing_shape  1705 non-null   float64
 15  Asymmetry_wing_vein   1717 non-null   float64
dtypes: float64(8), int64(5), object(3)
memory usage: 216.5+ KB
None
```

Figure 1. Data type and number of features

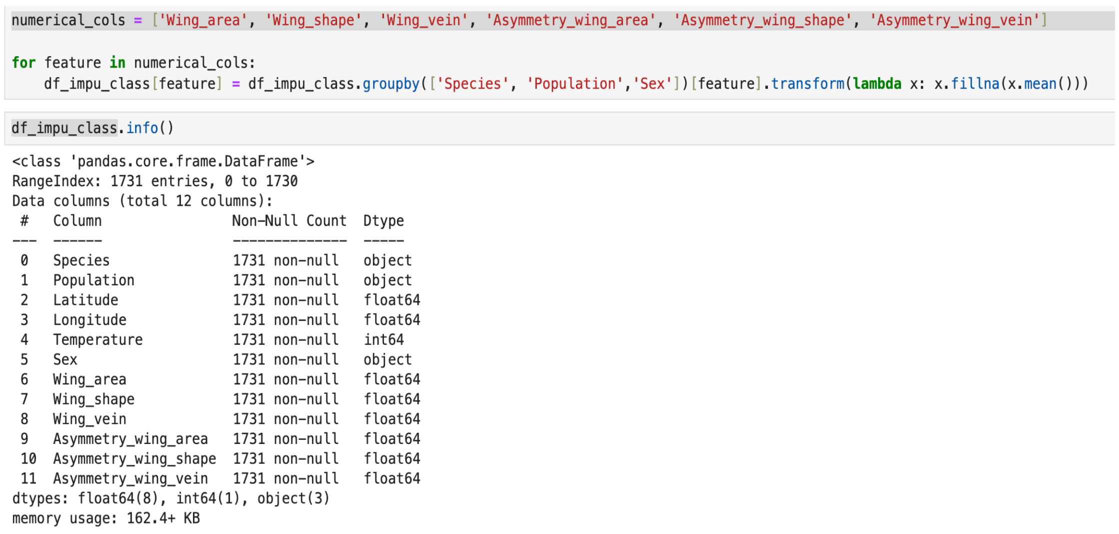
1.2 Data Preprocessing

In this dataset, we observe a total of 1731 instances, each containing 16 columns. However, some features have fewer than 1730 instances. Therefore, the dataset must have some missing values. In addition, the value ranges of features are also different. This means that we need to preprocess and standardize the data before analysing it for better performance.

1.2.1 Dealing missing values

There are generally two methods for dealing with missing values. One is to fill in missing data with the mean or median of all values, and the other is to impute based on specific category values. In this section, we replace missing values with the mean of the corresponding feature values in their respective categories. This approach can help maintain data integrity and improve model accuracy.

We use the **groupby** function to group the data by multiple categories ('Species', 'Population', 'Sex') and then apply the filled mean using the **transform** function.



```
numerical_cols = ['Wing_area', 'Wing_shape', 'Wing_vein', 'Asymmetry_wing_area', 'Asymmetry_wing_shape', 'Asymmetry_wing_vein']

for feature in numerical_cols:
    df_impu_class[feature] = df_impu_class.groupby(['Species', 'Population','Sex'])[feature].transform(lambda x: x.fillna(x.mean()))
```

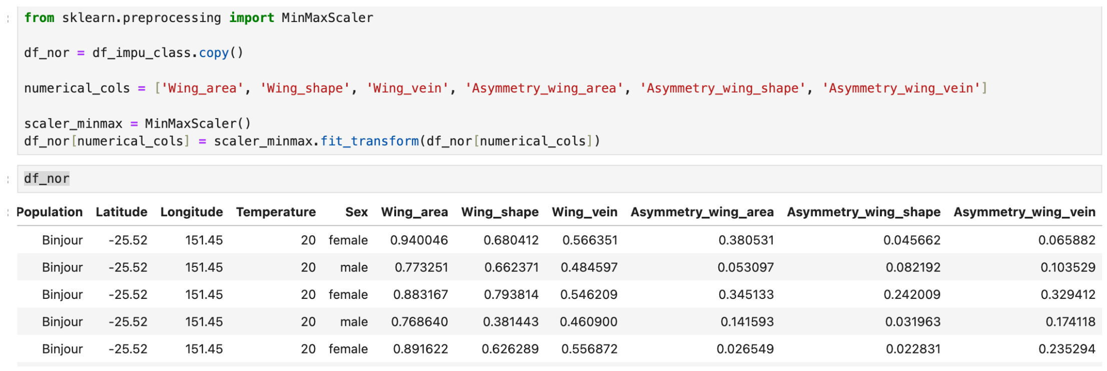
```
df_impu_class.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1731 entries, 0 to 1730
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype  
---  ------                --------------  -----  
 0   Species               1731 non-null   object 
 1   Population            1731 non-null   object 
 2   Latitude              1731 non-null   float64
 3   Longitude             1731 non-null   float64
 4   Temperature           1731 non-null   int64  
 5   Sex                   1731 non-null   object 
 6   Wing_area             1731 non-null   float64
 7   Wing_shape            1731 non-null   float64
 8   Wing_vein             1731 non-null   float64
 9   Asymmetry_wing_area   1731 non-null   float64
 10  Asymmetry_wing_shape  1731 non-null   float64
 11  Asymmetry_wing_vein   1731 non-null   float64
dtypes: float64(8), int64(1), object(3)
memory usage: 162.4+ KB
```

Figure 2. Missing data imputation

1.2.2 Normalization Data

we normalized the numerical features and scaled the features with different value ranges into the same range. This can improve the stability of model training and the accuracy of predictions.



```python
from sklearn.preprocessing import MinMaxScaler

df_nor = df_impu_class.copy()

numerical_cols = ['Wing_area', 'Wing_shape', 'Wing_vein', 'Asymmetry_wing_area', 'Asymmetry_wing_shape', 'Asymmetry_wing_vein']

scaler_minmax = MinMaxScaler()
df_nor[numerical_cols] = scaler_minmax.fit_transform(df_nor[numerical_cols])
```

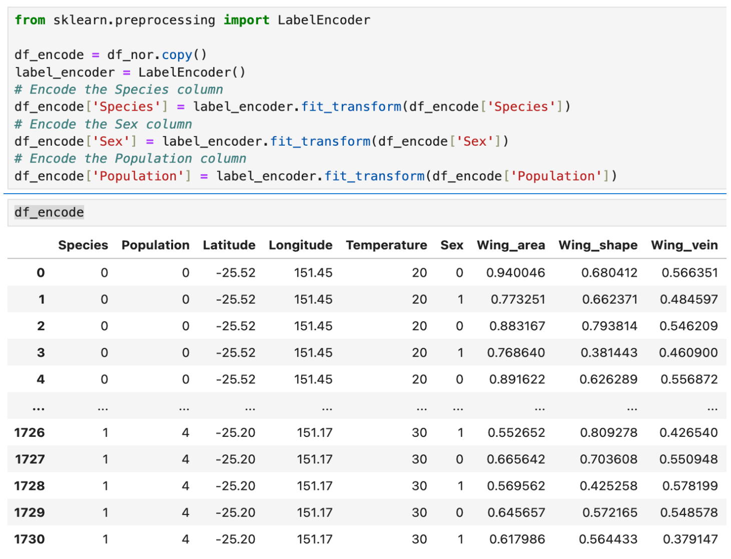
```python
df_nor
```

Population	Latitude	Longitude	Temperature	Sex	Wing_area	Wing_shape	Wing_vein	Asymmetry_wing_area	Asymmetry_wing_shape	Asymmetry_wing_vein
Binjour	-25.52	151.45	20	female	0.940046	0.680412	0.566351	0.380531	0.045662	0.065882
Binjour	-25.52	151.45	20	male	0.773251	0.662371	0.484597	0.053097	0.082192	0.103529
Binjour	-25.52	151.45	20	female	0.883167	0.793814	0.546209	0.345133	0.242009	0.329412
Binjour	-25.52	151.45	20	male	0.768640	0.381443	0.460900	0.141593	0.031963	0.174118
Binjour	-25.52	151.45	20	female	0.891622	0.626289	0.556872	0.026549	0.022831	0.235294

Figure 3. Normalization data

1.2.4 Encoding Data

In this part, we need to convert the classification ('Species', 'Population', 'Sex) features into a format that the model can recognise for subsequent data analysis. Three classifications are handled using Label Encoder.



```python
from sklearn.preprocessing import LabelEncoder

df_encode = df_nor.copy()
label_encoder = LabelEncoder()
# Encode the Species column
df_encode['Species'] = label_encoder.fit_transform(df_encode['Species'])
# Encode the Sex column
df_encode['Sex'] = label_encoder.fit_transform(df_encode['Sex'])
# Encode the Population column
df_encode['Population'] = label_encoder.fit_transform(df_encode['Population'])
```

```python
df_encode
```

	Species	Population	Latitude	Longitude	Temperature	Sex	Wing_area	Wing_shape	Wing_vein
0	0	0	-25.52	151.45	20	0	0.940046	0.680412	0.566351
1	0	0	-25.52	151.45	20	1	0.773251	0.662371	0.484597
2	0	0	-25.52	151.45	20	0	0.883167	0.793814	0.546209
3	0	0	-25.52	151.45	20	1	0.768640	0.381443	0.460900
4	0	0	-25.52	151.45	20	0	0.891622	0.626289	0.556872
...	...	...	...	...	...	...	...	...	...
1726	1	4	-25.20	151.17	30	1	0.552652	0.809278	0.426540
1727	1	4	-25.20	151.17	30	0	0.665642	0.703608	0.550948
1728	1	4	-25.20	151.17	30	1	0.569562	0.425258	0.578199
1729	1	4	-25.20	151.17	30	0	0.645657	0.572165	0.548578
1730	1	4	-25.20	151.17	30	1	0.617986	0.564433	0.379147

Figure 4. Encoding data

2. Exploratory Data Analysis (EDA)

Histograms and heatmaps were used in exploratory data analysis to display the distribution of numerical features and their relationships.

2.1 Histogram of Numerical Data

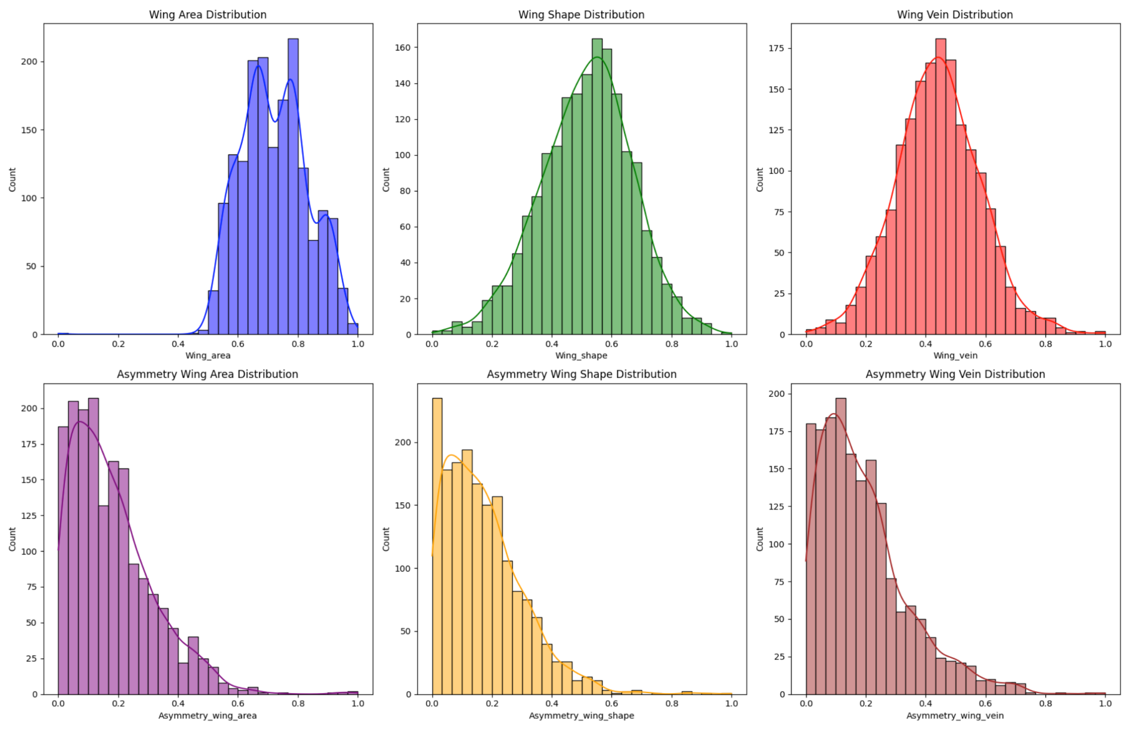


Figure 5. histogram of numerical data

As can be seen from Figure 5, the **Wing Area, Wing Shape, and Wing Vein** histograms indicate that they are roughly normally distributed, but with some slight skew. The histograms of asymmetry features (**Asymmetry in Wing Area, Asymmetry in Wing Shape and Asymmetry in Wing Vein**) show that the data are mainly concentrated in lower values, but there are also some large values present, indicating that some samples have higher asymmetry.

2.2 Heatmap of Data

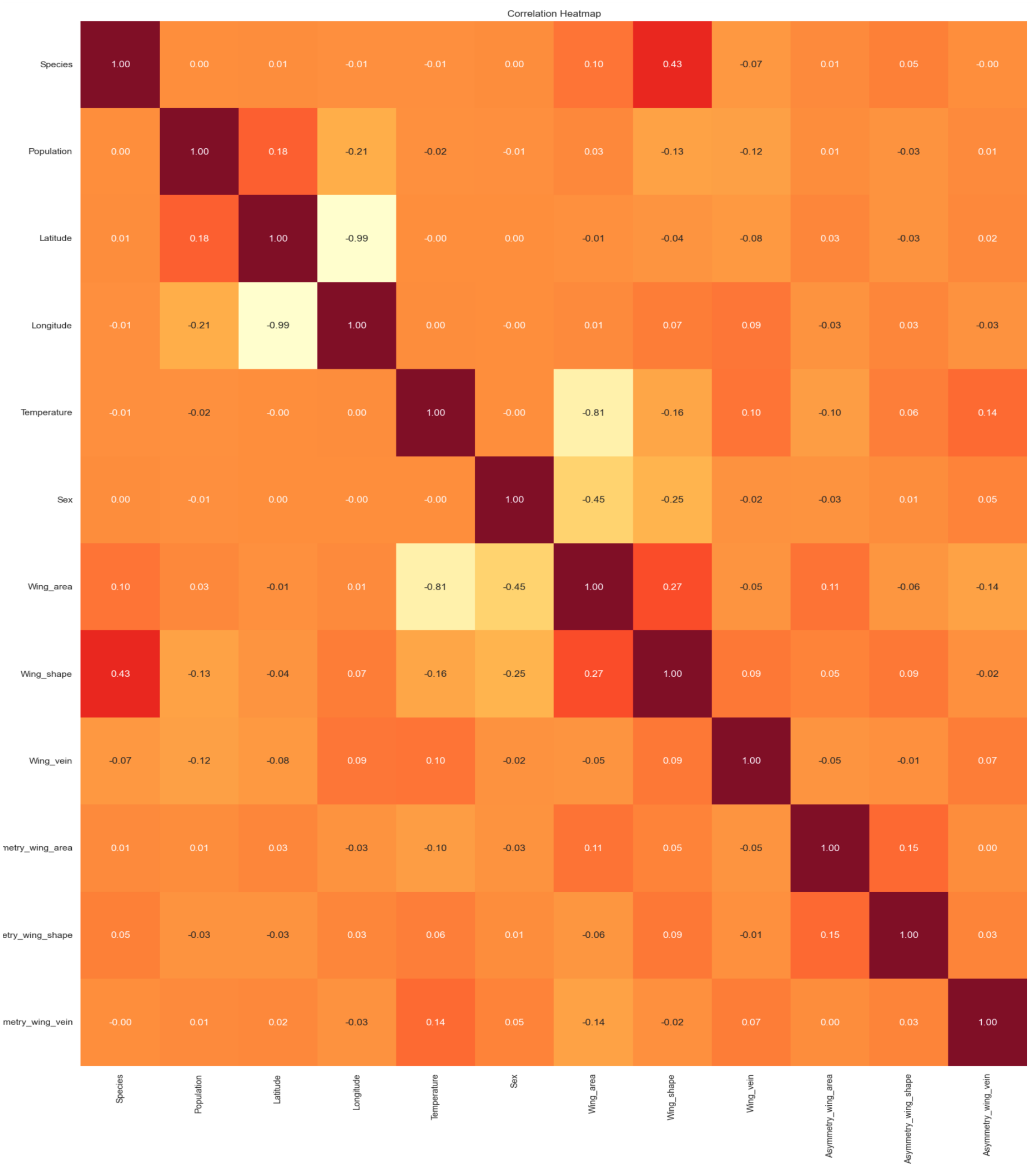


Figure 6. Heatmap of data

The correlation coefficient between **Temperature** and **wing area** is -0.81. This shows that temperature has a significant effect on wing size.

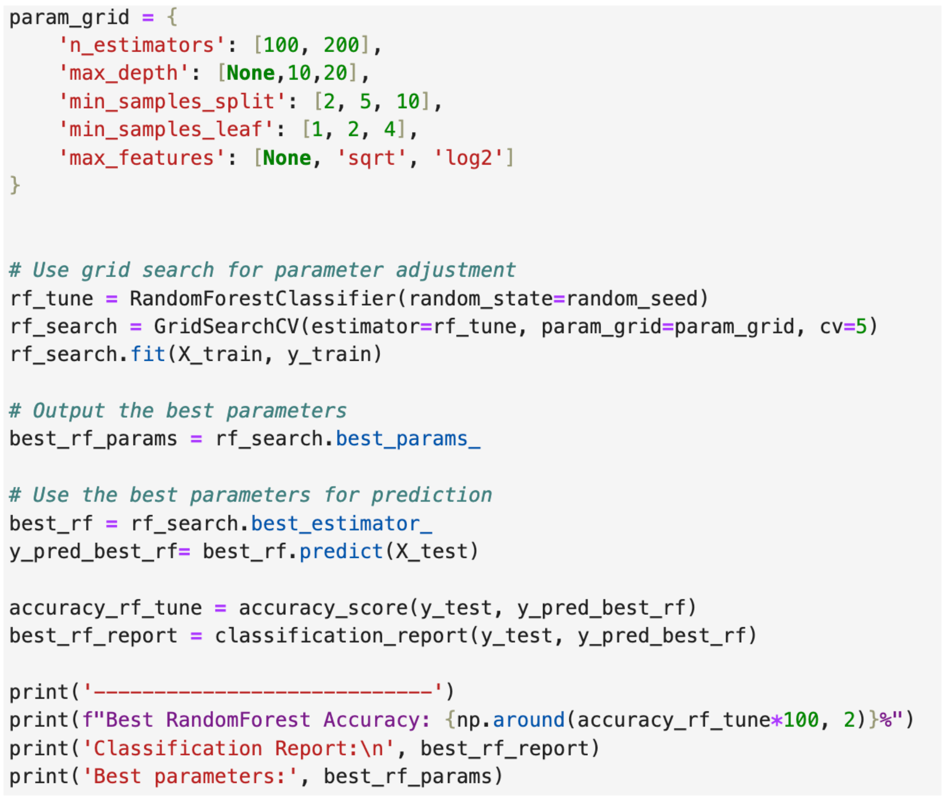
3. Models

We will predict Drosophila **species** using three different models and finally compare them to select the optimal model.

3.1 Random Forest Classifier

3.1.1 Training

We used GridSearch to tune the model's hyperparameters. The following parameters are particularly important when tuning the parameters of a random forest model. The common number of **n_estimators** is usually between 100 and 200 but can be increased to 500 or 1000 or even more depending on the specific problem and computational resources. Although model performance is improved by increasing the number of trees (n_estimators), it also increases computation time. For the dataset of this project, we choose the common 100 and 200 to reduce the running time. **max_depth**, the minimum number of samples required for node splitting (**min_samples_split**) and the minimum number of samples required for leaf nodes (**min_samples_leaf**) are three parameters that are adjusted to prevent model overfitting. **max_features** introduce diversity by limiting the number of features considered per split, common settings include None (all features), square root of the number of features (sqrt) and logarithm of the number of features (log2).

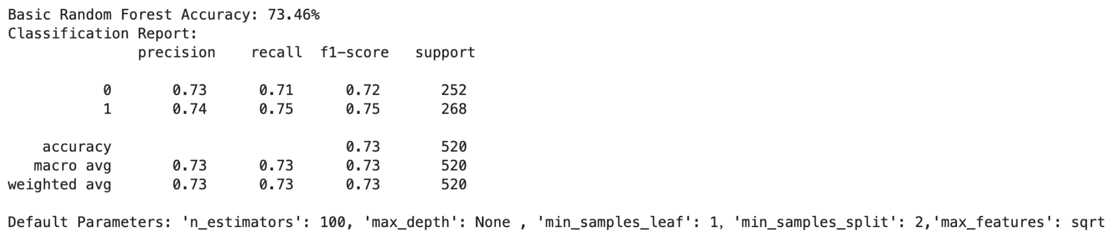


```
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None,10,20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}


# Use grid search for parameter adjustment
rf_tune = RandomForestClassifier(random_state=random_seed)
rf_search = GridSearchCV(estimator=rf_tune, param_grid=param_grid, cv=5)
rf_search.fit(X_train, y_train)

# Output the best parameters
best_rf_params = rf_search.best_params_

# Use the best parameters for prediction
best_rf = rf_search.best_estimator_
y_pred_best_rf= best_rf.predict(X_test)

accuracy_rf_tune = accuracy_score(y_test, y_pred_best_rf)
best_rf_report = classification_report(y_test, y_pred_best_rf)

print('-----------------------------')
print(f"Best RandomForest Accuracy: {np.around(accuracy_rf_tune*100, 2)}%")
print('Classification Report:\n', best_rf_report)
print('Best parameters:', best_rf_params)
```

Figure 7. Original best Random Forest model code

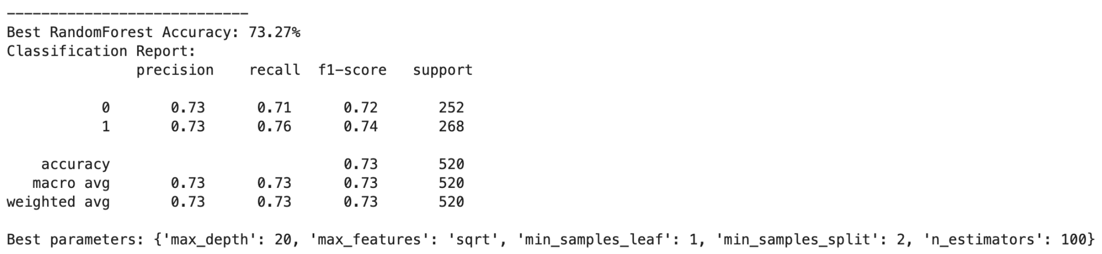
3.1.3 Problems & Analysis

From the following two figures, it can be observed that the optimized model exhibits lower accuracy compared to the default model. This suggests potential errors in the training process. After several attempts and analysis, I identified that the problem might be in the dataset-splitting method and the fold number of cross-validation. Specifically, when utilizing **train_test_split** to partition the dataset, the **stratify** parameter was not applied. Consequently, the dataset was divided randomly, potentially leading to great disparities in the distribution of data points across various categories within the training and test sets. This discrepancy adversely impacted the training and evaluation efficacy of the subsequent models. In addition, the number of folds used in cross-validation might have been insufficient, leading to unstable evaluation results. Therefore, subsequent analyses employed **StratifiedKFold**, ensuring that the class distribution within each fold is consistent with the overall dataset distribution. Figures 8 and 9 present the original models' data.



```
Basic Random Forest Accuracy: 73.46%
Classification Report:
              precision    recall  f1-score   support

           0       0.73      0.71      0.72       252
           1       0.74      0.75      0.75       268

    accuracy                           0.73       520
   macro avg       0.73      0.73      0.73       520
weighted avg       0.73      0.73      0.73       520

Default Parameters: 'n_estimators': 100, 'max_depth': None , 'min_samples_leaf': 1, 'min_samples_split': 2,'max_features': sqrt
```

Figure8. Original default Random Forest model metrics

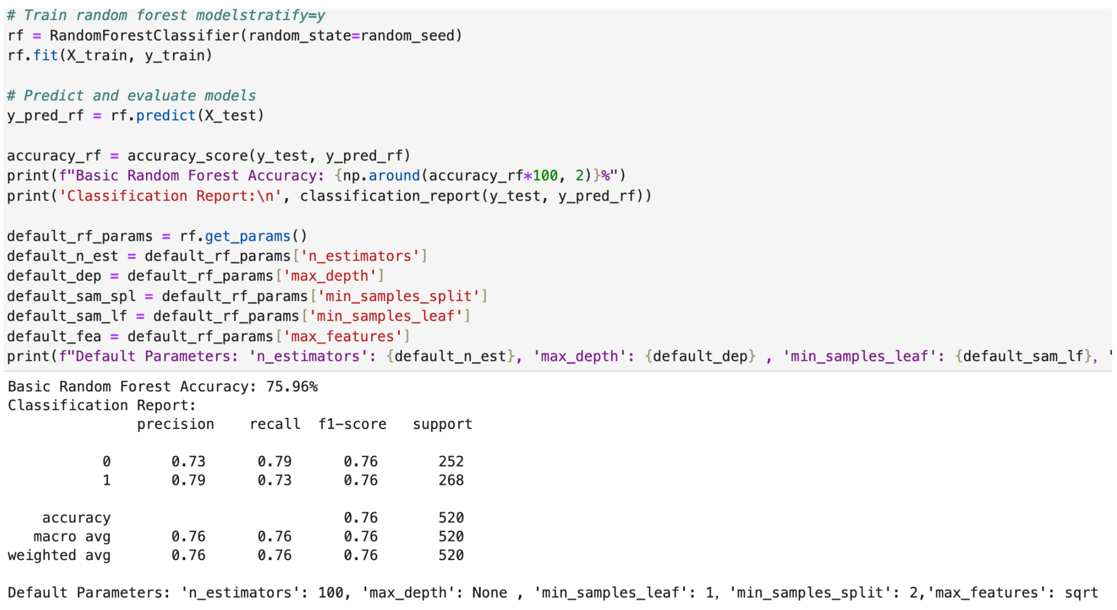


```
----------------------------
Best RandomForest Accuracy: 73.27%
Classification Report:
              precision    recall  f1-score   support

           0       0.73      0.71      0.72       252
           1       0.73      0.76      0.74       268

    accuracy                           0.73       520
   macro avg       0.73      0.73      0.73       520
weighted avg       0.73      0.73      0.73       520

Best parameters: {'max_depth': 20, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

Figure9. Original best Random Forest model metrics

3.1.4 Final result

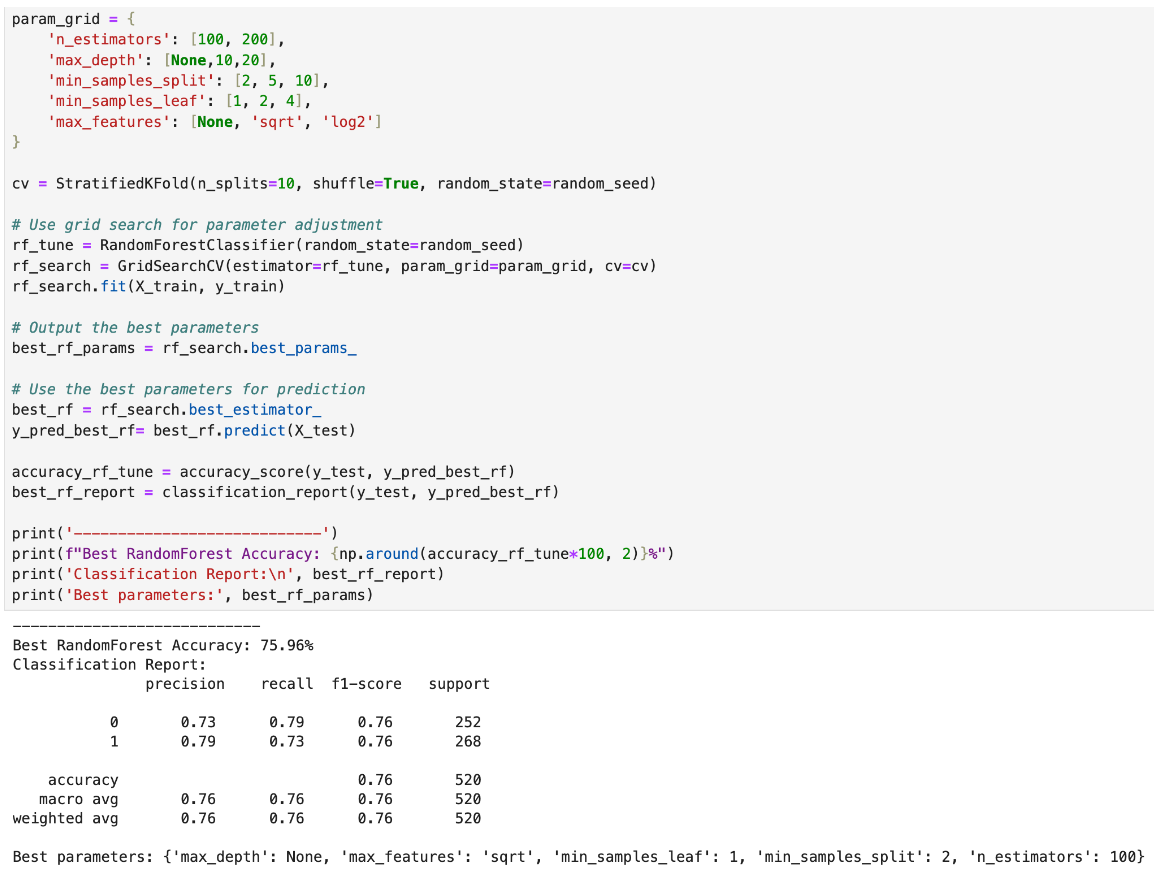
After changing the data splitting method, the accuracy of both models improved (75.96%). This confirms that the initial hypothesis was logical. The optimized model returned the default Random Forest parameters, indicating that the default model can be used directly to reduce computational costs.



```python
# Train random forest modelstratify=y
rf = RandomForestClassifier(random_state=random_seed)
rf.fit(X_train, y_train)

# Predict and evaluate models
y_pred_rf = rf.predict(X_test)

accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Basic Random Forest Accuracy: {np.around(accuracy_rf*100, 2)}%")
print('Classification Report:\n', classification_report(y_test, y_pred_rf))

default_rf_params = rf.get_params()
default_n_est = default_rf_params['n_estimators']
default_dep = default_rf_params['max_depth']
default_sam_spl = default_rf_params['min_samples_split']
default_sam_lf = default_rf_params['min_samples_leaf']
default_fea = default_rf_params['max_features']
print(f"Default Parameters: 'n_estimators': {default_n_est}, 'max_depth': {default_dep} , 'min_samples_leaf': {default_sam_lf}, '
```

```
Basic Random Forest Accuracy: 75.96%
Classification Report:
               precision    recall  f1-score   support

           0       0.73      0.79      0.76       252
           1       0.79      0.73      0.76       268

    accuracy                           0.76       520
   macro avg       0.76      0.76      0.76       520
weighted avg       0.76      0.76      0.76       520

Default Parameters: 'n_estimators': 100, 'max_depth': None , 'min_samples_leaf': 1, 'min_samples_split': 2,'max_features': sqrt
```

Figure10. Final default Random Forest model metrics



```python
param_grid = {
    'n_estimators': [100, 200],
    'max_depth': [None,10,20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'max_features': [None, 'sqrt', 'log2']
}

cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)

# Use grid search for parameter adjustment
rf_tune = RandomForestClassifier(random_state=random_seed)
rf_search = GridSearchCV(estimator=rf_tune, param_grid=param_grid, cv=cv)
rf_search.fit(X_train, y_train)

# Output the best parameters
best_rf_params = rf_search.best_params_

# Use the best parameters for prediction
best_rf = rf_search.best_estimator_
y_pred_best_rf= best_rf.predict(X_test)

accuracy_rf_tune = accuracy_score(y_test, y_pred_best_rf)
best_rf_report = classification_report(y_test, y_pred_best_rf)

print('----------------------------')
print(f"Best RandomForest Accuracy: {np.around(accuracy_rf_tune*100, 2)}%")
print('Classification Report:\n', best_rf_report)
print('Best parameters:', best_rf_params)
```

```
----------------------------
Best RandomForest Accuracy: 75.96%
Classification Report:
               precision    recall  f1-score   support

           0       0.73      0.79      0.76       252
           1       0.79      0.73      0.76       268

    accuracy                           0.76       520
   macro avg       0.76      0.76      0.76       520
weighted avg       0.76      0.76      0.76       520

Best parameters: {'max_depth': None, 'max_features': 'sqrt', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100}
```

Figure11. Final best Random Forest model metrics

3.2 Logistic Regression

3.2.1 Training

In the logistic regression model, the regularization strength *C* and the regularization type penalty are two important parameters used to control the complexity of the model and prevent overfitting. Regularization strength ***C*** value ranges are: 0.01, 0.1, 1, 10, 100. Regularization type penalty: **L1 (Lasso)** and **L2 (Ridge)**. Commonly used solvers for logistic regression models include **lbfgs** and **saga**. However, **lbfgs** only supports L2 regularization, hence we categorize the setting for param_grid.

```
param_grid = [
    {'solver': ['lbfgs'], 'penalty': ['l2'], 'C': [0.01, 0.1, 1, 10, 100]},
    {'solver': ['saga'], 'penalty': ['l1', 'l2'], 'C': [0.01, 0.1, 1, 10, 100]}
]

# Use grid search for parameter adjustment
log_reg_tune = LogisticRegression(max_iter=10000,random_state=random_seed)
log_grid_search = GridSearchCV(estimator=log_reg_tune, param_grid=param_grid, cv=5, scoring='accuracy')
log_grid_search.fit(X_train, y_train)

# Output the best parameters
best_logi_params = log_grid_search.best_params_

# Use the best parameters for prediction
best_log_reg = log_grid_search.best_estimator_
y_pred_best_log_reg = best_log_reg.predict(X_test)

accuracy_logi_tune = accuracy_score(y_test, y_pred_best_log_reg)
best_logi_report = classification_report(y_test, y_pred_best_log_reg)

print('----------------------------')
print(f"Best Logistic Accuracy: {np.around(accuracy_logi_tune*100, 2)}%")
print('Classification Report:\n', best_logi_report)
print('Best parameters:', best_logi_params)
```

Figure 7. Best logistic regression model code

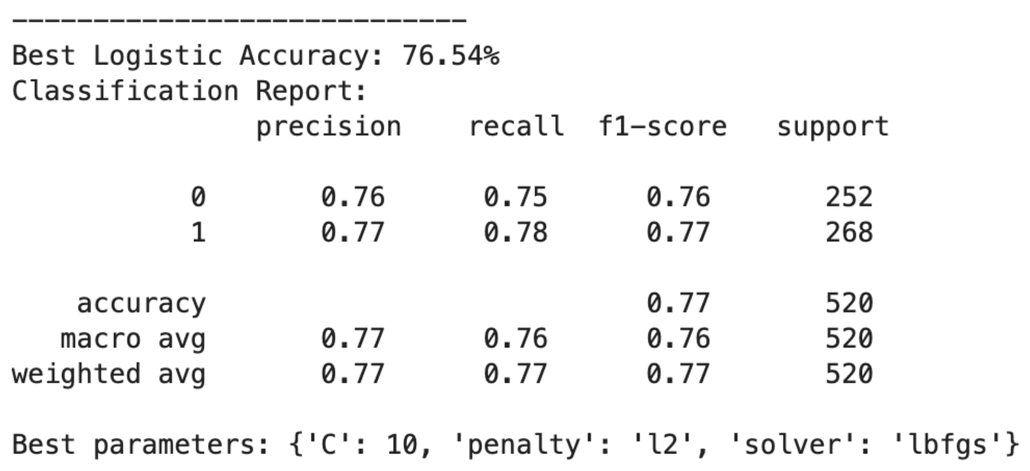
3.2.2 Result & Analysis

The accuracy of the basic logistic regression model is 74.42%.

```
----------------------------
Basic Logistic Accuracy: 74.42%
Classification Report:
               precision    recall  f1-score   support

           0       0.75      0.71      0.73       252
           1       0.74      0.77      0.76       268

    accuracy                           0.74       520
   macro avg       0.74      0.74      0.74       520
weighted avg       0.74      0.74      0.74       520

Default Parameters: C: 1.0 , penalty: l2 , solver: lbfgs
```

Figure 8. Basic logistic regression model metrics

The results (Fig. 9) show that 76.54% model accuracy is achieved with a regularization strength *C* of 100, a regularization type penalty of L2 (Ridge), and a solver that uses lbfgs. The optimized parameter *C* represents weaker regularization. This means that the model allows for higher coefficient values, thus enabling it to fit the training data better. The base model defaults to a value of 1 for *C*, and greater regularization strength limits the ability to fit the model. L2 regularization reduces the complexity of the model by using the sum of squares of the penalty coefficients thus preserving the information of all the features without directly turning some of the coefficients to zero as in L1. In addition to this, the lbfgs solver can efficiently handle multi-class classification problems and performs excellently when dealing with large-scale data. Therefore, the optimized regression model is more accurate than the basic model, and the adjusted model can prevent model overfitting to some extent.



```
----------------------------
Best Logistic Accuracy: 76.54%
Classification Report:
              precision    recall  f1-score   support

           0       0.76      0.75      0.76       252
           1       0.77      0.78      0.77       268

    accuracy                           0.77       520
   macro avg       0.77      0.76      0.76       520
weighted avg       0.77      0.77      0.77       520

Best parameters: {'C': 10, 'penalty': 'l2', 'solver': 'lbfgs'}
```

Figure 9. Best logistic regression model metrics

3.3 K-Nearest Neighbors Classifier

3.3.1 Training

The performance of the K-Nearest Neighbours (KNN) classifier is significantly influenced by various parameters, among which the number of nearest neighbours (**n_neighbors**) is particularly critical. Adjusting this parameter can have a significant impact on the accuracy of the model. Smaller n_neighbors values tend to increase the risk of overfitting, whereas larger values may lead to underfitting. To optimize model performance, we conduct a parameter search by selecting k values ranging from 1 to 100.

3.3.2 Result & Analysis

However, as illustrated in Figures 10 and 11, increasing the number of neighbours results in a decline in the accuracy of the KNN classifier, while the loss increases. This indicates that the model's performance deteriorates with a higher number of neighbours. When the K value is larger, the KNN classifier is more likely to include noise points and irrelevant points in its decision-making process. KNN primarily relies on local nearest neighbours' information for classification. A higher k value enables the classifier to make decisions over a broader range, but it weakens the influence of the features of adjacent points. This can result in the misclassification of points that should belong to a particular category.
Figure 12 shows that our accuracy is highest when we have 5 neighbours, with an accuracy of 72.69%.

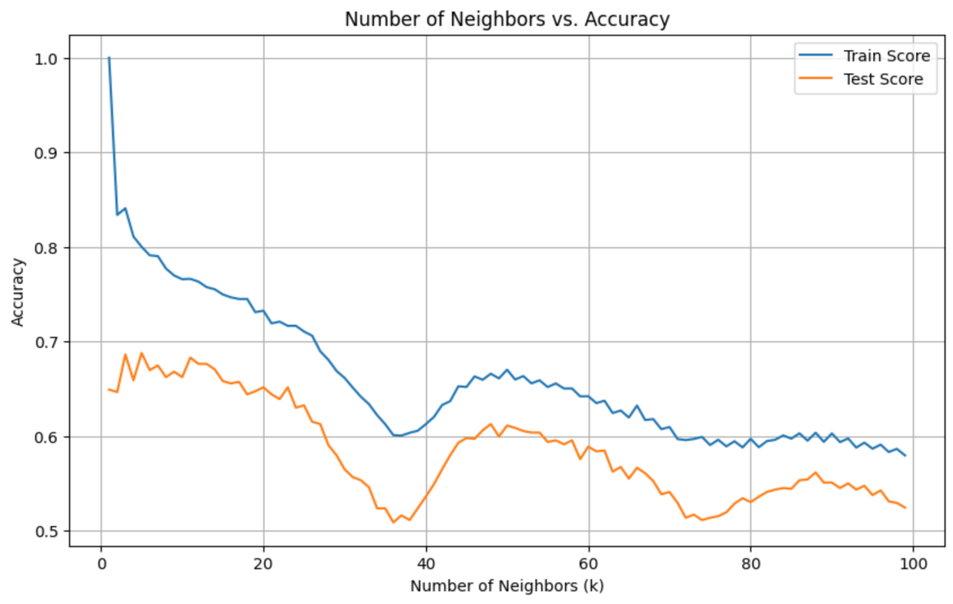


Figure 10. Number of Neighbours vs. Accuracy

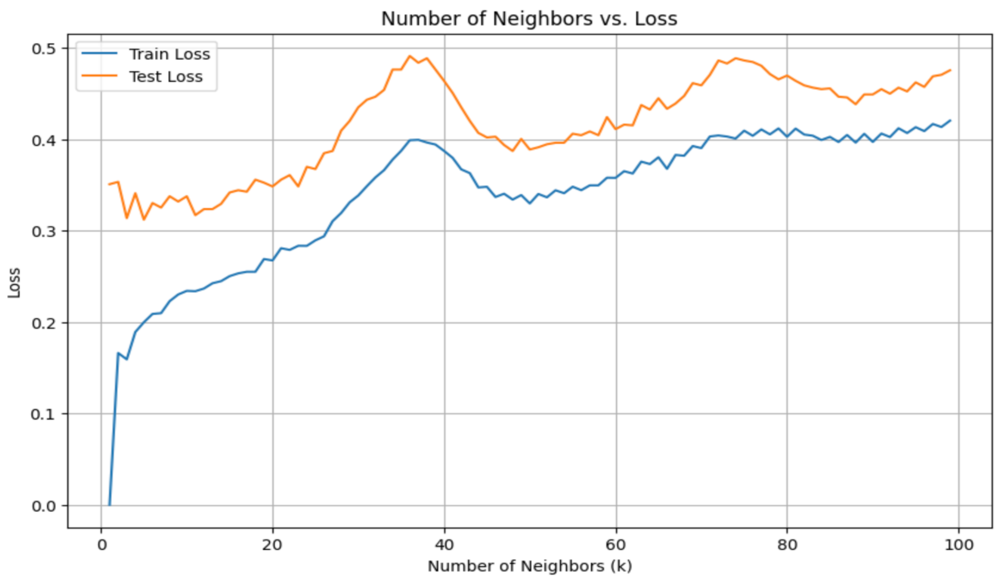
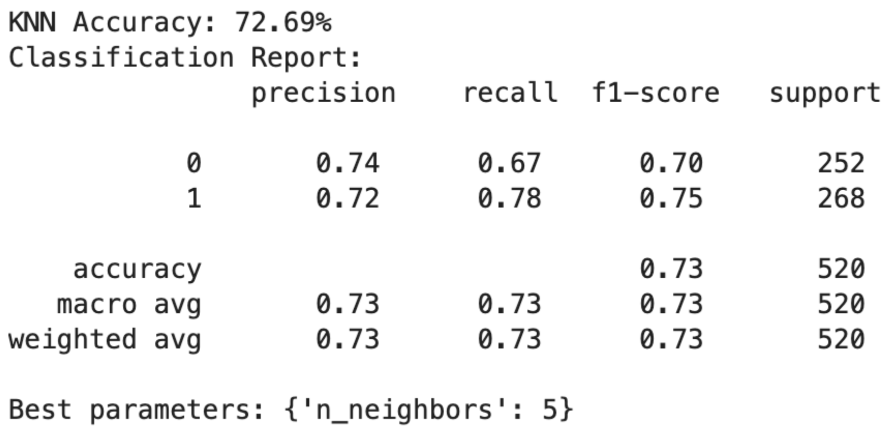


Figure 11. Number of Neighbours vs. loss



```
KNN Accuracy: 72.69%
Classification Report:
              precision    recall  f1-score   support

           0       0.74      0.67      0.70       252
           1       0.72      0.78      0.75       268

    accuracy                           0.73       520
   macro avg       0.73      0.73      0.73       520
weighted avg       0.73      0.73      0.73       520

Best parameters: {'n_neighbors': 5}
```

Figure 12. KNN model metrics

4. Conclusion

In this project, I analysed and modelled experimental data from two species of Drosophila to investigate the impact of different experimental conditions on wing morphological characteristics and their asymmetries for species classification. I conducted data cleaning, preprocessing, and feature encoding to ensure data integrity and consistency. Histograms and heatmaps were used in exploratory data analysis to display the distribution of numerical features and their relationships.

For species prediction, I used Random Forest, Logistic Regression, and K-Nearest Neighbours classifiers, optimizing each model's performance through hyperparameter tuning. Notably, the method of data splitting significantly influenced the performance of the Random Forest model. Initially, without stratified sampling, class distributions were inconsistent between training and test sets, affecting model training and evaluation. Introducing StratifiedKFold improved accuracy by ensuring consistent class distribution across folds.

The accuracy differences among the three models are minor, considering the data and practical implementation, I think the Random Forest model is the most suitable choice. It handles non-linear relationships well, and despite its higher computational cost, it performs excellently with sufficient resources.

The optimized Logistic Regression model achieved the highest accuracy of 76.54%, with high computational efficiency and simplicity in implementation, but it is less effective in handling non-linear relationships compared to Random Forest. The optimized Random Forest model achieved an accuracy of 75.96% and is suitable when computational resources are abundant. The KNN classifier's highest accuracy was 72.69% with n_neighbors set to 5. It is suitable for simple classification problems but is less efficient for large-scale data.

In conclusion, selecting the appropriate model based on specific problems and dataset characteristics is crucial. In this project, although the Logistic Regression model slightly outperformed in accuracy, the Random Forest model is more suitable as the final classification model due to its superior handling of non-linear relationships and overall performance.