Data Exploration and Preparation

Introduction to Data Analytics (31250)

Assessment Task 3
Data Mining in Action

Aiden Abignano
13911211

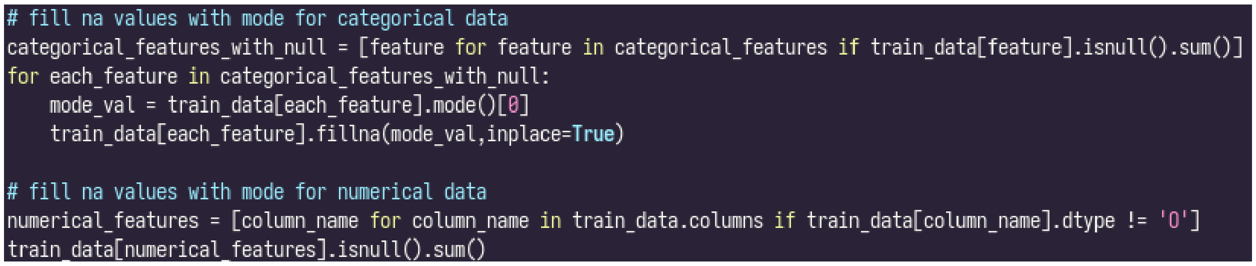
1. Data Mining Problem

This task requires a classifier capable of predicting RainTomorrow from an arrangement of twenty one data points recorded from the previous day with high precision. In order to successfully fulfill this problem, experiments must be conducted on a range of classifier techniques in order to identify the optimal algorithm including random forest, k-nearest neighbour, linear regression and SVM. The data being analysed includes integers, doubles, categories and boolean values which will need to be preprocessed and transformed in a way that utilises the strengths of each technique to compare them accurately. Comparisons will be made using an accuracy score, confusion matrix, Cohen's kappa coefficient, f-score and recall calculated from the provided weather data. Once the optimal model has been determined, it will be used on the unknown data and saved in a comma-separated value format.

2. Data Preprocessing and Transformations Performed

A series of data preprocessing and transformations were performed on the weather data before each classifier technique was executed. Each classifier investigated used a different combination of methods that aimed to employ its strengths.

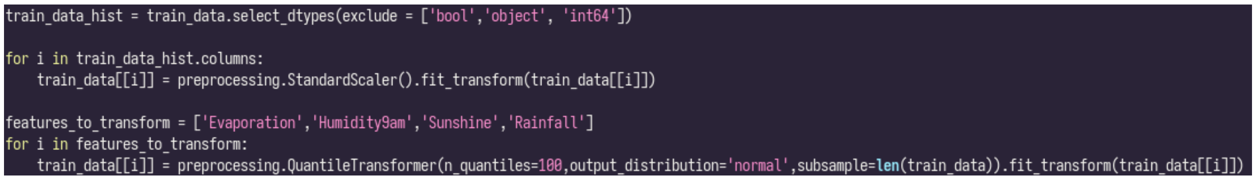
Some classifier techniques such as k-nearest neighbour and linear regression refused to evaluate rows with missing information of which there were many. On this account, it was critical to fill in the missing information in the least biased way to reduce distortion as much as possible. Two main approaches were tested, filling in the blanks with zero values or filling them with the most popular values (mode). Trials revealed using the mode had a much more positive impact on overall accuracy of the program with a difference in f-score of up to 10%.



```python
# fill na values with mode for categorical data
categorical_features_with_null = [feature for feature in categorical_features if train_data[feature].isnull().sum()]
for each_feature in categorical_features_with_null:
    mode_val = train_data[each_feature].mode()[0]
    train_data[each_feature].fillna(mode_val,inplace=True)

# fill na values with mode for numerical data
numerical_features = [column_name for column_name in train_data.columns if train_data[column_name].dtype != 'O']
train_data[numerical_features].isnull().sum()
```

[1] Python code used to fill in missing categorical and numerical data with mode values

The linear regression model could also only be trained on data that had been normalised between a value of 0 and 1 which meant an effective way to accurately map values between 0 and 1 without distorting the data was important. This was achieved using quantile transformation presented in figure [2]. Quantile transformation was used as it translates the features to follow a uniform or a normal distribution which tends to spread out the most frequent values reducing the impact of (marginal) outliers.

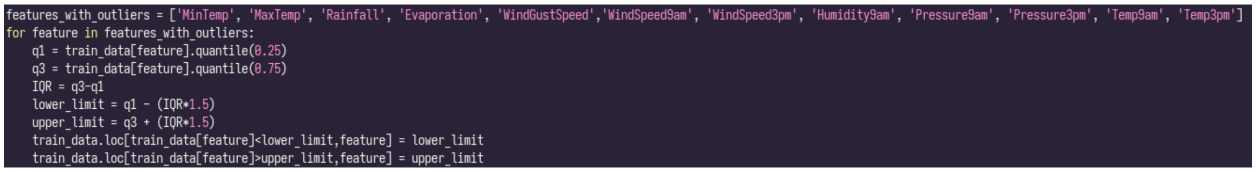


```python
train_data_hist = train_data.select_dtypes(exclude = ['bool','object', 'int64'])

for i in train_data_hist.columns:
    train_data[[i]] = preprocessing.StandardScaler().fit_transform(train_data[[i]])

features_to_transform = ['Evaporation','Humidity9am','Sunshine','Rainfall']
for i in features_to_transform:
    train_data[[i]] = preprocessing.QuantileTransformer(n_quantiles=100,output_distribution='normal',subsample=len(train_data)).fit_transform(train_data[[i]])
```

[2] Python code using Scikit-learn’s quantile transformation function.

Note that this transformation is non-linear and can distort linear correlations between variables measured at the same scale.

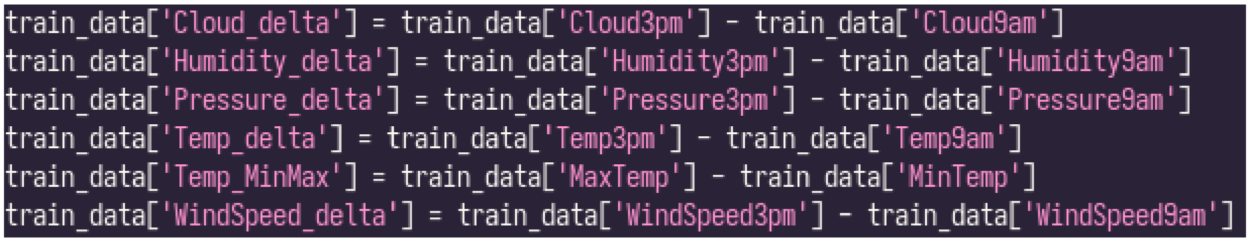
Other transformations were used that attempted to reduce the impact of outliers such as the code demonstrates in figure [3] which would alter any data points below q1-(IQR*1.5) or above q3+(IQR*1.5) into their respective lower and upper limits. However, further testing revealed this led to an increase in overfitting and had a negative effect on the accuracy and f-score of the classifier.



```python
features_with_outliers = ['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'WindGustSpeed','WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Pressure9am', 'Pressure3pm', 'Temp9am', 'Temp3pm']
for feature in features_with_outliers:
    q1 = train_data[feature].quantile(0.25)
    q3 = train_data[feature].quantile(0.75)
    IQR = q3-q1
    lower_limit = q1 - (IQR*1.5)
    upper_limit = q3 + (IQR*1.5)
    train_data.loc[train_data[feature]<lower_limit,feature] = lower_limit
    train_data.loc[train_data[feature]>upper_limit,feature] = upper_limit
```

[3] Python code that aimed to remove outliers

While most classifier techniques struggled with the high dimensionality of the dataset, the random forest and gradient boosting classifiers appeared to filter the higher correlated data more precisely, it was less effective forming more complex statistical relationships. For this reason, more columns were added to the database using information derived from the dataset such as the delta between the 3pm and 9am measurements including temperature, humidity, windspeed, cloud and pressure along with the variation between the maximum and minimum temperature. Although these observations can be deduced from the database directly, random forest and gradient boosting is much more effective with more direct relationships with minimal drawbacks if no correlation is measured. For example, the increase in cloud coverage over the day may have a stronger indication of rain tomorrow than cloud coverage measured at any single point in time. Extensive testing revealed a reliable increase in f-score from 0.63 to 0.67 using the added columns.



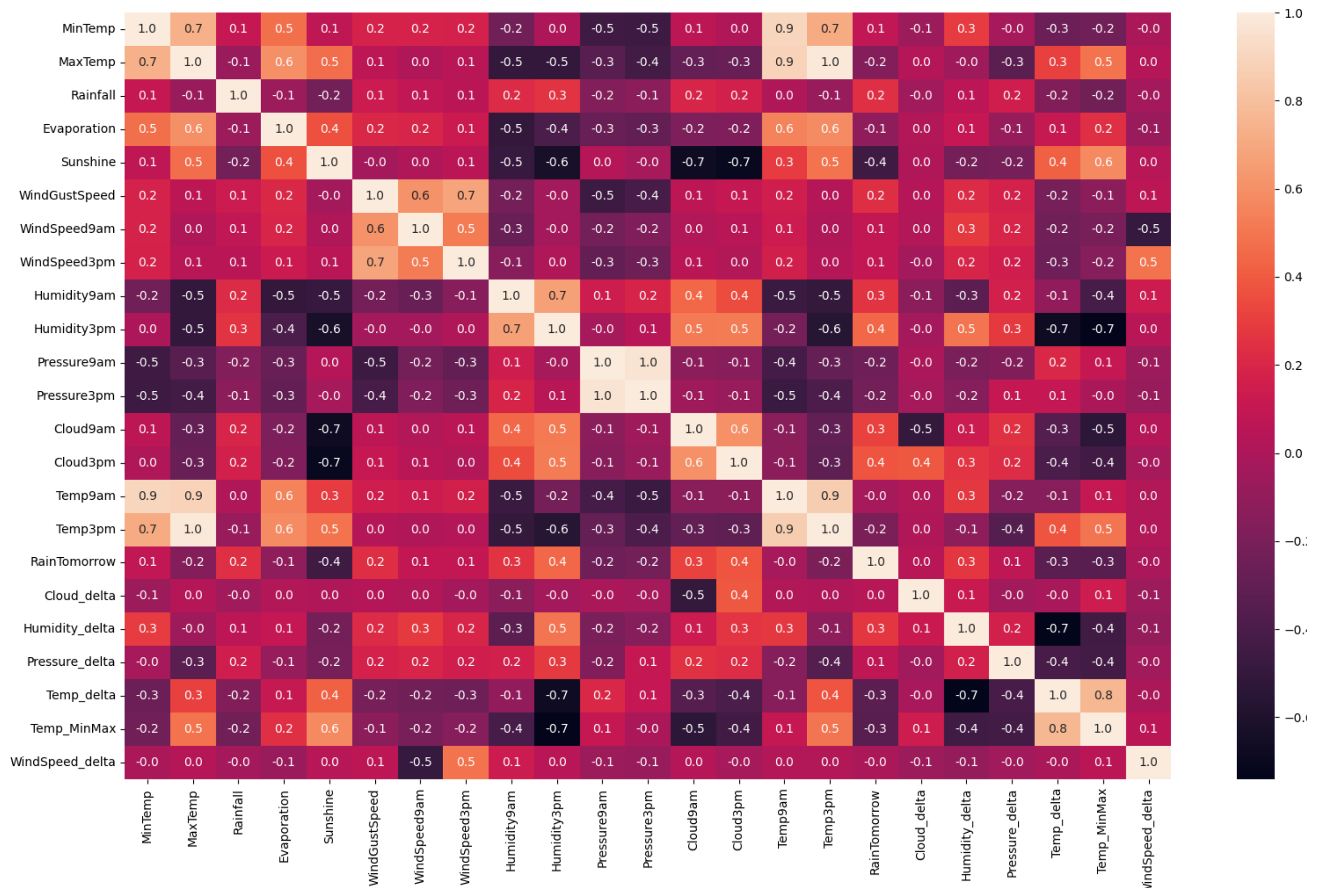
```python
train_data['Cloud_delta'] = train_data['Cloud3pm'] - train_data['Cloud9am']
train_data['Humidity_delta'] = train_data['Humidity3pm'] - train_data['Humidity9am']
train_data['Pressure_delta'] = train_data['Pressure3pm'] - train_data['Pressure9am']
train_data['Temp_delta'] = train_data['Temp3pm'] - train_data['Temp9am']
train_data['Temp_MinMax'] = train_data['MaxTemp'] - train_data['MinTemp']
train_data['WindSpeed_delta'] = train_data['WindSpeed3pm'] - train_data['WindSpeed9am']
```

[4] Added columns used in random forest and gradient boosting classifiers

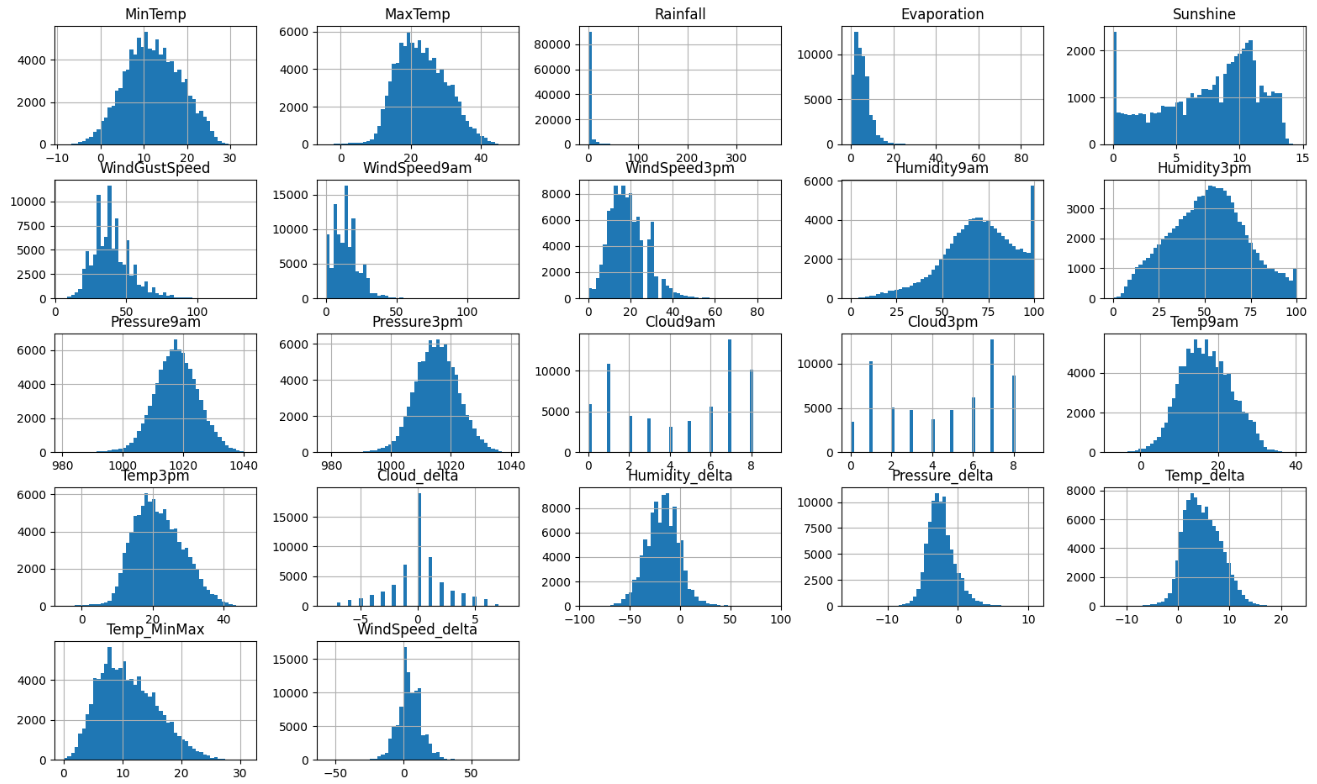
3. Solving the Problem

With the intention of solving the problem with the best classifier possible, several classifier techniques were examined using *KNIME* measuring their accuracy, Cohen's kappa and confusion matrix with the provided weather data using a 80/20 training/testing split for validation. Moreover, extensive research was used on the data such as the correlation map and histograms drawn in figure [5] and [6].

This allowed for a deeper understanding of the connections between different data points that were recorded instigating more effective preprocessing and data transformations.



[5] Correlation map between all data points in weather data plus added delta data



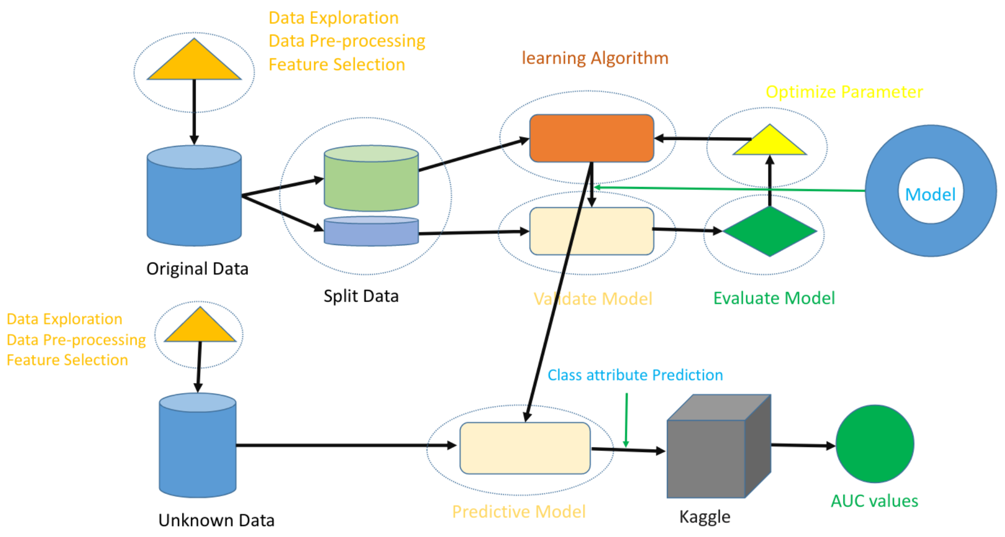
[6] Histograms of all the numerical and categorical data in weather data plus delta data

Each classifier also had hyper-parameter tuning performed; however, due to limited computing resources, random search was used to sample the tuning space rather than brute force running all combinations.

After all classifier techniques had been tested, they were compared by their confusion matrix and Cohen’s kappa, and the most promising one was in the Python programming language using Scikit-Learn and Pandas libraries adding finer control to the classifier and more ways to measure its performance such as recall and f-score.

4. Classification Techniques Used

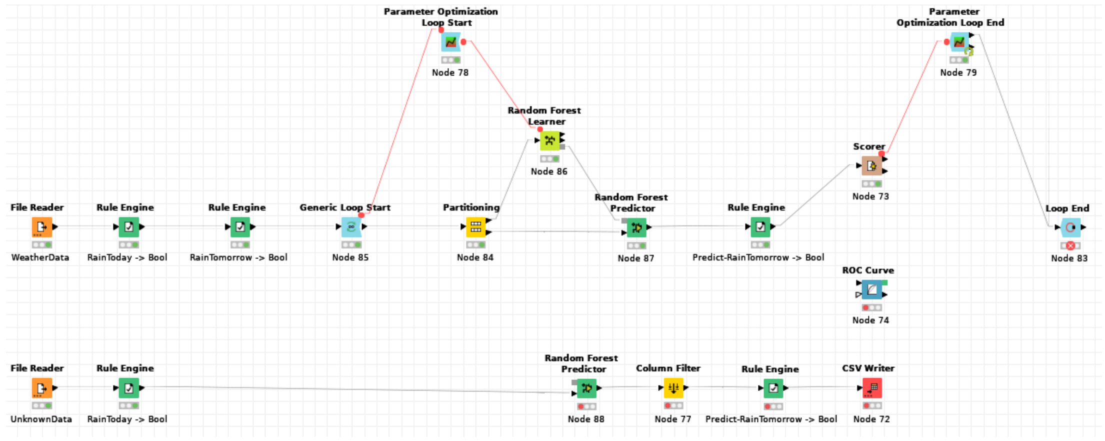
Each classifier technique was first executed and experimented with in KNIME based from this flow diagram, and later expanded upon using the Python programming language and related libraries.



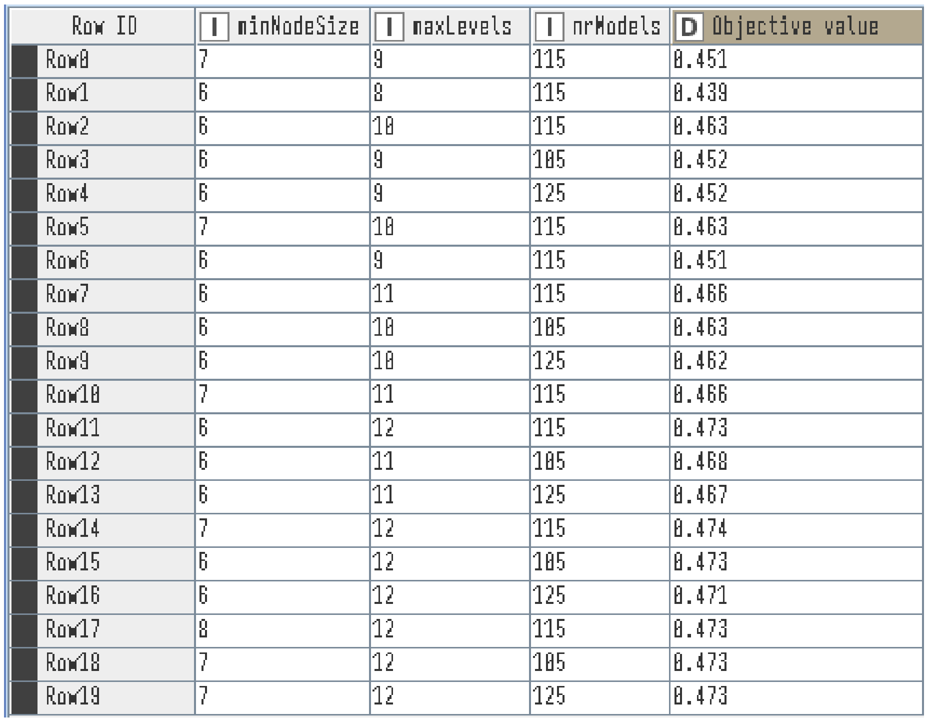
Each technique has also had computing resources dedicated to determining the optimal hyper-parameters using the random search method, and accuracy has been assessed using the weather data set with a randomly sampled (using the same seed for comparisons) 80/20 split between training and testing data.

4.1 Random Forest

Random forest model is an ensemble method that combines the output of multiple decision trees. Each tree is capable of predicting a boolean variable based on the provided information, however it is the majority of the trees that decide the final prediction.



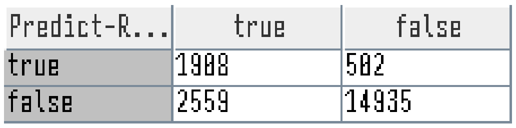
Three hyper parameters were optimised using including minimum node size, decision tree max depth and the number of trees.



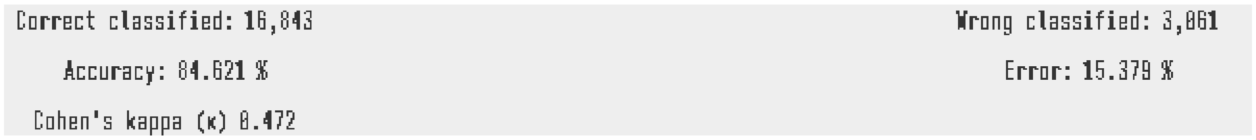
Row ID	I minNodeSize	I maxLevels	I nrModels	D Objective value
Row0	7	9	115	0.451
Row1	6	8	115	0.439
Row2	6	10	115	0.463
Row3	6	9	105	0.452
Row4	6	9	125	0.452
Row5	7	10	115	0.463
Row6	6	9	115	0.451
Row7	6	11	115	0.466
Row8	6	10	105	0.463
Row9	6	10	125	0.462
Row10	7	11	115	0.466
Row11	6	12	115	0.473
Row12	6	11	105	0.468
Row13	6	11	125	0.467
Row14	7	12	115	0.474
Row15	6	12	105	0.473
Row16	6	12	125	0.471
Row17	8	12	115	0.473
Row18	7	12	105	0.473
Row19	7	12	125	0.473

The optimal values are:
Minimum Node Size: 7
Maximum Tree Depth: 12
Number of Trees: 115

results:



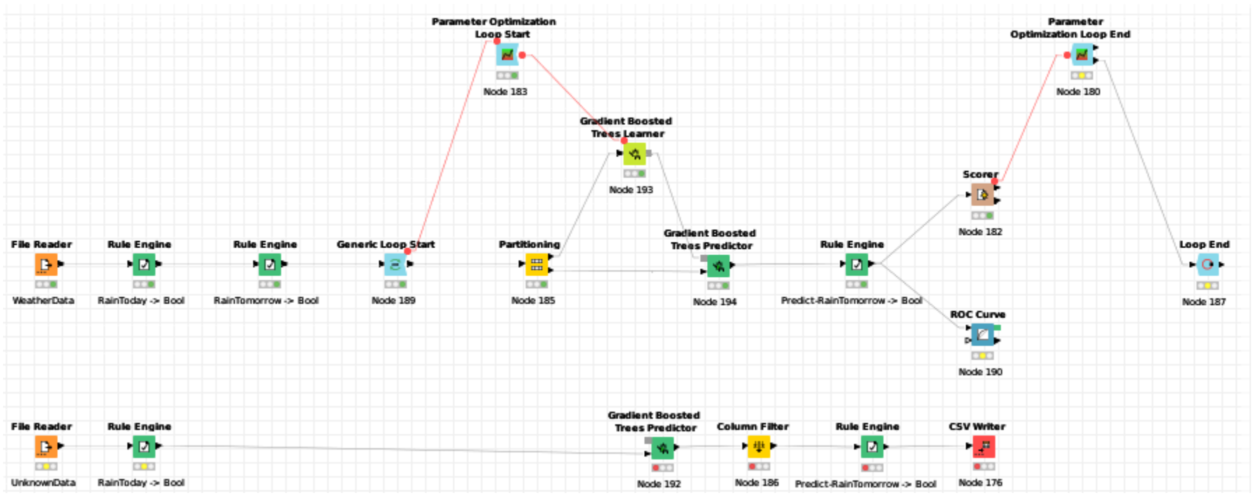
Predict-R...	true	false
true	1908	502
false	2559	14935



```
Correct classified: 16,843                                Wrong classified: 3,061

    Accuracy: 84.621 %                                      Error: 15.379 %

  Cohen's kappa (κ) 0.472
```

4.1.1 Random Forest Variations

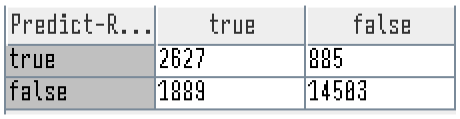
Variations of random forest were also tried, the most promising being gradient boosted trees learner, or more specifically, XGBoost. Gradient boosting is also an ensemble method that uses many smaller decision trees to predict a classification.



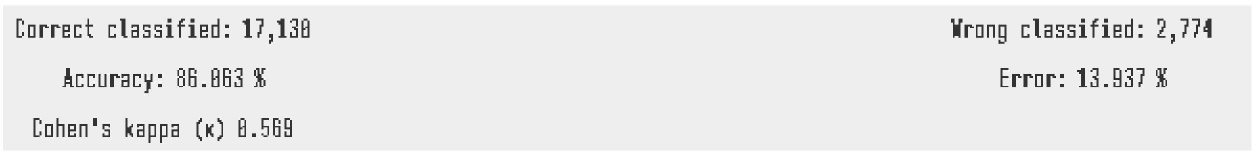
Three hyper parameters were optimised using random search including the learning rate, decision tree max depth and the number of trees.

The optimal values are:
Learning rate = 0.1
Max depth = 9
Number of estimators = 150

results:



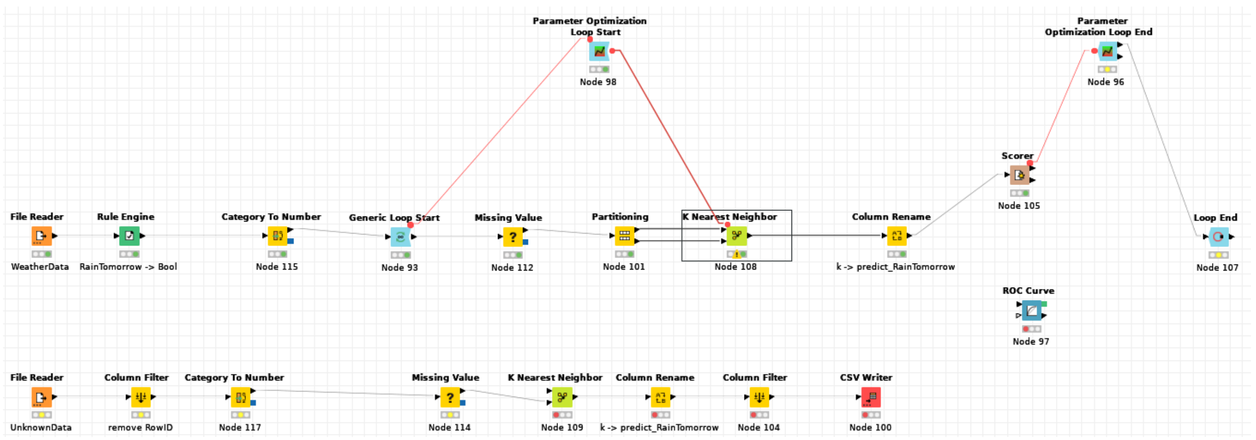
Predict-R...	true	false
true	2627	885
false	1889	14503



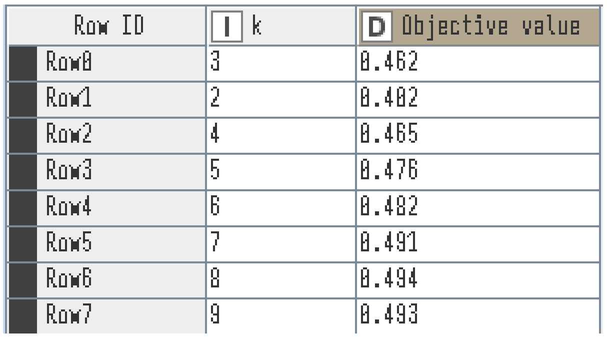
```
Correct classified: 17,130                                   Wrong classified: 2,774

    Accuracy: 86.063 %                                           Error: 13.937 %

  Cohen's kappa (κ) 0.569
```

4.2 K-Nearest Neighbour

K-Nearest Neighbour stores all available data and classifies new data based on its similarity determined by a plurality vote of its neighbours.



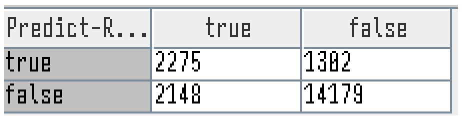
The number of neighbours considered was tested from 0 to 10:



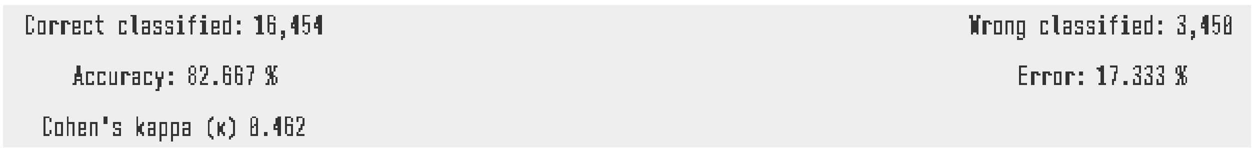
Row ID	k	Objective value
Row0	3	0.462
Row1	2	0.402
Row2	4	0.465
Row3	5	0.476
Row4	6	0.482
Row5	7	0.491
Row6	8	0.494
Row7	9	0.493

The optimal values are:
Number of neighbours: 8

Results:



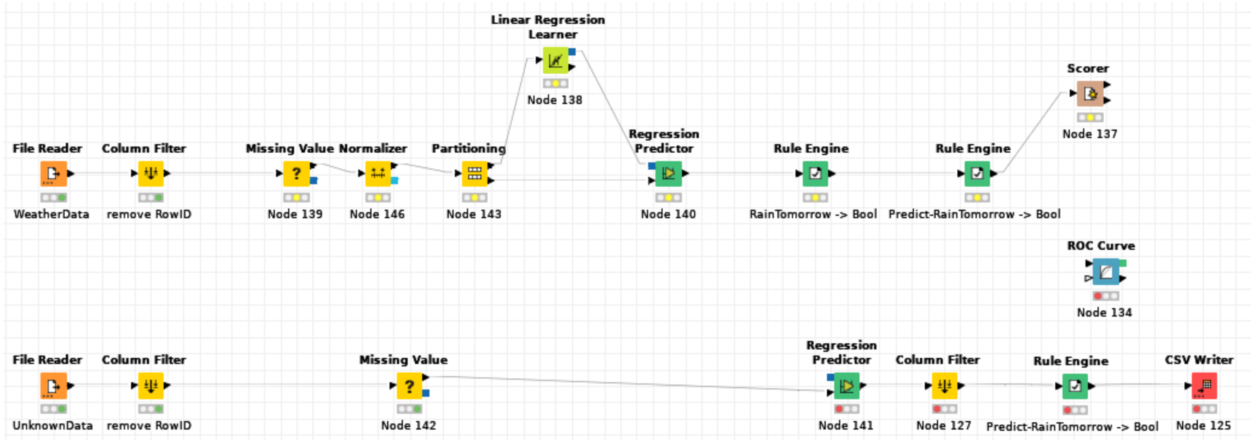
Predict-R...	true	false
true	2275	1302
false	2148	14179



```
Correct classified: 16,454                                  Wrong classified: 3,450

    Accuracy: 82.667 %                                          Error: 17.333 %

  Cohen's kappa (κ) 0.462
```

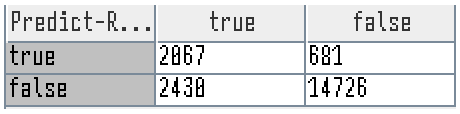
4.3 Linear Regression

Linear regression is the process of calculating the distance of points from the mean and computing how close it fits on a straight line. This method is then used to find potential linear relationships between data by plotting multiple data points together.

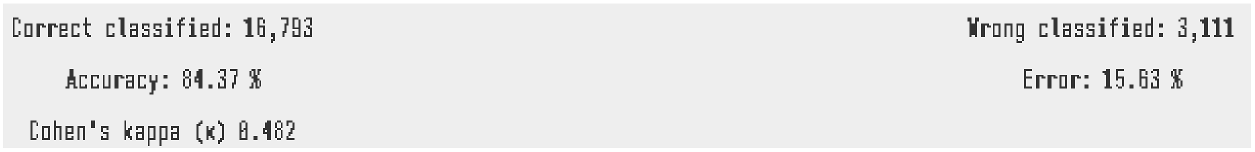


No hyper parameters were available for tuning.

Results:



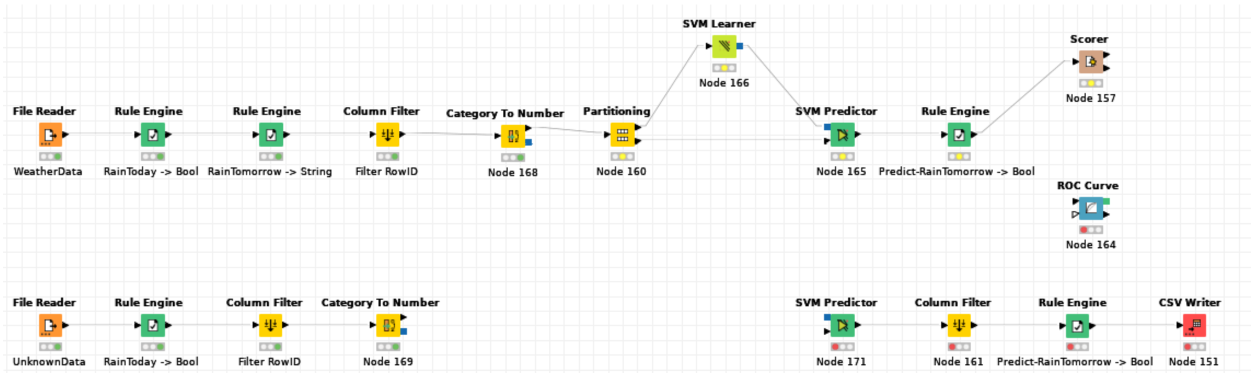
Predict-R...	true	false
true	2067	681
false	2430	14726



```
Correct classified: 16,793                                   Wrong classified: 3,111

    Accuracy: 84.37 %                                              Error: 15.63 %

  Cohen's kappa (κ) 0.482
```

4.4 SVM

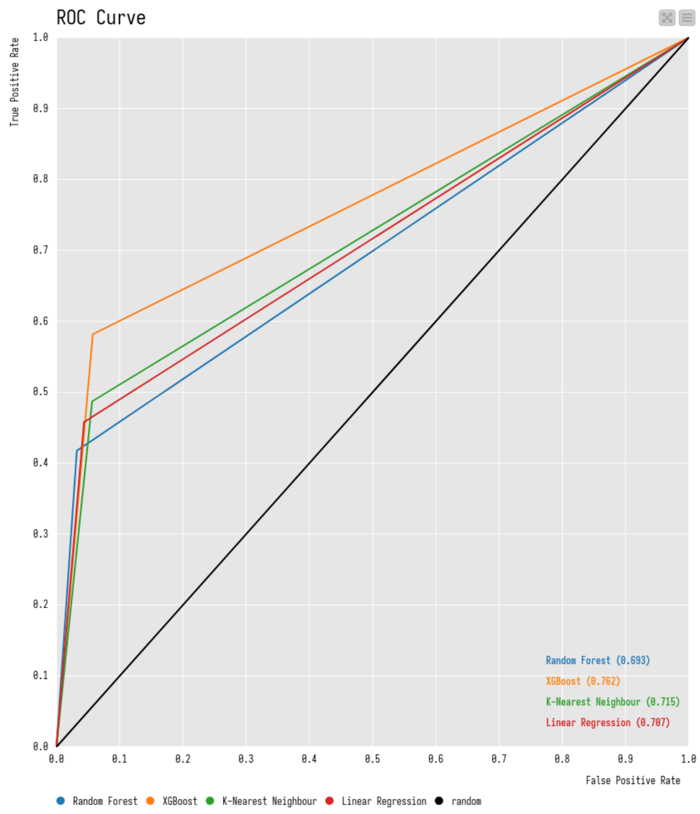
Support-vector machine is a supervised learning model that selects a small number of critical data points in order build a linear discriminant function that separates the data as widely as possible with a hyperplane.



Unfortunately due to time and computing resources constraints the svm classifier was not able to be thoroughly tested as planned. This node was estimated to take 68 consecutive hours with full CPU utilisation.

5. Best Classifier Selection

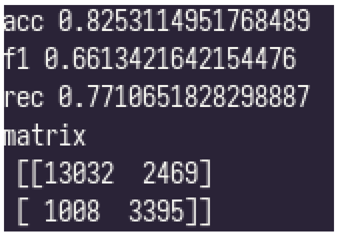
The first step in evaluating which classifier to use for the final submission was to compare the results with all preprocessing and transformations considered. A ROC curve has been used to visualise the accuracy difference between them:



Since XGBoost was by far the most successful despite having the least (at the time) preprocessing performed. It was selected for further experimentation.

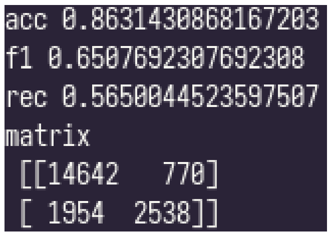
First, the XGBoost classifier was rewritten in the Python programming language using Scikit-Learn and Pandas libraries. This allowed for much more granular control over the program, increasing it's potential accuracy ceiling.The first major change after rewriting in Python was the added hyper parameters that could be tuned. The most impactful hyper parameter was *scale_pos_weight* which allows modifying the ratio of negative and positive classes which is now set to the number of negative classes divided by the number of positive classes. Since the dataset being trained only had ~22k recordings of positive classes and 77k negative classes, the classifier was skewed causing a confusion matrix with roughly as many true positive to true negative scores hindering the classifier's capability. However this hyper-parameter allowed the algorithm to adjust for this scenario.

Here are the results with and without *scale_pos_weight* using the same training and testing data:



```
acc 0.8253114951768489
f1 0.6613421642154476
rec 0.7710651828298887
matrix
 [[13032  2469]
 [ 1008  3395]]
```

With *scale_pos_weight*

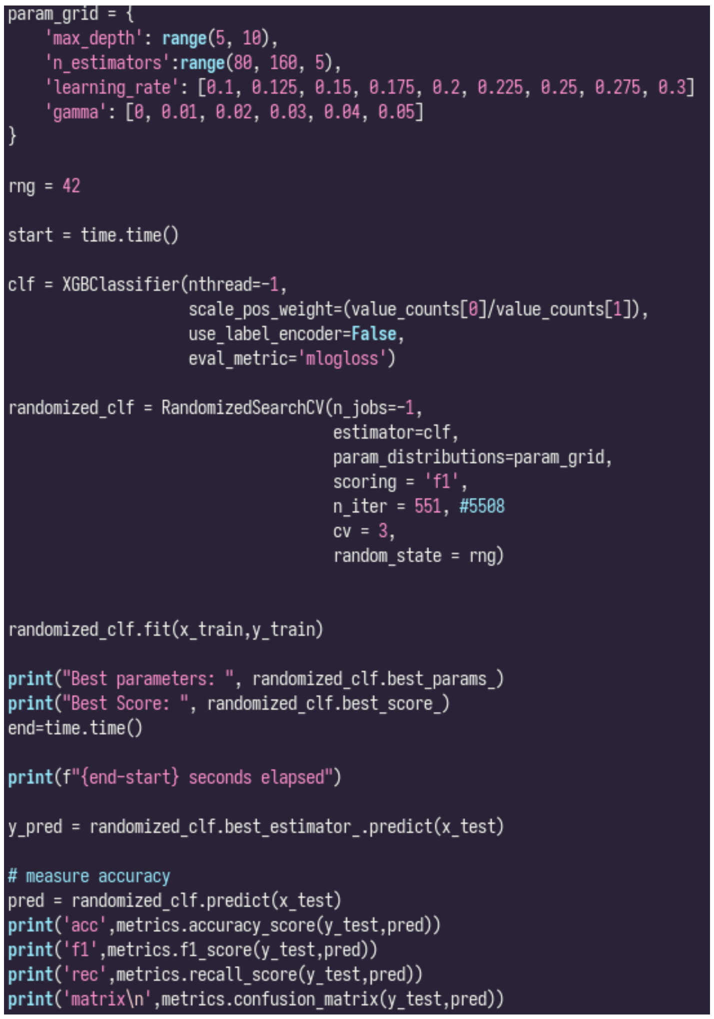


```
acc 0.8631430868167203
f1 0.6507692307692308
rec 0.5650044523597507
matrix
 [[14642   770]
 [ 1954  2538]]
```

Without *scale_pos_weight*

While the overall accuracy might be higher, the recall score has significantly decreased with up to double the amount of true negative predictions when *scale_pos_weight* was removed.

Furthermore, more intensive hyper-parameter tuning was performed using the code shown:



```
param_grid = {
    'max_depth': range(5, 10),
    'n_estimators':range(80, 160, 5),
    'learning_rate': [0.1, 0.125, 0.15, 0.175, 0.2, 0.225, 0.25, 0.275, 0.3]
    'gamma': [0, 0.01, 0.02, 0.03, 0.04, 0.05]
}

rng = 42

start = time.time()

clf = XGBClassifier(nthread=-1,
                    scale_pos_weight=(value_counts[0]/value_counts[1]),
                    use_label_encoder=False,
                    eval_metric='mlogloss')

randomized_clf = RandomizedSearchCV(n_jobs=-1,
                                    estimator=clf,
                                    param_distributions=param_grid,
                                    scoring = 'f1',
                                    n_iter = 551, #5508
                                    cv = 3,
                                    random_state = rng)


randomized_clf.fit(x_train,y_train)

print("Best parameters: ", randomized_clf.best_params_)
print("Best Score: ", randomized_clf.best_score_)
end=time.time()

print(f"{end-start} seconds elapsed")

y_pred = randomized_clf.best_estimator_.predict(x_test)

# measure accuracy
pred = randomized_clf.predict(x_test)
print('acc',metrics.accuracy_score(y_test,pred))
print('f1',metrics.f1_score(y_test,pred))
print('rec',metrics.recall_score(y_test,pred))
print('matrix\n',metrics.confusion_matrix(y_test,pred))
```

Final parameters:

max_depth: 7

n_estimators: 140

learning_rate: 0.125

gamma: 0.025

scale_pos_weight: (number of negative classes)/(number of positive classes)