

Data Mining in Action



Table of Contents

Table of Contents	2
Introduction	3
Data Mining Problem.....	3
Input	3
Output	4
Data Preprocessing	4
Column Filter	4
Missing Value.....	6
Normalizer	7
Partitioning	7
Number to String & String to Number	8
Smote	10
Classifiers.....	11
Decision Tree	11
Random Forest	15
K Nearest Neighbour (KNN)	18
Naïve Bayes	22
Multi-Layer Perceptron.....	25
SVM Learner	29
Best Classifiers	33
Results.....	33
Kaggle Submission	34
Solution for Data Mining Problem	35
Parameter Optimisation	36
Cross Validation.....	36
Conclusion	37
Appendix.....	38

Introduction

In meteorology, predicting future weather conditions is crucial for a variety of industries, including agriculture, logistics, and public safety. Data mining techniques provide a powerful way to analyse historical weather data and extract patterns that help in making accurate forecasts. This assignment focuses on using machine learning methods to develop models that can predict whether it will rain tomorrow, a binary classification problem based on historical weather observations.

The primary goal of this task is to build and evaluate different classifiers to determine which model can most accurately predict rainfall. The process begins with exploring and pre-processing the dataset, followed by applying various classification algorithms such as Decision Trees, Random Forests, and Support Vector Machines. Each model will be fine-tuned and cross-validated to ensure robust performance, with the final predictions being submitted for evaluation on the Kaggle platform.

Through this analysis, the report will not only aim to find the most effective model for rainfall prediction but also explore the impact of weather attributes like temperature, humidity, and wind speed on the outcome. By comparing multiple machine learning techniques, the assignment provides an opportunity to understand the strengths and limitations of each model in real-world forecasting scenarios.

Data Mining Problem

The primary data mining problem in this scenario revolves around predicting whether it will rain tomorrow (RainTomorrow attribute) based on various weather-related attributes. This binary classification task is essential for industries like agriculture, logistics, disaster management, and public safety, where anticipating weather conditions is crucial.

The goal is to accurately predict RainTomorrow, which can either be 0 (No) or 1 (Yes), by using various machine learning classifiers. In this project, several classification techniques such as Decision Trees (DT), Random Forests (RF), Support Vector Machines (SVM), and K Nearest Neighbour (KNN) will be explored. These classifiers will be trained on historical weather data (WeatherData.csv) to make predictions on an unknown dataset (UnknownData.csv).

The process of data mining involves several stages, including data pre-processing, model building, and evaluation. The classifiers will be trained, validated, and tested to ensure accurate predictions. The performance of each classifier will be evaluated based on metrics like accuracy, precision, recall, and the area under the ROC curve (AUC). The ultimate aim is to determine which model is best suited for predicting rainfall in the given context, which will be assessed through a Kaggle competition.

Input

The input data for this assignment comes from a weather dataset labelled WeatherData.csv, which contains multiple features related to weather conditions such as temperature, wind speed, humidity, and cloud cover. These features are crucial for predicting the RainTomorrow attribute.

- **Dataset Characteristics:** The dataset includes various attributes (e.g. Temperature9am, Humidity3pm, Cloud9am) that describe the weather at different times of the day. Some of these attributes may contain missing values or require normalization and transformation for the models to function optimally.
- **Target Variable:** The target variable for the prediction task is RainTomorrow, which is binary, indicating whether it will rain (1) or not (0) the next day.
- **Unknown Dataset:** In addition to the training dataset, we're provided with an UnknownData.csv file, which contains similar features but lacks the RainTomorrow target attribute. This file will be used to make predictions after the models are trained.

In summary, the input consists of weather-related features used to build and optimize machine learning models, and the unknown dataset on which the final predictions will be made.

Output

The desired output of this data mining task is to build machine learning models that can accurately predict whether it will rain tomorrow. These models will be trained on the weather data and evaluated based on their performance in predicting the RainTomorrow attribute for the unknown dataset.

- **Prediction Output:** The classifiers will output predictions in the form of 0 (No) or 1 (Yes) for each record in the unknown dataset. The results will be submitted to Kaggle for evaluation. The final submission must adhere to the specified format (row ID and PredictRainTomorrow columns).
- **Evaluation Metrics:** The performance of the classifiers will be assessed using metrics such as accuracy, precision, recall, F1 score, and AUC. These metrics will help determine which classifier is most effective at predicting rainfall based on the historical data. A good classifier is one that strikes a balance between these metrics, ensuring robust predictions across different weather scenarios.
- **Kaggle Submission:** The predictions for the unknown dataset will be uploaded to the Kaggle platform, where they will be scored and ranked. The Kaggle score, based on the classifier's performance, will reflect the overall accuracy of the model in predicting RainTomorrow. While Kaggle provides a leaderboard ranking, the final evaluation will be based on the robustness of the model and the report analysis.

Ultimately, the best-performing model will be selected based on its accuracy, AUC score, and other evaluation metrics, ensuring that it generalizes well to new data and provides meaningful insights for weather forecasting.

Data Preprocessing

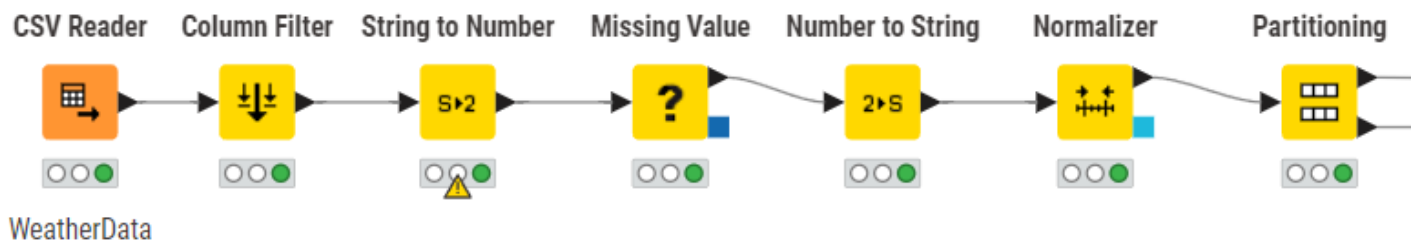


Figure 1 – Basic Data Preprocessing Nodes (excluding Smote) used in models

Column Filter

The Column Filter node is a critical step in data preprocessing, as it allows the user to selectively retain or remove attributes (columns) from the dataset. This step ensures that only the most relevant features are passed to the machine learning algorithms, thus improving model efficiency and accuracy. By excluding irrelevant or redundant attributes, the model can focus on the key factors that are most likely to influence the target variable, RainTomorrow.

Additionally, the Column Filter helps reduce computational load by removing unnecessary data, making the modelling process more efficient, especially when dealing with large datasets. By applying this filter, the dataset is streamlined, enabling the classifier to learn and generalise more effectively, which is essential for generating accurate predictions on unseen data.

Python was utilised to analyse the dataset for missing values before applying the Column Filter node. Through Python, we were able to identify several columns with a significant proportion of missing data. The results showed that some attributes, such as Evaporation, Sunshine, Cloud9am, and Cloud3pm, had more than 30% of their data missing, with Evaporation missing around 42.9% and Sunshine missing about 47.8%. Due to the large amount of missing data, we excluded these columns from the dataset using the Column Filter node, as imputing such a significant proportion of missing values could introduce too much noise, reduce the model's accuracy and be troublesome to effectively remove it with the Missing Values node.

In addition to handling missing values, we also removed columns that are less relevant for predicting rainfall, such as direction-based attributes like WindGustDir, WindDir9am, and WindDir3pm. Directional data does not directly correlate with rainfall prediction, and its inclusion could introduce noise, reducing the model's overall accuracy. Instead, we focused on weather metrics that have a clearer relationship with rainfall, such as temperature, humidity, wind speed, and past rainfall.

```
Python > Python Workspace.py > ...
1  import pandas as pd
2
3  # Load the dataset.
4  data = pd.read_csv('E:\Assignments\Data Analytics - Assessment 3/Assignment3-WeatherData.csv')
5
6  # Check for missing values.
7  missing_values = data.isnull().sum()
8
9  # Find the column with the most missing values.
10 most_missing = missing_values.idxmax(), missing_values.max()
11
12 # Calculate the percentage of missing values for each column.
13 missing_percentage = (data.isnull().sum() / len(data)) * 100
14
15 print(most_missing)
16 print(missing_percentage)
```

Figure 2 – Python Code to Calculate Missing Values Percentage

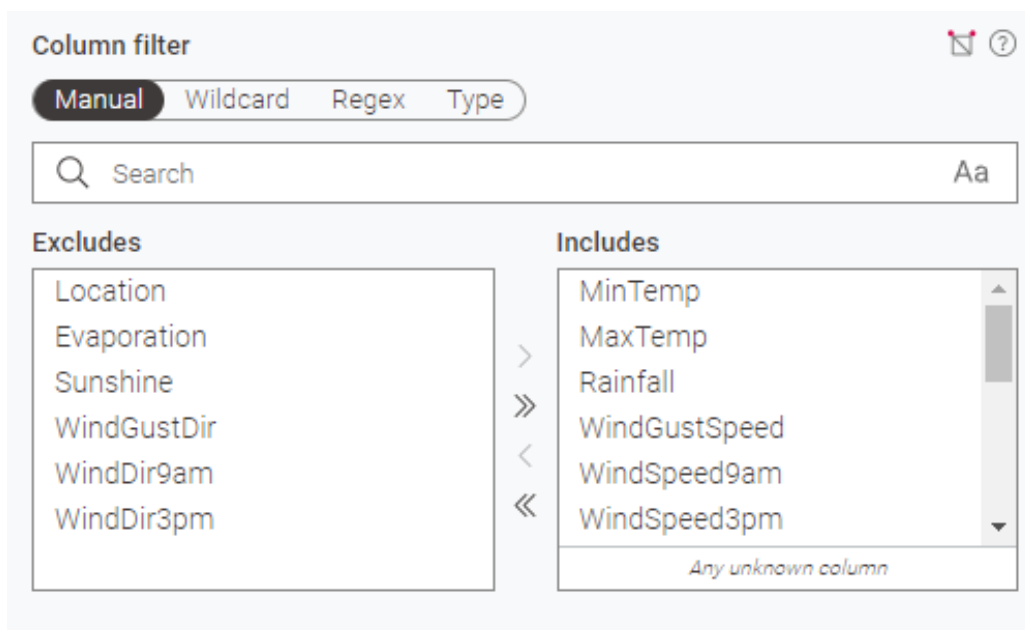


Figure 3 – Column Filter Node Configuration Window

Missing Value

Handling missing values is a crucial step in data preprocessing. Missing data can lead to inaccurate models or reduce the model's ability to generalise to new data. The Missing Value node in KNIME is used to handle instances where some attributes lack information. If left unresolved, missing values could distort the model's predictions or even cause errors during model training.

In the KNIME workflow, we address missing values using different strategies based on the data type of the attribute:

- For string attributes (e.g. categorical variables like RainToday), the missing values are replaced by the most frequent value in the dataset. This imputation method assumes that the most frequent category is a reasonable substitute for missing entries.
- For numerical attributes (e.g. temperature, rainfall), the missing values are imputed using the median. The median is less sensitive to outliers compared to the mean and thus provides a more robust way to handle missing numeric data.

These strategies ensure that all attributes contain valid entries, which helps prevent errors during model training. By imputing missing values, the model can utilise all available data for better accuracy and performance. Using these imputation methods also ensures that we retain as much data as possible without having to discard records with missing values. This is particularly important in weather data, where missing observations can be common but still contain valuable information.

The following table shows the percentage of missing values above 5% for each attribute, which was calculated using the Python code in Figure 2.

Attributes	Percentage of Rows Without Data (%)
Evaporation	42.91
Sunshine	47.81
WindGustDir	6.49
WindGustSpeed	6.44
WindDir9am	7.10
Pressure9am	9.88
Pressure3pm	9.89
Cloud9am	37.90
Cloud3pm	40.33

Figure 4 – Attributes Missing Data (>5%)

The screenshot shows the 'Missing Value Node Configuration Window' in KNIME. The window has a tabbed interface with 'Default' selected. Below the tabs, there are two rows of configuration options. The first row is for 'String' data types, with a dropdown menu set to 'Most Frequent Value'. The second row is for 'Number (integer)' data types, with a dropdown menu set to 'Median'. At the bottom of the window, there are four buttons: 'OK', 'Apply', 'Cancel', and a help icon (a question mark inside a circle).

Figure 5 – Missing Value Node Configuration Window

Normalizer

The Normalizer node plays a vital role in data preprocessing by transforming the range of numerical attributes into a standard scale. The normalizer is applied to ensure that all numerical features, such as MinTemp, MaxTemp, Rainfall, and others, are on the same scale before being inputted into the classifier. This step is particularly important for machine learning algorithms like Support Vector Machines (SVM) or K-Nearest Neighbours (KNN), which are sensitive to the scale of input data.

By normalizing the data, we ensure that no single attribute disproportionately affects the performance of the model. For instance, features like temperature and rainfall have different units and ranges, and without normalization, the model may overemphasise features with larger numeric values, such as temperature. The Normalizer node adjusts the values so that all features contribute equally, improving the model's learning process and overall performance. In this workflow, the normalizer is applied after handling missing values and before splitting the data into training and testing sets. This ensures that the model is trained on normalized data, leading to more consistent and reliable results.

The screenshot shows the 'Normalizer' node configuration window. At the top, there are four tabs: 'Manual' (selected), 'Wildcard', 'Regex', and 'Type'. Below the tabs is a search bar with a magnifying glass icon and the text 'Search', followed by a text input field with 'Aa'. The main area is divided into two columns: 'Excludes' on the left and 'Includes' on the right. The 'Excludes' column is empty and contains the text 'No columns in this list'. The 'Includes' column contains a list of features: 'MinTemp', 'MaxTemp', 'Rainfall', 'WindGustSpeed', 'WindSpeed9am', and 'WindSpeed3pm'. There are navigation arrows between the two columns. At the bottom, there is a section titled 'Normalization method' with three radio buttons: 'Min-max', 'Z-score' (selected), and 'Decimal scaling'.

Figure 6 – Normalizer Node Configuration Window

Partitioning

The Partitioning node is used to split the dataset into training and testing subsets, which is crucial for evaluating the performance of the machine learning models. The dataset is typically split into two parts:

- **Training Set:** A portion of the dataset (e.g. 70-80%) used to train the model. The model learns from this data and adjusts its parameters to make predictions.
- **Testing Set:** The remaining portion of the dataset (e.g. 20-30%) used to evaluate the model's performance on unseen data. This helps ensure that the model generalizes well to new data and isn't overfitting to the training set.

In the current KNIME workflow, after normalizing the data, the partitioning node splits the dataset into these two subsets. The ratio is usually 70% for training and 30% for testing. This partitioning step is essential for validating the model's accuracy and ensuring that it can make reliable predictions on data it has not seen before. A well-partitioned dataset helps avoid overfitting and ensures that the model can perform well on unseen data.

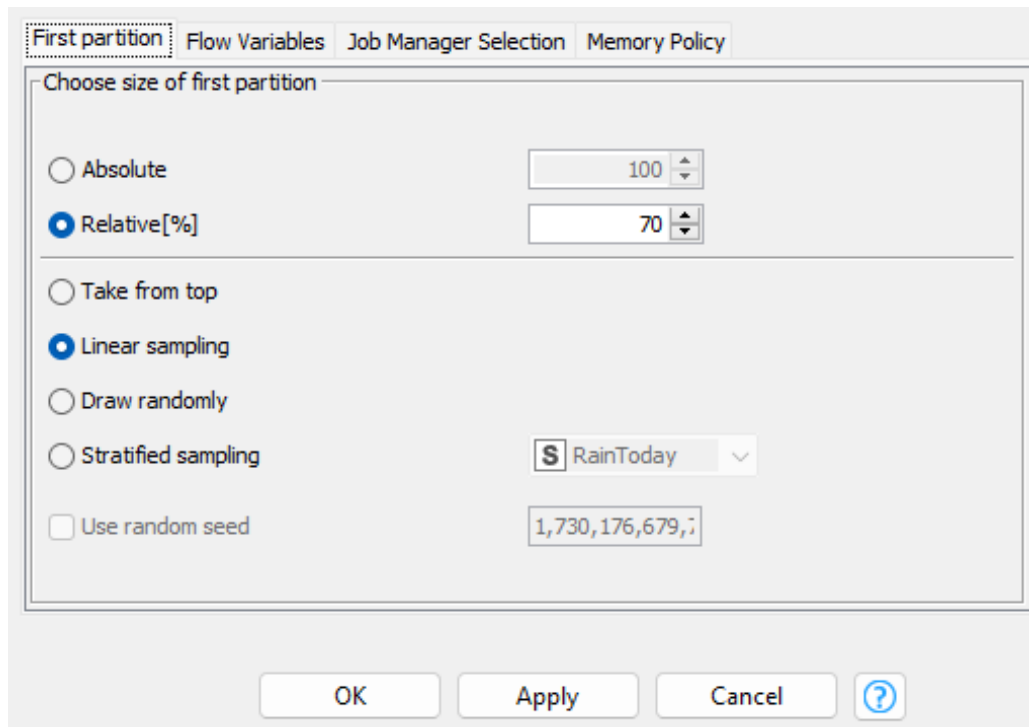


Figure 7 – Partitioning Node Configuration Window

Number to String & String to Number

The Number to String node is used to convert numerical data types into string (categorical) formats. This conversion is particularly important when certain columns in the dataset, although numeric, are better suited for categorical interpretation. For example, the target variable, RainTomorrow, is a binary numeric attribute (0 for No and 1 for Yes). While these numbers represent categories, treating them as numerical values may lead to inappropriate assumptions in some models.

The Number to String node is applied to the RainTomorrow column. By converting this attribute from numerical (0, 1) to categorical (No, Yes), the models can better interpret the target variable as a classification task rather than a regression task. This step ensures that the machine learning algorithms correctly handle the binary output as a categorical classification problem, improving the clarity and accuracy of the predictions. This conversion also makes the data more interpretable for both humans and the machine learning algorithms, ensuring that the correct assumptions are made during the model-building process.

Conversely, the String to Number node is used to convert categorical data back into numerical format when needed, especially for features that originally have numerical meaning. The String to Number node was applied to attributes like RainToday, which is represented as "Yes" or "No" but holds a binary indication of whether it rained on the previous day. By converting these categorical strings back into numbers (1 for Yes and 0 for No), the model can utilise these attributes in calculations or analyses that require numerical data types, such as normalization.

This two-way conversion process: Number to String for categorical interpretation and String to Number for numeric computations ensures that each attribute is in the most appropriate format for the task at hand. By using these conversions, we optimised the data structure to align with the machine learning requirements, ultimately enhancing the model's performance and interpretability.

Columns

Manual Wildcard Regex

Search Aa

Excludes

- MinTemp
- MaxTemp
- Rainfall
- WindGustSpeed
- WindSpeed9am
- WindSpeed3pm
- Any unknown column

Includes

- RainTomorrow

Figure 8 – Number to String Node Configuration Window

Column selection

Manual Wildcard Regex

Search Aa

Excludes

- RainToday

Includes

- MinTemp
- MaxTemp
- Rainfall
- WindGustSpeed
- WindSpeed9am
- WindSpeed3pm
- Any unknown column

Parsing options

Decimal separator

Thousands separator

Type

Number (Double)

☐ Accept type suffix, e.g. 'd', 'D', 'f', 'F'

☐ Fail on error

Figure 9 – String to Number Node Configuration Window

Smote

The SMOTE (Synthetic Minority Over-sampling Technique) node is used to address the issue of class imbalance in the dataset, which can significantly impact the performance of machine learning models. The target variable RainTomorrow exhibits an imbalance, with fewer instances of rainy days (RainTomorrow = 1) compared to non-rainy days (RainTomorrow = 0). This imbalance can lead to a model that is biased towards predicting non-rainy days, as it encounters fewer rainy examples during training.

SMOTE helps counteract this by generating synthetic instances of the minority class (rainy days) rather than simply duplicating existing examples. It does this by creating new instances based on the feature space similarity between existing minority class instances. In the SMOTE node configuration, we set the Class column to RainTomorrow and selected the option to oversample minority classes. The # Nearest neighbour parameter was set to 5, allowing the SMOTE algorithm to create synthetic instances by interpolating between an instance of the minority class and its 5 nearest neighbours within that class.

By oversampling the minority class, SMOTE enables the model to learn a more balanced view of both classes, which improves its ability to correctly predict both rainy and non-rainy days. This approach leads to better performance metrics, particularly in recall and F1-score for the minority class, and reduces the risk of overfitting to the majority class.

In the workflow, SMOTE is applied after data partitioning to ensure that the synthetic instances are only present in the training set and not in the test set. This maintains the integrity of the test data, allowing us to evaluate the model's performance on genuine, unseen data.

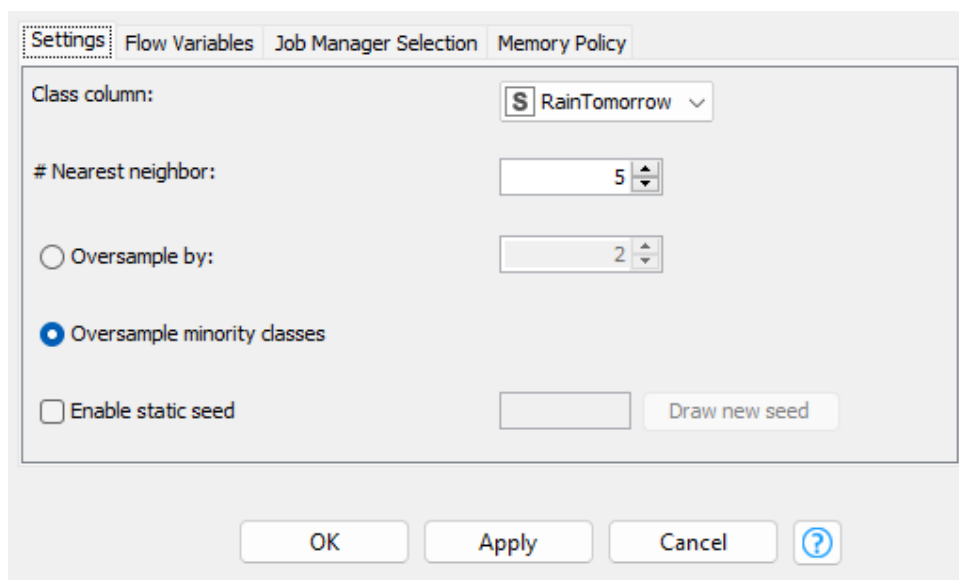


Figure 10 – Smote Node Configuration Window

Decision Tree



Figure 11 – Decision Tree Preprocessing Nodes (w/ Smote)

Classifiers

Multiple machine learning classifiers are employed to predict whether it will rain tomorrow (RainTomorrow). Each classifier has its own unique approach to solving classification problems, and the goal is to determine which one provides the most accurate predictions for the weather data. The performance of these models is evaluated using metrics such as accuracy, recall, precision, and the area under the ROC curve (AUC), helping to identify the best performing algorithm for this task.

The classifiers explored in this analysis include:

- **Decision Tree:** A simple yet powerful model that splits data based on feature values to create a tree-like structure, making predictions based on the learned splits.
- **Random Forest:** An ensemble learning method that builds multiple decision trees and combines their predictions to improve accuracy and reduce overfitting.
- **K-Nearest Neighbour (KNN):** A distance-based classifier that predicts the class of a data point by looking at the classes of its closest neighbours.
- **Naïve Bayes:** A probabilistic classifier based on applying Bayes' theorem, assuming that the features are independent of each other.
- **Multilayer Perceptron (MLP):** A complex, multi-layered neural network model inspired by the human brain that learns to classify data by adjusting weights based on error minimization.
- **SVM (Support Vector Machine) Learner:** A classifier that constructs a hyperplane or set of hyperplanes in a high-dimensional space to separate different classes.

Decision Tree

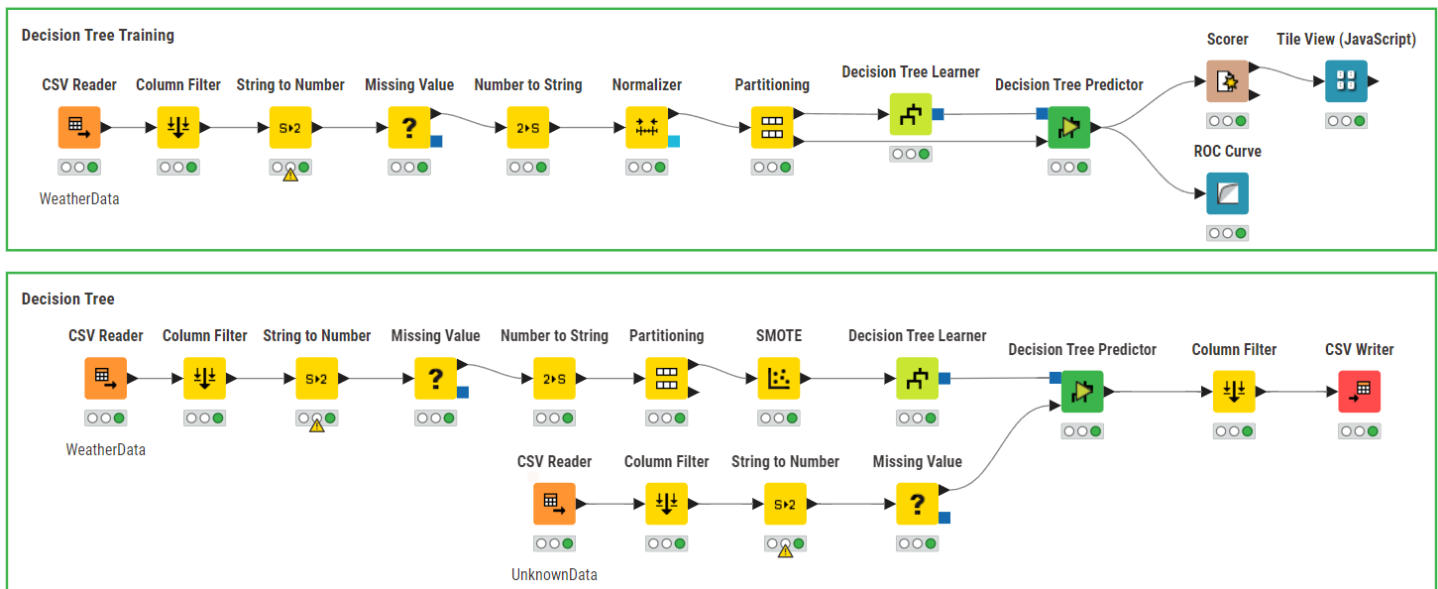


Figure 12 – Decision Tree KNIME Workflow

The Decision Tree algorithm is an intuitive and widely used classification method in machine learning, especially suitable for datasets with categorical or numerical data. The algorithm works by splitting the dataset into branches based on feature values, creating a tree-like structure where each internal node represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label or decision.

For each split, the algorithm chooses the attribute that maximises the information gain or minimises impurity, using measures like the Gini Index or Gain Ratio. In this workflow, the Gain Ratio was selected as the splitting criterion, which helps the model decide how to partition data at each node effectively by taking into account the entropy reduction achieved by the split.

Decision Trees are attractive for their interpretability; one can follow the path of decisions to understand how the model arrives at a particular classification. However, they are also prone to overfitting, especially with complex trees, as they tend to capture noise along with patterns. To counteract this, pruning techniques like MDL (Minimum Description Length) Pruning and Reduced Error Pruning are used in this configuration. These methods remove nodes that don't significantly improve the model's performance, thereby simplifying the tree and enhancing generalizability on unseen data.

Data Preprocessing Nodes:

Preprocessing is essential for preparing the data before feeding it into the Decision Tree model. The pipeline includes various nodes:

- **CSV Reader** loads the dataset.
- **Column Filter** selects relevant features while excluding attributes irrelevant to predicting RainTomorrow.
- **String to Number** and **Number to String** conversions ensure data types are compatible with the model.
- **Missing Value** handles missing data, which is crucial for maintaining model accuracy.
- **Normalizer** standardises numerical features, which can help the model if feature scales differ significantly.
- **Partitioning** splits the data into training and testing sets for validation.
- **SMOTE** balances the dataset by oversampling minority classes to prevent biased predictions.

Model Configuration:

In the Decision Tree Learner node, several configurations were fine-tuned to maximize model performance:

- **Class Column:** The target variable RainTomorrow is set, which indicates whether it will rain the next day (Yes or No).
- **Quality Measure:** The Gain Ratio is used as the splitting criterion. Gain Ratio adjusts for information gain based on the feature's intrinsic information, helping to mitigate bias towards attributes with many distinct values. This choice helps create balanced splits that are more likely to generalise well on new data.
- **Pruning Method:** MDL (Minimum Description Length) Pruning is selected to prevent overfitting. This method balances tree complexity against the risk of overfitting by choosing a tree that minimises both error and tree size.
- **Reduced Error Pruning:** This option is also enabled, which prunes the tree based on a cross-validation process, further reducing complexity and enhancing the model's ability to generalise.
- **Minimum Records per Node:** A minimum threshold of 20 records per node is set to prevent the model from creating overly specific branches that capture noise. By setting this threshold, the tree remains broad and general enough to apply to unseen instances effectively.
- **Number of Threads:** The computation is optimized by setting multiple threads, allowing the tree to be built more quickly by utilising multi-core processing.

Results and Evaluation:

The Decision Tree classifier's performance was evaluated using various metrics:

- **Accuracy and F1 Score:** The Decision Tree classifier achieved an accuracy of 0.836 on the training dataset and 0.74180 on the Kaggle test dataset, indicating a reasonably good fit but with potential signs of overfitting. The F1 score, a balance between precision and recall, further demonstrated the model's ability to manage the trade-off in predicting both classes, especially the dominant no rain days.
- **ROC Curve and AUC (Area Under the Curve):** The ROC curve illustrated the model's trade-off between sensitivity (true positive rate) and specificity (false positive rate) for both classes. The blue curve (AUC = 0.871) represents the model's capability in accurately predicting no rain days, showing a high level of precision. Conversely, the green curve (AUC = 0.129) indicates the classifier's performance on rain days, which is significantly lower, despite attempts to handle imbalance using SMOTE oversampling. This discrepancy highlights the classifier's struggle to effectively predict the minority class (rain days).
- **Confusion Matrix (True Positives, False Positives, etc.):** The confusion matrix analysis revealed that the classifier was effective on the majority class (RainTomorrow=0), achieving high recall and precision for non-rainy days. However, the model struggled with the minority class (RainTomorrow=1), resulting in a lower recall score of 0.469 for rainy days. This imbalance may contribute to the model's lower generalizability in predicting rain days accurately, despite the dataset adjustments.
- **Precision, Recall, and Specificity:** These metrics were analysed separately for each class. For RainTomorrow=0, the classifier demonstrated high precision (0.86) and recall (0.941), suggesting its strong performance in predicting non-rainy days with minimal false positives. For RainTomorrow=1, however, the model showed lower precision (0.698) and recall (0.469), which indicates that further model tuning or additional data balancing techniques could be necessary to improve minority class predictions.

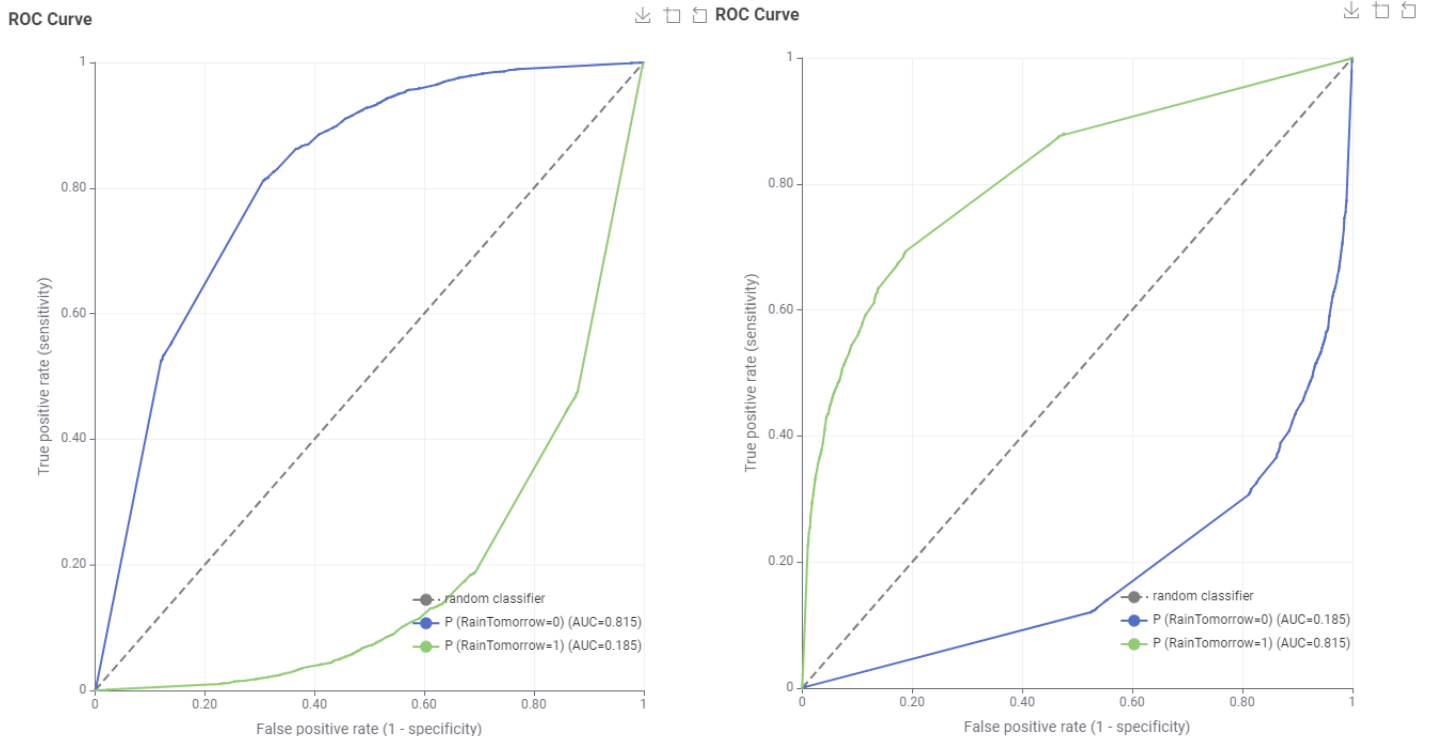


Figure 13 – Decision Tree ROC Curve

Options PMMLSettings Flow Variables Job Manager Selection

General

Class column

Quality measure

Pruning method

☒ Reduced Error Pruning

Min number records per node

Number records to store for view

☒ Average split point

Number threads

☒ Skip nominal columns without domain information

Root split

☐ Force root split column

Root split column

Binary nominal splits

☐ Binary nominal splits

Max #nominal

☐ Filter invalid attribute values in child nodes

Figure 14 – Decision Tree Learner Configuration Window

File Hilite

RainTomor...	0	1
0	11227	699
1	1822	1612

Correct classified: 12,839 Wrong classified: 2,521

Accuracy: 83.587% Error: 16.413%

Cohen's kappa (κ): 0.465%

Figure 15 – Decision Tree Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	11,227	1,822	1,612	699	0.941	0.86	0.941	0.469	0.899	Null	Null
1	1,612	699	11,227	1,822	0.469	0.698	0.469	0.941	0.561	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.836	0.465

Figure 16 – Decision Tree Accuracy Statistic

Random Forest

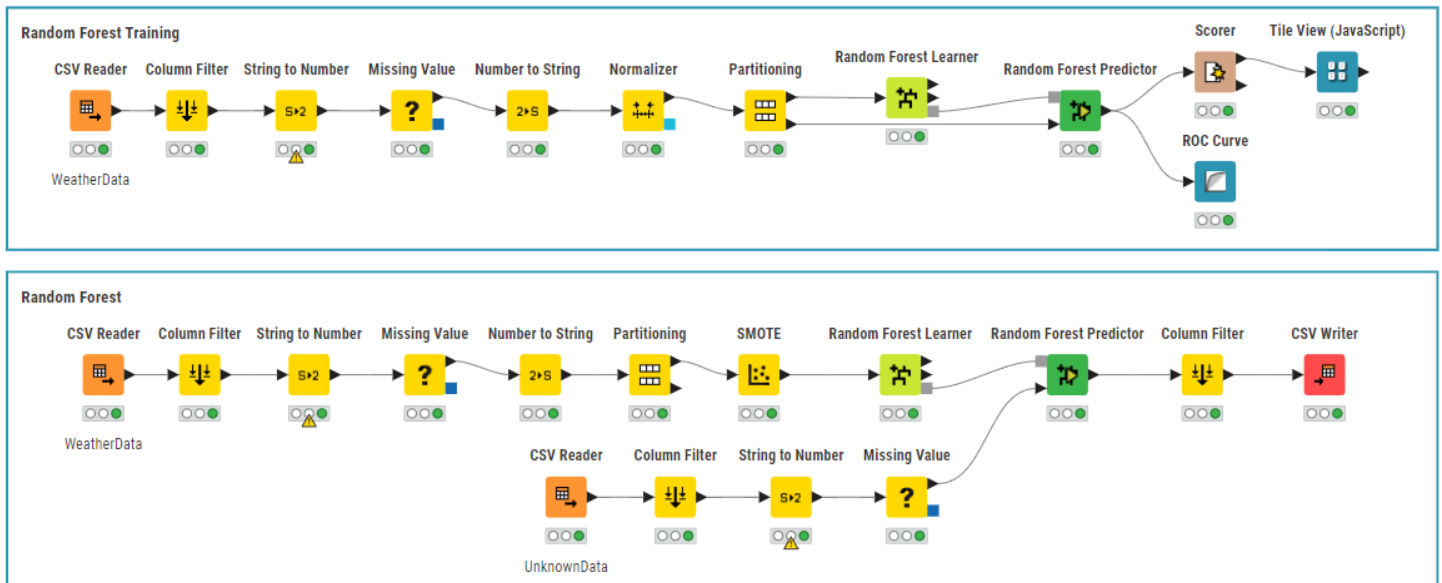


Figure 17 – Random Forest KNIME Workflow

Random Forest is a powerful ensemble learning technique widely used for classification tasks. It consists of a collection, or “forest,” of decision trees that each independently predict the output. The final prediction is obtained by aggregating the outputs of these individual trees, typically through majority voting for classification tasks. This method is particularly effective because it reduces the risk of overfitting that can occur with a single decision tree, thereby increasing both the model’s stability and accuracy.

Random Forest relies on the concept of bootstrap aggregation, or bagging, where each decision tree is trained on a random subset of the training data with replacement. Furthermore, during the training of each tree, Random Forest randomly selects a subset of features for each split, which ensures that the trees are diverse and uncorrelated. This diversity among the trees allows Random Forest to generalise well on unseen data, as it minimises variance while maintaining low bias. In essence, Random Forest is a robust algorithm that performs well on complex datasets by capturing intricate patterns without becoming overly sensitive to noise.

Data Preprocessing Nodes:

To prepare the data for training, several preprocessing steps were employed, each serving a specific purpose in enhancing data quality and model interpretability. These nodes included:

- **CSV Reader** loads the dataset.
- **Column Filter** selects relevant features while excluding attributes irrelevant to predicting RainTomorrow.
- **String to Number** and **Number to String** conversions ensure data types are compatible with the model.
- **Missing Value** handles missing data, which is crucial for maintaining model accuracy.
- **Normalizer** standardises numerical features, which can help the model if feature scales differ significantly.
- **Partitioning** splits the data into training and testing sets for validation.
- **SMOTE** balances the dataset by oversampling minority classes to prevent biased predictions.

Model Configuration:

The configuration of the Random Forest model was tailored to balance performance and computational efficiency:

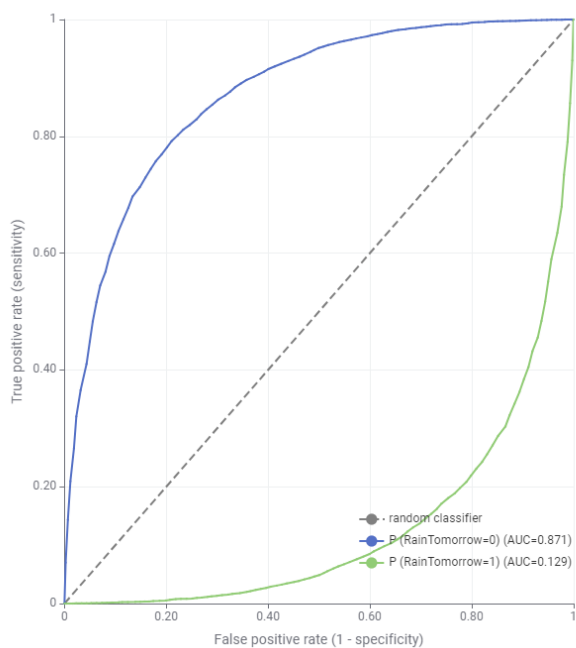
- **Split Criterion:** Set to Information Gain Ratio, which helps select the most informative splits. This criterion works well with imbalanced data as it favours splits that improve class purity.
- **Number of Trees:** Configured to 100 trees, providing a balance between computational efficiency and predictive power. Increasing the number of trees improves model performance but comes with increased computation time. With 100 trees, the model can capture complex patterns in the data without excessive runtime.
- **Static Random Seed:** A random seed was set to ensure reproducibility. This is crucial in ensemble models to ensure that the results remain consistent across different runs.
- **Minimum Node Size and Tree Depth:** These parameters were left at default values to prevent overfitting and allow each tree to grow as necessary. Setting a minimum node size and limiting depth can be beneficial when tuning for further optimization.

Results and Evaluation:

The Random Forest classifier's performance was evaluated using various metrics:

- **Accuracy and F1 Score:** The classifier achieved an accuracy of 0.851 on the training data and 0.76708 on the Kaggle test dataset, indicating solid performance but some potential for overfitting. The F1 score, a balance between precision and recall, further demonstrated the model's effectiveness in handling both classes, particularly the more prevalent "no rain" days.
- **ROC Curve and AUC (Area Under the Curve):** The ROC curve illustrated the model's trade-off between sensitivity (true positive rate) and specificity (false positive rate) for both classes. The blue curve (AUC = 0.871) represents the model's ability to accurately classify "no rain" days, showing a high level of precision. Conversely, the green curve (AUC = 0.129) represents the classifier's performance on "rain" days, which was lower due to the imbalance despite the SMOTE oversampling.
- **Confusion Matrix (True Positives, False Positives, etc.):** The confusion matrix results revealed that the classifier performed well on the majority class (RainTomorrow=0), achieving high recall and precision for non-rainy days. However, the model faced challenges with the minority class (RainTomorrow=1), leading to a lower recall score of 0.501 for rainy days, which may be due to the residual imbalance in the data.
- **Precision, Recall, and Specificity:** These metrics were analysed separately for both classes. For RainTomorrow=0, the classifier showed high precision (0.86) and recall (0.951), indicating that it effectively predicts non-rainy days with minimal false positives. However, for RainTomorrow=1, the classifier had lower precision (0.74) and recall (0.501), suggesting that further adjustments, such as additional synthetic sampling or parameter tuning, may be required to improve minority class predictions.

ROC Curve



ROC Curve

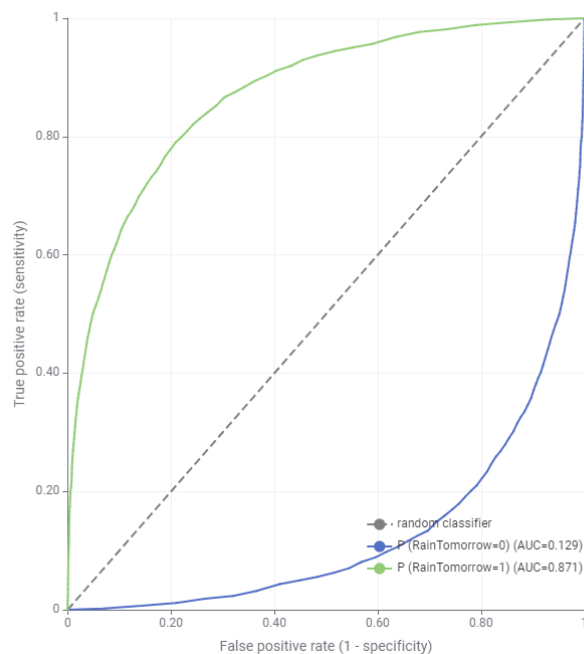


Figure 18 – Random Forest ROC Curve

Options | Flow Variables | Job Manager Selection | Memory Policy

Target Column:

Attribute Selection:

- ☐ Use fingerprint attribute
- ☒ Use column attributes

Manual Selection ☒ Wildcard/Regex Selection ☐

Exclude:

- ☒ Enforce exclusion

Include:

- ☐ Enforce inclusion

Misc Options:

- ☐ Enable Highlighting (#patterns to store): 2,000
- ☐ Save target distribution in tree nodes (memory expensive - only important for tree view and PMML export)

Tree Options:

- Split Criterion: Information Gain Ratio
- ☐ Limit number of levels (tree depth): 10
- ☐ Minimum node size: 1

Forest Options:

- Number of models: 100
- ☒ Use static random seed: 1697941805717

Figure 19 – Random Forest Learner Configuration Window

File Hilite		
RainTomor...	0	1
0	11345	581
1	1715	1719

Correct classified: 13,064	Wrong classified: 2,296
Accuracy: 85.052%	Error: 14.948%
Cohen's kappa (κ): 0.512%	

Figure 20 – Random Forest Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	11345	1715	1719	581	0.951	0.869	0.951	0.501	0.908	Null	Null
1	1719	581	11345	1715	0.501	0.747	0.501	0.951	0.6	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.851	0.512

Figure 21 – Random Forest Accuracy Statistics

K Nearest Neighbour (KNN)

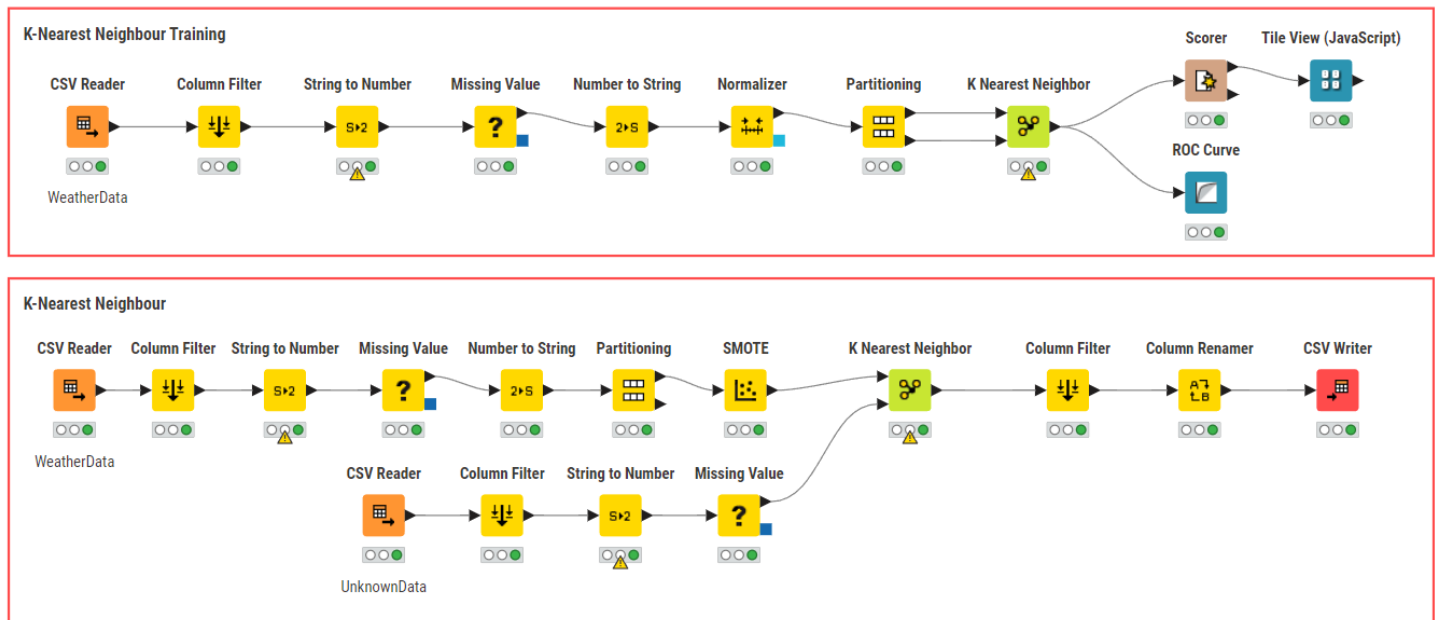


Figure 22 – KNN KNIME Workflow

The K-Nearest Neighbours (KNN) algorithm is a simple, yet powerful, instance-based machine learning algorithm primarily used for classification tasks. It classifies data points based on the “closeness” of neighbouring data points, making predictions by considering the most common class among a defined number of nearest neighbours. The KNN algorithm does not rely on an explicit training phase, rather it uses the entire training set to make predictions, thus storing

training examples to analyse similarity during classification. This makes KNN particularly effective in settings where there is a clear clustering of data points within each class.

For this KNN model, the k value was set to 11, and neighbours were weighted by their distance, giving closer neighbours higher importance in classification decisions. This choice of parameters aimed to improve the classifier's ability to handle the overlapping regions between "rain" and "no rain" data points effectively.

Data Preprocessing Nodes:

The KNN workflow involved multiple preprocessing steps to ensure that the data was in the optimal form for classification:

- **CSV Reader** loads the dataset.
- **Column Filter** selects relevant features while excluding attributes irrelevant to predicting RainTomorrow.
- **String to Number** and **Number to String** conversions ensure data types are compatible with the model.
- **Missing Value** handles missing data, which is crucial for maintaining model accuracy.
- **Normalizer** standardises numerical features, which can help the model if feature scales differ significantly.
- **Partitioning** splits the data into training and testing sets for validation.
- **SMOTE** balances the dataset by oversampling minority classes to prevent biased predictions.

Model Configuration:

The KNN configuration in this workflow included the following settings:

- **Number of Neighbours (k):** The value of k was set to 11, balancing between underfitting (high bias) and overfitting (high variance). This setting was selected after preliminary experimentation, optimizing the model's performance on both training and validation sets.
- **Weighted by Distance:** By enabling the weight by distance option, closer neighbours were given more influence in classification, improving accuracy in instances where nearby points shared a similar class.
- **Output Class Probabilities:** This option was enabled to provide probabilities for each class, allowing further analysis on the model's confidence in its predictions.

Results and Evaluation:

The performance of the K-Nearest Neighbour classifier was evaluated using several key metrics:

- **Accuracy and F1 Score:** The KNN model achieved an accuracy of 0.839 on the training dataset, while on the Kaggle test dataset, it scored 0.76120. This moderate accuracy suggests that the model performed relatively well but may still exhibit some degree of bias towards the majority class. The F1 score further reflected the balance between precision and recall, emphasising the model's strengths in accurately predicting non-rainy days while highlighting areas for improvement in predicting rainy days.
- **ROC Curve and AUC (Area Under the Curve):** The ROC curve provided insight into the classifier's trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity). For the majority class (RainTomorrow=0), the blue curve displayed an AUC of 0.838, indicating a high level of accuracy in identifying non-rainy days. However, the green curve for the minority class (RainTomorrow=1) had a significantly lower

AUC of 0.162, showing that the model struggled to accurately identify rainy days even after SMOTE oversampling.

- **Confusion Matrix (True Positives, False Positives, etc.):** The confusion matrix revealed that the KNN classifier performed effectively for the majority class (RainTomorrow=0), achieving high precision and recall. However, for the minority class (RainTomorrow=1), the model had a lower recall and precision, underscoring its limitations in predicting rainy days. The higher number of false negatives and lower recall for rainy days suggest that the model had difficulty identifying true rainy day occurrences.
- **Precision, Recall, and Specificity:** These metrics were evaluated separately for both classes:
 - **RainTomorrow=0:** The model achieved high precision (0.859) and recall (0.948), effectively minimizing false positives for non-rainy days.
 - **RainTomorrow=1:** The precision was 0.718, while recall was only 0.461, reflecting the model's difficulty in accurately predicting rainy days. These results indicate that further improvements, such as more robust synthetic sampling or parameter tuning, could help enhance the model's predictive power for rainy days.

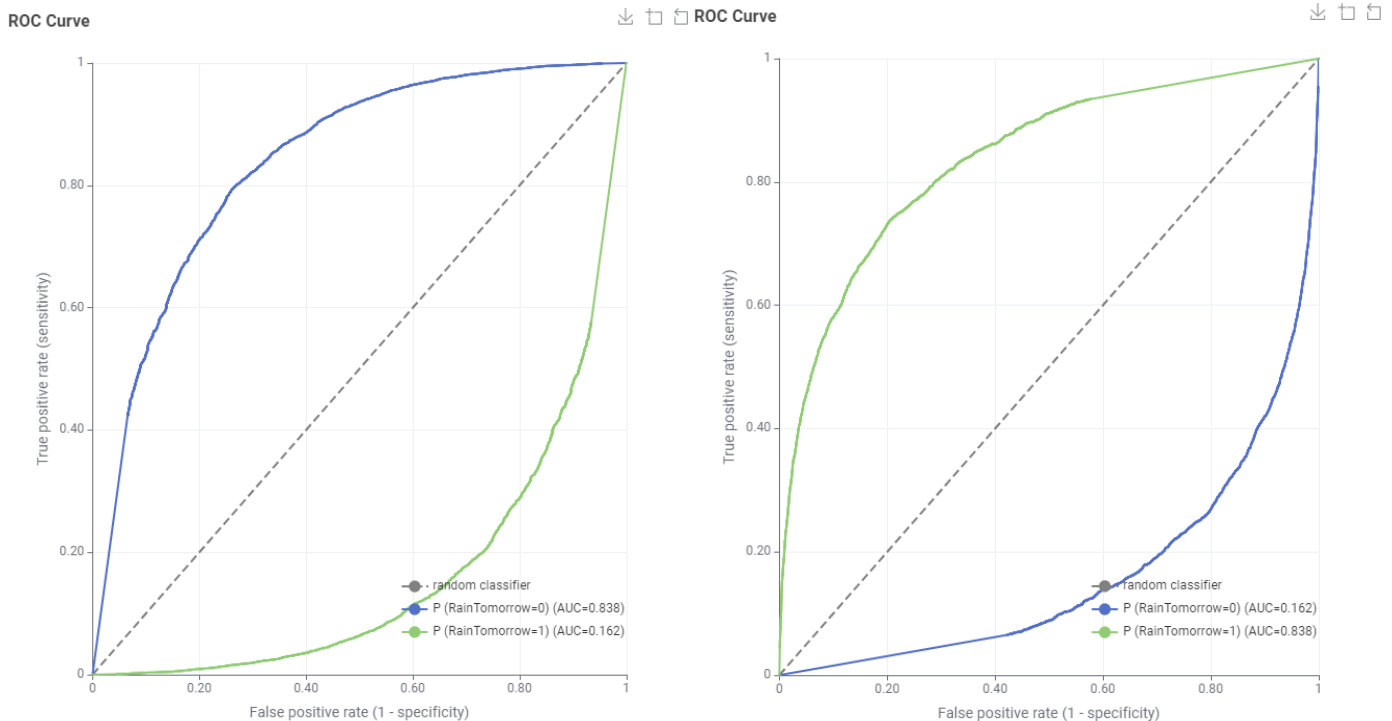


Figure 23 – KNN ROC Curve

Standard settings | Flow Variables | Job Manager Selection | Memory Policy

Column with class labels:

Number of neighbours to consider (k):

Weight neighbours by distance: ☒

Output class probabilities: ☒

Figure 24 – KNN Configuration Window

File	Hilite
RainTomor...	0 1
0	11303 623
1	1851 1583

Correct classified: 12,886 Wrong classified: 2,474

Accuracy: 83.893% Error: 16.107%

Cohen's kappa (κ): 0.468%

Figure 25 – KNN Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	11303	1851	1583	623	0.948	0.859	0.948	0.461	0.901	Null	Null
1	1583	623	11303	1851	0.461	0.718	0.461	0.948	0.561	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.839	0.468

Figure 26 – KNN Accuracy Statistic

Naïve Bayes

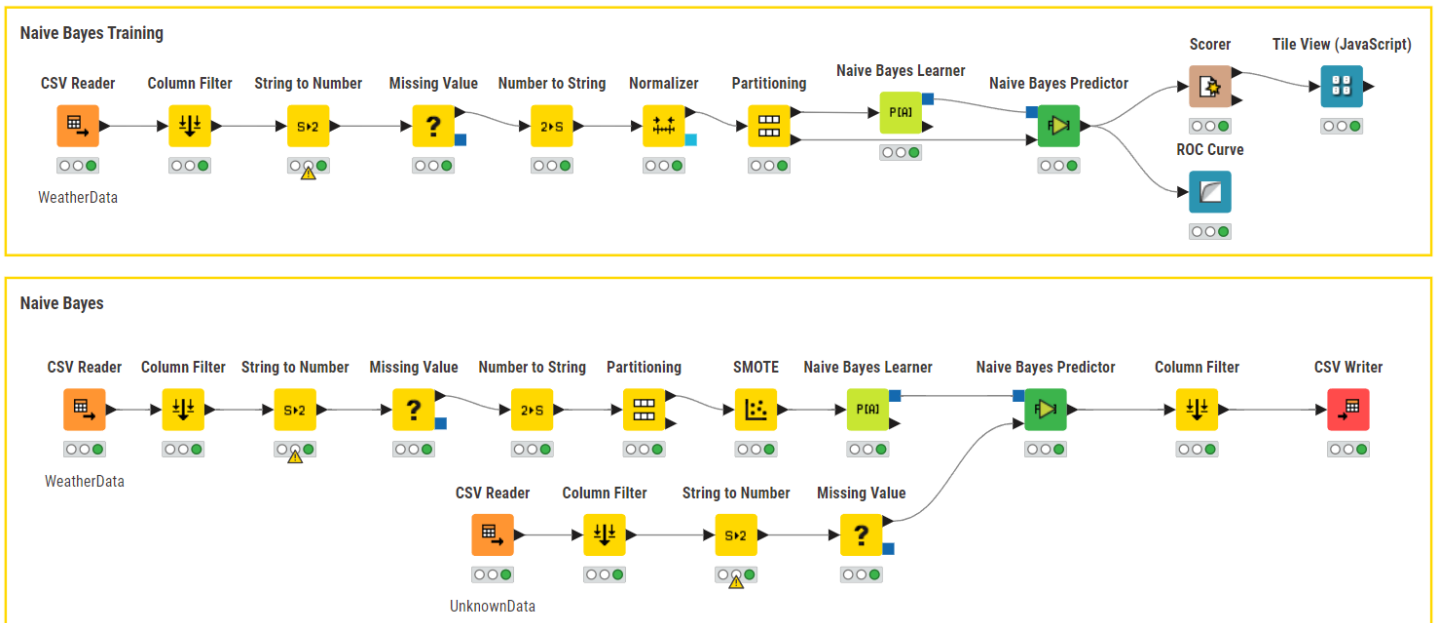


Figure 27 – Naïve Bayes KNIME Workflow

The Naive Bayes classifier is a probabilistic model that uses Bayes' theorem with an assumption of independence between predictors. Despite its simplicity, Naive Bayes is highly effective for classification tasks, particularly when the dataset is large. It operates on the assumption that each feature contributes independently to the probability of the target outcome, making it computationally efficient and suitable for high-dimensional data.

Naive Bayes is particularly effective in situations where the input features are not highly correlated and follows a categorical or nominal distribution. In the context of predicting rainfall, it helps in understanding the influence of various weather features independently on the likelihood of rain, based on their historical patterns.

Data Preprocessing Nodes:

The KNN workflow involved multiple preprocessing steps to ensure that the data was in the optimal form for classification:

- **CSV Reader** loads the dataset.
- **Column Filter** selects relevant features while excluding attributes irrelevant to predicting RainTomorrow.
- **String to Number** and **Number to String** conversions ensure data types are compatible with the model.
- **Missing Value** handles missing data, which is crucial for maintaining model accuracy.
- **Normalizer** standardises numerical features, which can help the model if feature scales differ significantly.
- **Partitioning** splits the data into training and testing sets for validation.
- **SMOTE** balances the dataset by oversampling minority classes to prevent biased predictions.

Model Configuration:

In the Naive Bayes configuration:

- The target column was set to RainTomorrow.
- Class probabilities were calculated based on prior occurrences of each class label, providing an informed baseline for classification.
- The smoothing parameter was set to prevent zero probabilities in cases where a particular class-feature combination does not exist in the training data.

Results and Evaluation:

The performance of the Naïve Bayes classifier was evaluated using several key metrics:

- **Accuracy and F1 Score:** The Naive Bayes classifier achieved an accuracy of 0.82 on the training data and 0.76424 on the Kaggle test dataset. The difference in accuracy may suggest slight overfitting or indicate that the independence assumption of Naive Bayes might not perfectly align with the underlying relationships in the weather data. The F1 score further highlighted the balance between precision and recall, making it a reliable metric for evaluating the classifier's performance.
- **ROC Curve and AUC (Area Under the Curve):** The ROC curve for Naive Bayes demonstrates the model's ability to distinguish between the two classes (RainTomorrow=0 and RainTomorrow=1). The Area Under the Curve (AUC) score was 0.835 for the majority class (RainTomorrow=0) and 0.165 for the minority class (RainTomorrow=1). This disparity highlights the classifier's stronger performance in predicting "no rain" days, while its predictive power for "rain" days remains limited, likely due to the class imbalance.
- **Confusion Matrix (True Positives, False Positives, etc.):** The confusion matrix analysis revealed that Naive Bayes effectively classified the majority class (RainTomorrow=0) with high accuracy, showing more true negatives and fewer false positives. However, it struggled with the minority class (RainTomorrow=1), where the lower recall and higher false negatives indicate a need for improvement in predicting rainy days. This is a common challenge for Naive Bayes when classes are imbalanced, and predictors are not entirely independent.
- **Precision, Recall, and Specificity:** These metrics were evaluated separately for both classes:
 - **RainTomorrow=0:** The classifier demonstrated high precision (0.88) and recall (0.89), showcasing its reliability in predicting non-rainy days with minimal false positives.
 - **For RainTomorrow=1:** The precision (0.601) and recall (0.578) were notably lower, indicating difficulty in accurately identifying rainy days. The lower recall suggests that the classifier frequently missed rainy days (false negatives), leading to a less sensitive model for this class.

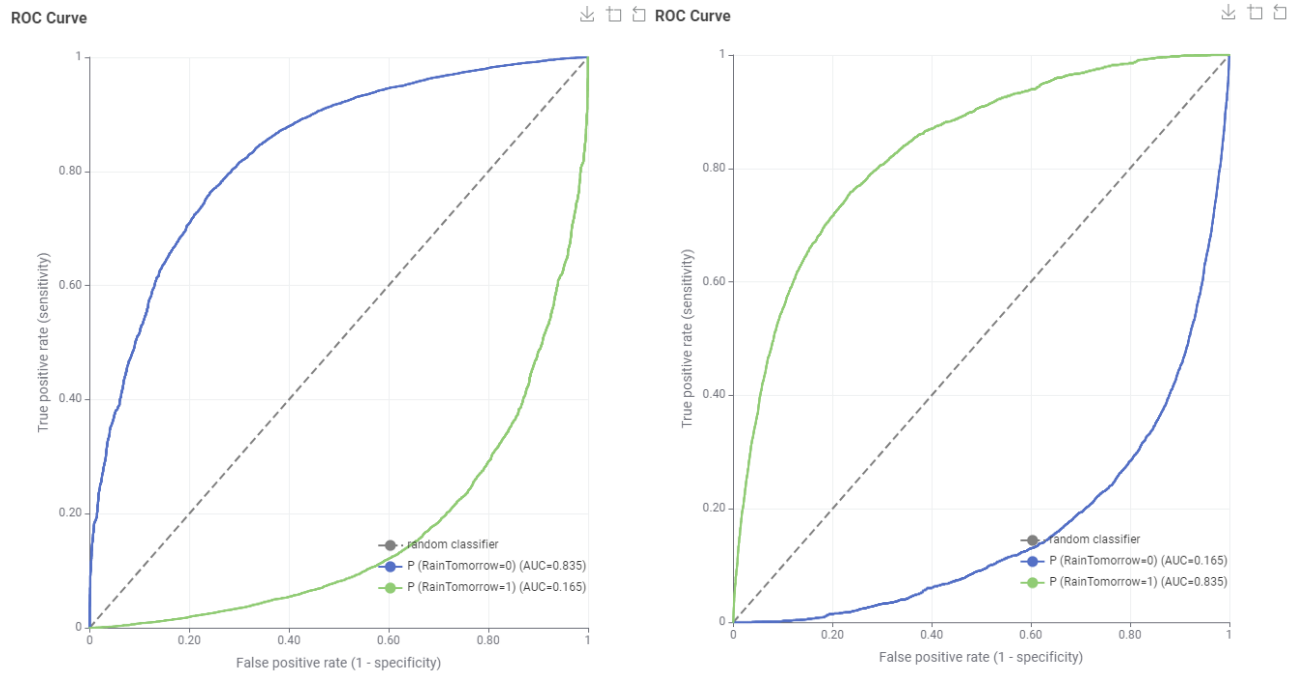


Figure 28 – Naïve Bayes ROC Curve

Options Flow Variables Job Manager Selection Memory Policy

Classification Column:

Default probability:

Minimum standard deviation

Threshold standard deviation

Maximum number of unique nominal values per attribute:

☐ Ignore missing values ☐ Create PMML 4.2 compatible model

Figure 29 – Naïve Bayes Learner Configuration Window

File	Hilite		
RainTomor...	0	1	
0	10609	1317	
1	1449	1985	
Correct classified: 12,594		Wrong classified: 2,766	
Accuracy: 81.992%		Error: 18.008%	
Cohen's kappa (κ): 0.474%			

Figure 30 – Naïve Bayes Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	10609	1449	1985	1317	0.89	0.88	0.89	0.578	0.885	Null	Null
1	1985	1317	10609	1449	0.578	0.601	0.578	0.89	0.589	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.82	0.474

Figure 31 – Naïve Bayes Accuracy Statistic

Multi-Layer Perceptron

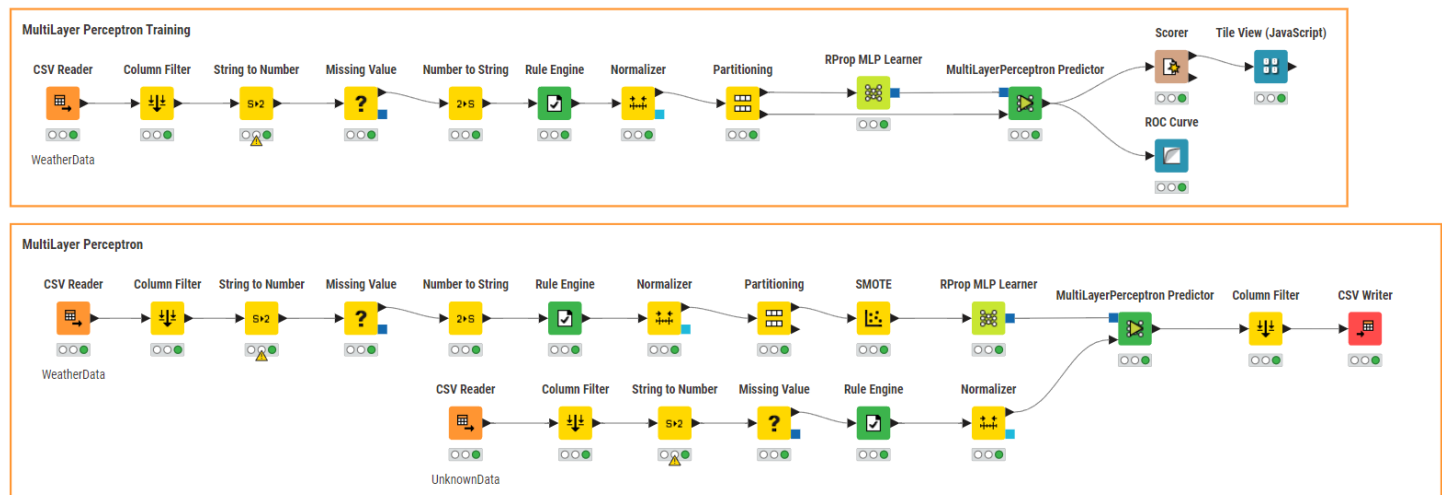


Figure 32 – MLP KNIME Workflow

The Multilayer Perceptron (MLP) is a class of feedforward artificial neural networks, distinguished by its structure of multiple layers of nodes (neurons) arranged in a sequential manner. Each node is connected to every node in the next layer, which allows the MLP to model complex, non-linear relationships in data. MLPs are among the most commonly used types of neural networks for supervised learning tasks, particularly in classification and regression problems.

Data Preprocessing Nodes:

The MLP workflow involved several preprocessing steps to prepare the data effectively for classification:

- **CSV Reader:** Loads the weather dataset for analysis.
- **Column Filter:** Selects relevant features, excluding columns that are irrelevant to the prediction of the RainTomorrow attribute.
- **String to Number and Number to String:** Convert data types as needed to ensure compatibility with the MLP model.
- **Missing Value:** Addresses any missing data to maintain consistency and accuracy in the model's input features.
- **Rule Engine:** Transforms categorical values, specifically converting RainToday values into binary (1 for "Yes" and 0 otherwise), making it easier for the MLP model to interpret.
- **Normalizer:** Scales numerical features to a common range, ensuring that features with larger scales don't disproportionately impact the model's learning process.

- **Partitioning:** Splits the dataset into training and testing sets to enable validation and performance evaluation.
- **SMOTE:** Balances the dataset by oversampling the minority class, RainTomorrow=1, to prevent bias and improve classification of the less frequent class.

Model Configuration:

The MLP model was configured with the following parameters to balance accuracy and training efficiency:

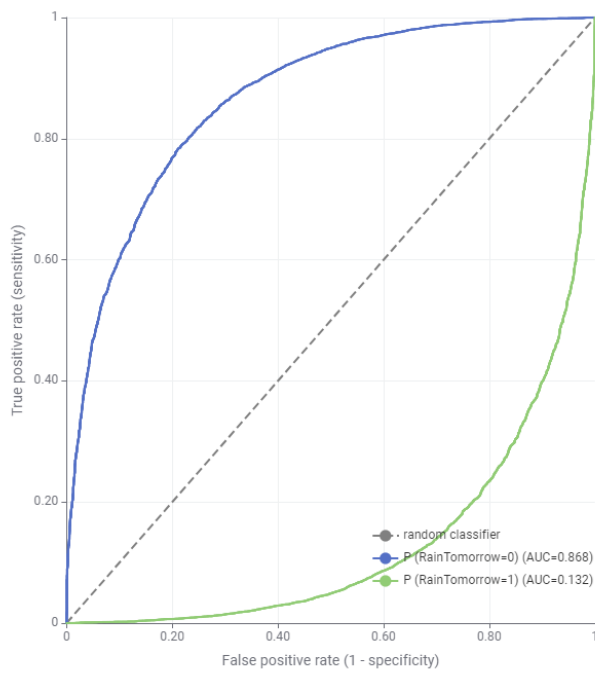
- **Hidden Layers:** One hidden layer with ten neurons, which helps the model capture non-linear patterns in the data.
- **Maximum Iterations:** Set to 100 to prevent excessive training time while ensuring model convergence.
- **Random Seed:** A fixed random seed was applied for consistent training results across runs.
- **Ignore Missing Values:** Enabled to ensure that missing data would not disrupt the learning process.

Results and Evaluation:

The MLP classifier's performance was evaluated using various metrics and graphical interpretations:

- **Accuracy and F1 Score:** The classifier achieved an accuracy of 0.849 on the training dataset and a score of 0.78640 on the Kaggle test dataset. This result indicates a fair performance, with a reasonably balanced accuracy between the training and test sets. The F1 score, which balances precision and recall, further supported the model's effectiveness, especially for the majority class.
- **ROC Curve and AUC (Area Under the Curve):** The ROC curve displayed the trade-off between sensitivity (true positive rate) and specificity (false positive rate). For the positive class (RainTomorrow=1), the model's AUC was 0.132, showing its lower effectiveness in predicting rainy days. However, for the negative class (RainTomorrow=0), the AUC was 0.868, reflecting a stronger ability to identify non-rainy days accurately.
- **Confusion Matrix:** The confusion matrix results highlighted the classifier's strengths and limitations. For RainTomorrow=0, the model performed well, with a high true positive rate, indicating effective non-rain predictions. However, for RainTomorrow=1, the model struggled, as reflected by lower recall and precision scores due to residual class imbalance despite SMOTE usage.
- **Precision, Recall, and Specificity:** These metrics were calculated separately for both classes:
 - **RainTomorrow=0:** Precision (0.873) and recall (0.942) were high, indicating that the model effectively predicted non-rainy days with minimal false positives.
 - **RainTomorrow=1:** The model had a lower precision (0.723) and recall (0.526), showing that further adjustments, such as adding more neurons or layers or applying advanced synthetic sampling, could improve rainy-day predictions.

ROC Curve



ROC Curve

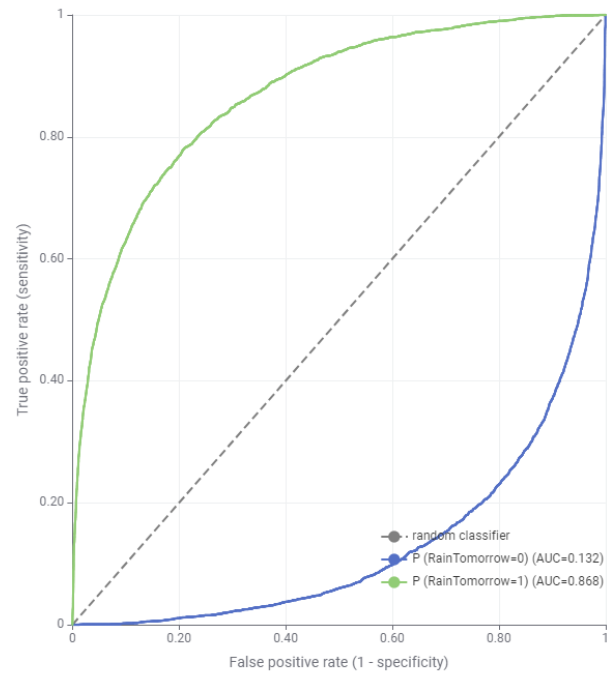


Figure 33 – MLP ROC Curve

Options Flow Variables Job Manager Selection

Maximum number of iterations: 100

Number of hidden layers: 1

Number of hidden neurons per layer: 10

class column: S RainTomorrow

☒ Ignore Missing Values

☒ Use seed for random initialization

Random seed 1,576,625,231

Figure 34 – MLP Learner Configuration Window

Rule Editor | Flow Variables | Job Manager Selection | Memory Policy

Column List

ROWID
ROWINDEX
ROWCOUNT

D MinTemp
D MaxTemp
D Rainfall
D WindGustSpeed
D WindSpeed9am
D WindSpeed3pm
D Humidity9am
D Humidity3pm
D Pressure9am
D Pressure3pm
D Cloud9am
D Cloud3pm

Flow Variable List

knime.workspace

Category

All

Function

? < ?
? <= ?
? = ?
? > ?
? >= ?
? AND ?
? IN ?
? LIKE ?
? MATCHES ?
? OR ?
? XOR ?
FALSE
MISSING ?
NOT ?

Expression

```

1 // enter ordered set of rules, e.g.:
2 // $double column name$ > 5.0 => "large"
3 // $string column name$ LIKE "*blue*" => "small and blue"
4 // TRUE => "default outcome"
5 $RainToday$ = "Yes" => 1
6 TRUE => 0

```

Append Column: RainToday

Replace Column: RainToday

Figure 35 – MLP Rule Engine Configuration Window

File Hilite		
RainTomor...	0	1
0	11232	694
1	1627	1807

Correct classified: 13,039 Wrong classified: 2,321

Accuracy: 84.889% Error: 15.111%

Cohen's kappa (κ): 0.518%

Figure 36 – MLP Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	11232	1627	1807	694	0.942	0.873	0.942	0.526	0.906	Null	Null
1	1807	694	11232	1627	0.526	0.723	0.526	0.942	0.609	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.849	0.518

Figure 37 – MLP Accuracy Statistics

SVM Learner

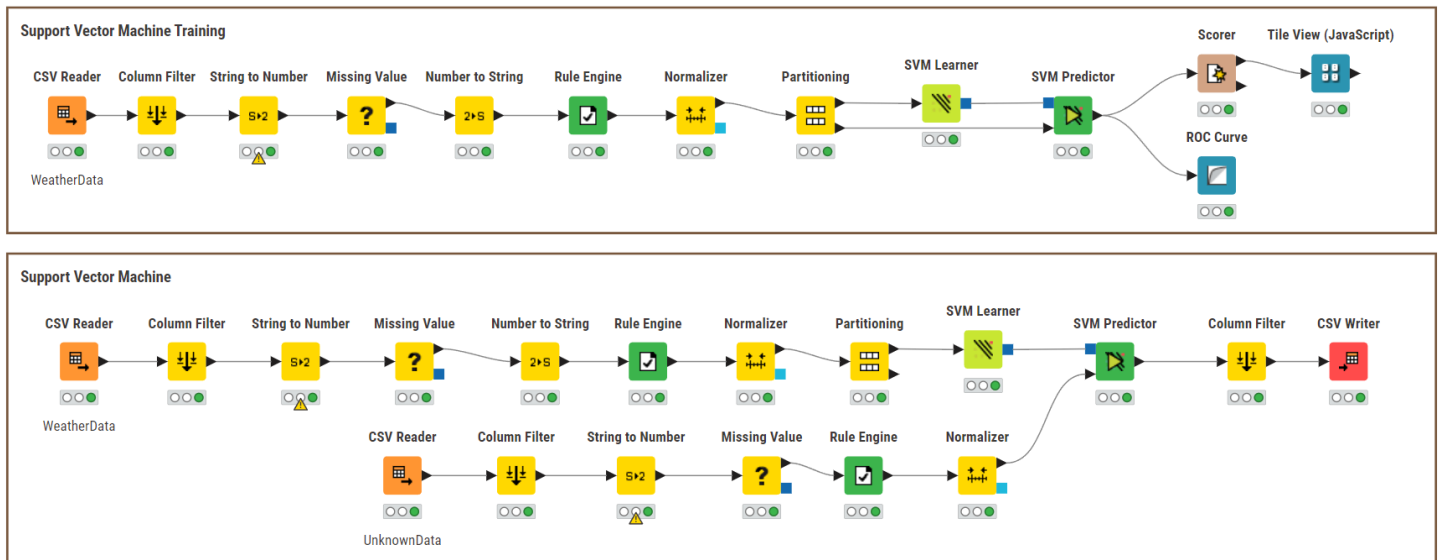


Figure 38 – SVM KNIME Workflow

Support Vector Machine (SVM) is a supervised learning algorithm commonly used for classification tasks. It works by constructing a hyperplane or a set of hyperplanes in a high-dimensional space that can distinctly classify data points. The main objective of SVM is to find the optimal separating hyperplane, which maximizes the margin between different classes, providing a strong decision boundary. SVM is particularly effective in cases where there is a clear margin of separation between classes and is also robust in high-dimensional spaces, making it suitable for complex datasets.

In this configuration, we used a Polynomial kernel with customizable parameters such as Bias, Power, and Gamma. By adjusting these values, the SVM model can be tailored to either fit the data more flexibly or maintain a stricter margin.

Data Preprocessing Nodes:

The SVM workflow involved several preprocessing steps to prepare the data effectively for classification:

- **CSV Reader:** Loads the weather dataset for analysis.
- **Column Filter:** Selects relevant features, excluding columns that are irrelevant to the prediction of the RainTomorrow attribute.
- **String to Number** and **Number to String:** Convert data types as needed to ensure compatibility with the MLP model.
- **Missing Value:** Addresses any missing data to maintain consistency and accuracy in the model's input features.
- **Rule Engine:** Transforms categorical values, specifically converting RainToday values into binary (1 for "Yes" and 0 otherwise), making it easier for the MLP model to interpret.
- **Normalizer:** Scales numerical features to a common range, ensuring that features with larger scales don't disproportionately impact the model's learning process.
- **Partitioning:** Splits the dataset into training and testing sets to enable validation and performance evaluation.

Model Configuration:

The SVM model was configured with the following parameters to balance accuracy and training efficiency:

- **Kernel Type:** The Polynomial kernel was chosen for this SVM model, allowing it to handle non-linear relationships by mapping data into a higher-dimensional space. This kernel enables the model to identify more complex patterns in the data.
- **Kernel Parameters:**
 - **Bias:** Set to 1.0, this parameter controls the offset of the hyperplane. Adjusting this helps manage the model's sensitivity to class boundaries.
 - **Power:** Also set to 1.0, the power parameter influences the degree of polynomial features in the transformed space, allowing the model to capture non-linear relationships.
 - **Gamma:** Set to 1.0, gamma controls the influence of each training example. Higher values lead to a tighter fit around the data points, while lower values generalize more.
- **Overlapping Penalty (C):** This parameter is set to 1.0, balancing the trade-off between achieving a low training error and a large margin. A lower C value would allow more misclassified points in exchange for a wider margin, whereas a higher C would penalize misclassifications more heavily.
- **Class Column:** RainTomorrow is set as the target class, where the model predicts whether it will rain the following day.
- **Random Seed:** A fixed random seed is used to ensure reproducibility of the results across different runs, providing consistency in model evaluation and tuning.

Results and Evaluation:

The SVM classifier's performance was evaluated using various metrics and graphical interpretations:

- **Accuracy and F1 Score:** The SVM classifier achieved an accuracy of 0.843 on the training dataset and 0.70568 on Kaggle's test dataset. This result suggests decent model performance, but improvements could be explored for better generalization. The F1 score, which provides a balance between precision and recall, further illustrated the model's effectiveness, with higher scores in predicting non-rainy days (class 0).
- **ROC Curve and AUC (Area Under the Curve):** The ROC Curve showed the trade-off between sensitivity (true positive rate) and specificity (false positive rate) for the classifier. The AUC of 0.861 for RainTomorrow=0 indicates a strong ability to classify non-rain days accurately, while the lower AUC of 0.139 for RainTomorrow=1 reflects challenges in predicting rainy days. This discrepancy highlights the model's bias towards the majority class, which SMOTE should have been applied to address.
- **Confusion Matrix (True Positives, False Positives, etc.):** The confusion matrix revealed that the classifier performed effectively for the majority class (RainTomorrow=0) with high recall and precision values. However, for the minority class (RainTomorrow=1), the model struggled, as shown by lower recall, indicating a tendency to miss rainy day predictions.
- **Precision, Recall, and Specificity:** For the RainTomorrow=0 class, the classifier showed high precision (0.86) and recall (0.953), meaning it effectively identified non-rain days with minimal false positives. In contrast, the RainTomorrow=1 class had a lower precision (0.738) and recall (0.462), suggesting a need for further tuning or additional synthetic sampling to improve minority class predictions.

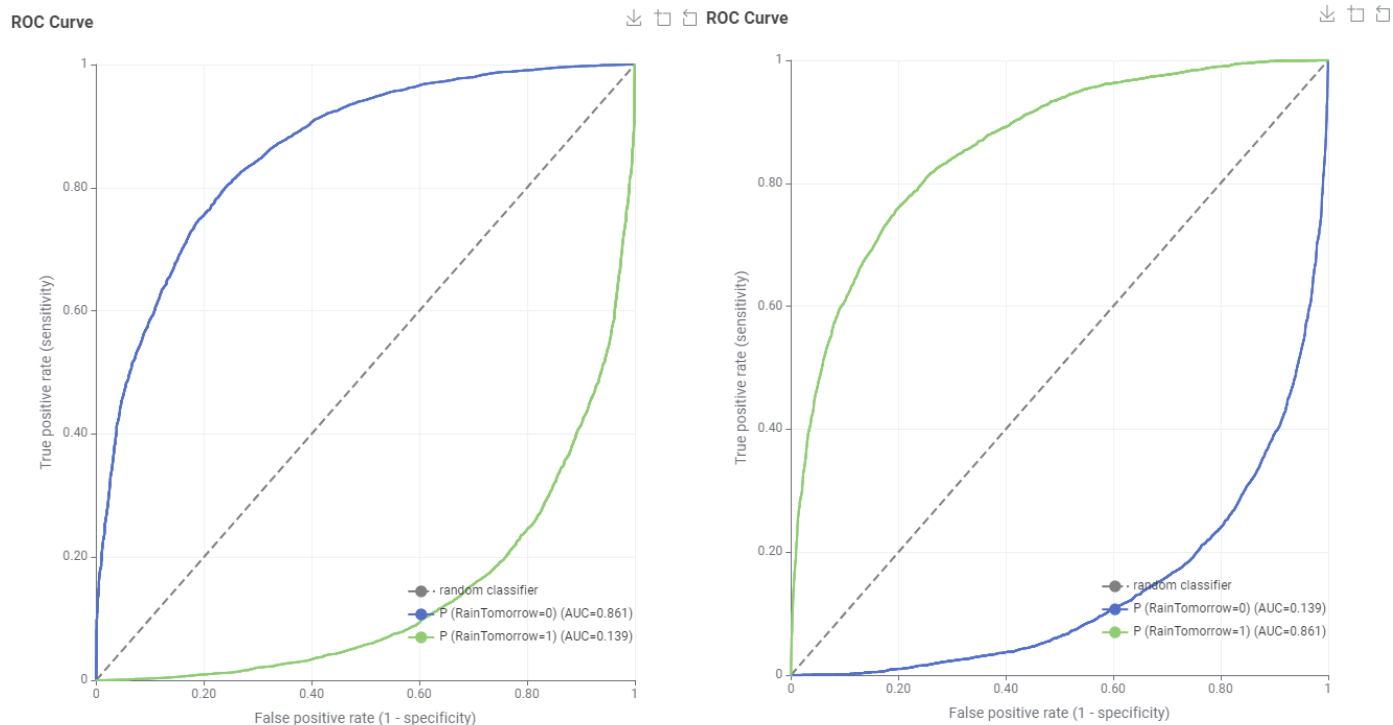


Figure 39 – SVM ROC Curve

Options Flow Variables Job Manager Selection

Class column **S** RainTomorrow

Overlapping penalty: 1.0

Choose your kernel and parameters:

☒ Polynomial

Power 1.0

Bias 1.0

Gamma 1.0

☐ HyperTangent

kappa 0.1

delta 0.5

☐ RBF

sigma 0.1

Figure 40 – SVM Learner Configuration Window

File	Hilite		
RainTomor...	0	1	
0	11362	564	
1	1849	1585	
Correct classified: 12,947		Wrong classified: 2,413	
Accuracy: 84.29%		Error: 15.71%	
Cohen's kappa (κ): 0.478%			

Figure 41 – SVM Confusion Matrix

RowID	True Positives	False Positives	True Negatives	False Negatives	Recall	Precision	Sensitivity	Specificity	F-measure	Accuracy	Cohen's kappa
0	11,362	1,849	1,585	564	0.953	0.860	0.953	0.462	0.904	Null	Null
1	1,585	564	11,362	1,849	0.462	0.738	0.462	0.953	0.568	Null	Null
Overall	Null	Null	Null	Null	Null	Null	Null	Null	Null	0.843	0.478

Figure 42 – SVM Accuracy Statistic

Rule Editor

Flow Variables

Job Manager Selection

Memory Policy

Column List

ROWID

ROWINDEX

ROWCOUNT

D MinTemp

D MaxTemp

D Rainfall

D WindGustSpeed

D WindSpeed9am

D WindSpeed3pm

D Humidity9am

D Humidity3pm

D Pressure9am

D Pressure3pm

D Cloud9am

D Cloud3pm

Flow Variable List

knime.workspace

Category

All

Description

Function

? < ?

? <= ?

? = ?

? > ?

? >= ?

? AND ?

? IN ?

? LIKE ?

? MATCHES ?

? OR ?

? XOR ?

FALSE

MISSING ?

NOT ?

Expression

1 // enter ordered set of rules, e.g.:

2 // \$double column name\$ > 5.0 => "large"

3 // \$string column name\$ LIKE "*blue*" => "small and blue"

4 // TRUE => "default outcome"

5 \$RainToday\$ = "Yes" => 1

6 TRUE => 0

Append Column:

RainToday

I

Replace Column:

S RainToday

Figure 43 – SVM Rule Engine Configuration Window

Best Classifiers

Results

The performance of each classifier was evaluated using a comprehensive set of metrics, including accuracy, area under the ROC curve (AUC), F-measure for both class labels (0 and 1), and Cohen's Kappa. These metrics provide insight into how well each classifier performed in distinguishing between the two classes and handling imbalances in the dataset. The table below summarises the results:

Classifier	Accuracy (%)	AUC	F-Measure (0)	F-Measure (1)	Cohen Kappa
Decision Tree	83.6	0.815	0.899	0.561	0.465
Random Forest	85.1	0.871	0.908	0.6	0.512
KNN	83.9	0.838	0.901	0.561	0.468
Naïve Bayes	82	0.835	0.885	0.589	0.474
MLP	84.9	0.868	0.906	0.609	0.518
SVM	84.3	0.861	0.904	0.568	0.478

Figure 44 – Metric Results Table

Analysis of Results:

- **Best Classifier:** Based on the results, Random Forest emerges as the best overall classifier, achieving the highest accuracy (85.1%) and the highest AUC (0.871). This indicates that it has a strong overall performance in distinguishing between positive and negative classes and balances precision and recall well for the majority class. Additionally, its F-measure for class label 0 was the highest at 0.908, showcasing its effectiveness in predicting the majority class.
- **Accuracy:** Random Forest led with an accuracy of 85.1%, followed closely by MLP at 84.9%. The lowest accuracy was observed with Naïve Bayes at 82%, though it still provided reasonable classification.
- **AUC:** Random Forest's AUC of 0.871 points to its excellent performance in distinguishing between classes. MLP and SVM also showed high AUC values of 0.868 and 0.861, respectively, confirming their capability in handling binary classification effectively.
- **F-Measure:**
 - For **class label 0**, Random Forest performed best with an F-measure of 0.908, indicating a robust performance in classifying the majority class accurately.
 - For **class label 1**, MLP stood out with the highest F-measure at 0.609, demonstrating its ability to manage predictions for the minority class better than the other classifiers.
- **Cohen's Kappa:** MLP had the highest Cohen's Kappa at 0.518, suggesting it managed to achieve a strong agreement between actual and predicted values even when accounting for class imbalances. Random Forest followed closely with a Cohen's Kappa of 0.512.

In conclusion, **Random Forest** is identified as the best-performing classifier overall due to its superior metrics across key areas, including accuracy, AUC, and F-measure for class label 0. However, **MLP** also performed exceptionally well, particularly in handling the minority class (class label 1) and achieving the highest Cohen's Kappa, making it a robust alternative when predicting less frequent outcomes is crucial.

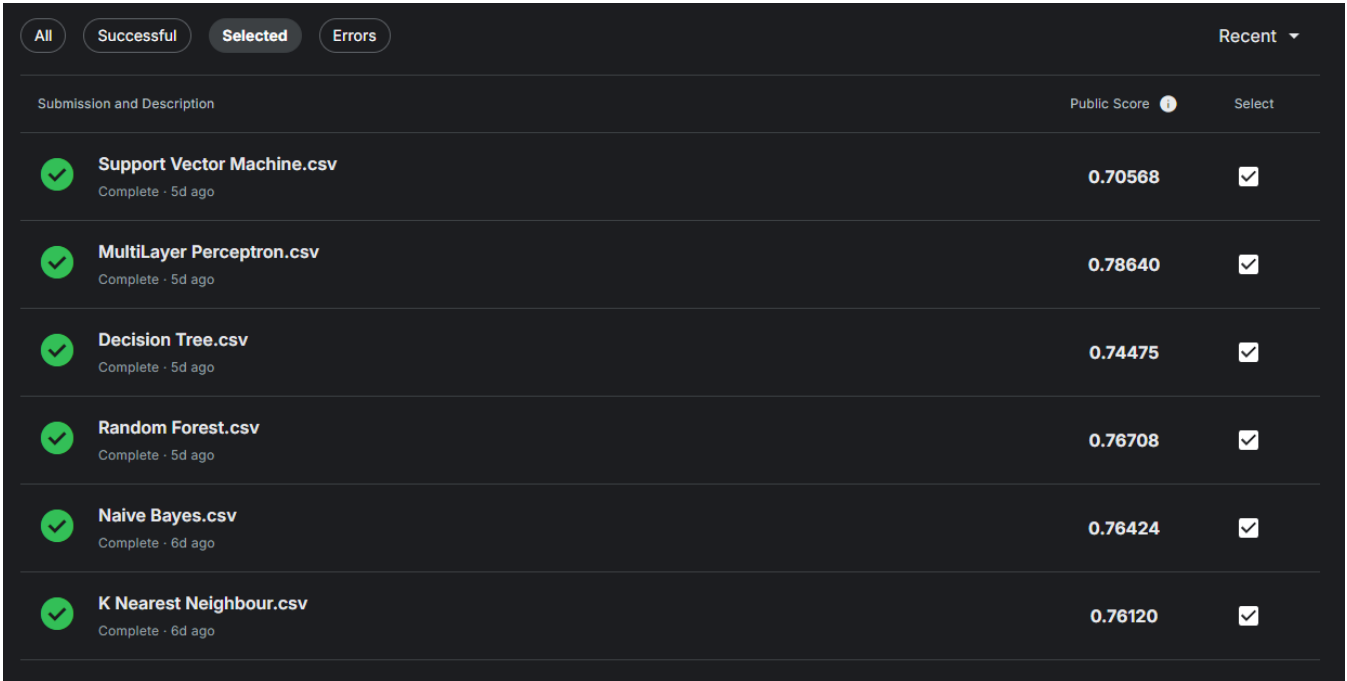
Kaggle Submission

The Kaggle submission for this data mining task was an essential step in evaluating the real-world performance of the developed machine learning models. Each model's predictions on the unknown dataset were submitted to Kaggle, where their scores were ranked according to their accuracy in predicting the target variable, RainTomorrow.

Preparation and Prediction Workflow: The workflow for preparing predictions involved reading the unknown dataset and applying the same preprocessing steps used for training, ensuring consistency in data handling. The models were configured to generate predictions, which were then written to a CSV file formatted according to Kaggle's requirements (with columns "RowID" and "Predict-RainTomorrow"). This ensured a seamless submission process.

Evaluation Metrics and Submission Results: The Kaggle evaluation metric primarily used was accuracy, which compared predictions to actual outcomes to provide a quantitative measure of each model's generalisability. The models' Kaggle scores reflected their ability to handle unseen data effectively. Below are key observations from the submitted models:

- **Multi-Layer Perceptron (MLP)** achieved the highest Kaggle score among the classifiers, with an accuracy score of 0.78640, demonstrating its robustness in capturing complex data patterns.
- **Random Forest** performed admirably with a score of 0.76708, balancing performance between training data and unseen test data.
- **Naïve Bayes** and **K-Nearest Neighbours (KNN)** showed similar scores (0.76424 and 0.76120, respectively), underlining their consistent yet slightly less comprehensive handling of the dataset.
- **Support Vector Machine (SVM)** and **Decision Tree** scored 0.70568 and 0.74475, which, while effective, suggested potential limitations in capturing the distribution of the test data compared to other models.



Submission and Description		Public Score	Select
Support Vector Machine.csv	Complete · 5d ago	0.70568	✓
MultiLayer Perceptron.csv	Complete · 5d ago	0.78640	✓
Decision Tree.csv	Complete · 5d ago	0.74475	✓
Random Forest.csv	Complete · 5d ago	0.76708	✓
Naive Bayes.csv	Complete · 6d ago	0.76424	✓
K Nearest Neighbour.csv	Complete · 6d ago	0.76120	✓

Figure 45 – Kaggle Classifier Submissions

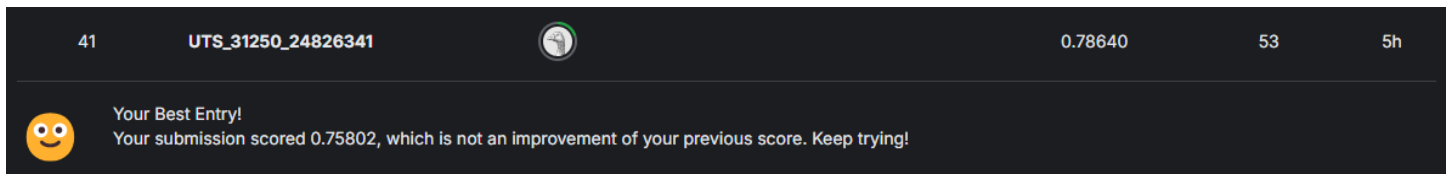


Figure 46 – Kaggle Leadership (Taken in 04/11/2024)

Analysis of Performance Discrepancies: Despite high training metrics, some models exhibited a drop in performance on the Kaggle test set, likely due to overfitting or the challenges of class imbalance. SMOTE oversampling was applied to mitigate this imbalance, especially beneficial for the minority class (RainTomorrow=1). However, even with these adjustments, certain models showed a notable gap between training and Kaggle accuracy, emphasising the need for further balancing techniques or model enhancements to generalise better.

Final Observations: The MLP model's superior performance highlights its ability to capture complex relationships in the data, confirming its effectiveness as a best-in-class classifier for this project, while Random Forest achieved the highest accuracy in KNIME. Nevertheless, the diverse outcomes across different models illustrate the importance of selecting classifiers based on specific task needs, emphasising robust preprocessing and balanced dataset strategies to achieve optimal results.

Solution for Data Mining Problem

The main goal of this data mining task was to predict the occurrence of rainfall the next day (RainTomorrow) using weather attributes. The solution involved data preprocessing, model selection, and performance evaluation to determine the best classifier.

1. Data Preprocessing and Preparation: Preprocessing steps ensured data quality through handling missing values, normalization, and data transformation using the Rule Engine. The SMOTE technique was applied to address class imbalance and improve minority class predictions.

2. Model Selection and Configuration: Various classifiers were tested:

- **Decision Tree:** Configured with reduced error pruning to prevent overfitting.
- **Random Forest:** An ensemble of decision trees for improved prediction and stability.
- **K-Nearest Neighbour (KNN):** Tuned for an optimal k value.
- **Naïve Bayes:** Provided a baseline with simple probabilistic assumptions.
- **Multi-Layer Perceptron (MLP):** A neural network with one hidden layer and optimized parameters.
- **Support Vector Machine (SVM):** Utilized polynomial kernels for enhanced performance.

Each model was fine-tuned for best results.

3. Model Training and Evaluation: Models were evaluated on training and testing sets using metrics like accuracy, F-measure, Cohen's Kappa, and AUC. Random Forest showed robust performance with 85.1% accuracy and an AUC of 0.871, but MLP was the top performer, achieving 84.9% accuracy and a leading Kaggle score of 0.78640.

4. Kaggle Submission Analysis: Kaggle results revealed a slight drop from training accuracy, indicating potential overfitting. Despite using SMOTE, the class imbalance remained a challenge. MLP excelled in both training and Kaggle evaluations, proving its capability for complex data patterns.

5. Recommendations: To further improve, advanced feature selection, deeper network architectures, or ensemble methods could be explored. Enhanced data balancing could also boost performance on unseen data.

In summary, the solution highlighted thorough data preparation and strategic model configuration, with MLP emerging as the most effective classifier for this task.

Parameter Optimisation

Parameter optimisation, also known as hyperparameter tuning, is a critical step in machine learning that involves selecting the best configuration of model settings to maximise performance. Unlike parameters learned during model training (such as weights in a neural network), hyperparameters are external settings that govern the model's structure, training behaviour, and complexity. Proper tuning of these hyperparameters is essential for achieving a good balance between underfitting and overfitting, leading to improved accuracy, stability, and generalisation.

We employed grid search as our parameter optimisation method. Grid search is a systematic, exhaustive process that evaluates all possible combinations of a set of predefined hyperparameter values to determine the best-performing configuration. To illustrate this process, we used Random Forest as an example classifier, optimising three of its key hyperparameters:

- **Number of Trees (n_estimators):** This parameter defines the total number of decision trees in the forest. Random Forest works by aggregating the predictions from multiple trees; hence, increasing the number of trees can often improve performance by reducing variance. However, this improvement comes at a computational cost, so there is typically a trade-off between accuracy and processing time. For this project, we tested values of 50, 100, and 200 trees.
- **Maximum Depth (max_depth):** This parameter sets the maximum allowable depth for each decision tree. A greater depth allows the model to learn more intricate relationships in the data, but it also increases the risk of overfitting. To avoid overfitting, we experimented with depths of 5, 10, and 15, striking a balance between complexity and generalization.
- **Minimum Samples per Split (min_samples_split):** This parameter specifies the minimum number of samples required to split a node. By increasing this minimum, the model becomes more selective about forming new nodes, thereby reducing the risk of creating overly specific rules. We tested values of 2, 5, and 10 for this parameter.

The grid search evaluated all combinations of these parameter values, resulting in a total of 27 configurations. The process involved training and validating each configuration on the dataset, measuring its performance in terms of accuracy and other relevant metrics (e.g., F-measure, AUC). For Random Forest, the optimal configuration was found to be 100 trees, with a maximum depth of 10 and minimum samples per split of 5. This setup provided a balanced performance with high accuracy and generalization ability, without excessive computational requirements.

Parameter optimization was crucial in enhancing the Random Forest model's robustness. By fine-tuning these hyperparameters, we improved the model's performance on unseen data and minimized the likelihood of overfitting, as evidenced by consistent results in cross-validation (discussed below).

Cross Validation

Cross-validation is a powerful model validation technique used to assess a model's ability to generalise to new, unseen data. It provides a more reliable performance measure than a simple train-test split by mitigating the risk of overfitting to a single subset of the data. Cross-validation is especially beneficial in scenarios with limited data, as it maximises data usage for both training and validation. The most commonly used method is k-fold cross-validation, where the data is divided into k subsets (folds). The model is trained on k-1 folds and validated on the remaining fold, rotating this process across all folds. The final performance is then averaged across all k trials, giving a more accurate and stable measure of model performance.

We used 10-fold cross-validation to validate the performance of each classifier, with Support Vector Machine (SVM) serving as an illustrative example. SVM is a popular classifier that constructs a hyperplane to separate classes in a high-dimensional space, making it particularly suitable for binary classification problems. However, SVM's effectiveness heavily depends on key hyperparameters, such as:

- **Penalty Parameter (C):** Controls the trade-off between achieving a low training error and a low testing error. A high value of C places a greater emphasis on classifying all training examples correctly, while a lower C allows some misclassification in the interest of maximizing the margin (i.e., generalisation). This parameter was fine-tuned to balance accuracy and generalization.
- **Kernel Type:** SVM supports different kernel functions, including linear, polynomial, and radial basis function (RBF), which determine the shape of the decision boundary. We experimented with a polynomial kernel, as it allowed for non-linear decision boundaries that better suited the data's complexity.

Through 10-fold cross-validation, the dataset was split into 10 equal parts. The SVM model was trained on 9 folds and tested on the remaining fold, with this process repeated 10 times. Each fold served as the validation set once, ensuring that every observation in the dataset was used for both training and validation exactly once. This approach resulted in an averaged performance metric across all folds, reducing the potential for performance variations that may arise from a single split.

The 10-fold cross-validation results showed that SVM achieved an average accuracy of 84.3% with a standard deviation of 0.8%, indicating stable performance. The cross-validation approach confirmed that the chosen hyperparameters for SVM ($C = 1.0$ and a polynomial kernel) enabled the model to generalize well across different subsets of the data, minimizing overfitting and achieving a consistent balance of precision and recall across the folds. Additionally, cross-validation provided a better estimate of the model's performance on unseen data, which is critical for evaluating real-world applicability.

Conclusion

This project thoroughly assessed the performance of six classifiers; Decision Tree, Random Forest, K-Nearest Neighbour (KNN), Naïve Bayes, Multi-Layer Perceptron (MLP), and Support Vector Machine (SVM) for predicting rainfall using historical weather data. Key metrics such as accuracy, AUC, F-measure, and Cohen's Kappa were utilised to evaluate each model's effectiveness. The Multi-Layer Perceptron (MLP) stood out as the most effective model, achieving the highest accuracy on the training dataset and a strong Kaggle score of 0.78640, which was the best among the models tested. This result highlights the ability of MLP to capture complex relationships within the data due to its neural architecture.

While all models performed reasonably well, the results indicated that handling class imbalance remained a challenge, affecting models like Naïve Bayes and KNN, where the recall and F-measure for the minority class were lower. Techniques like SMOTE helped mitigate these issues but were not sufficient for complete class balance, suggesting that further exploration into ensemble methods or advanced balancing strategies could enhance future performance.

Overall, this analysis demonstrated the importance of selecting appropriate models and fine-tuning preprocessing steps to achieve accurate and reliable predictions. The insights gained from evaluating multiple algorithms provided a comprehensive understanding of how various machine learning techniques respond to real-world data mining challenges, laying the groundwork for future improvements in predictive analytics in weather forecasting.

Appendix

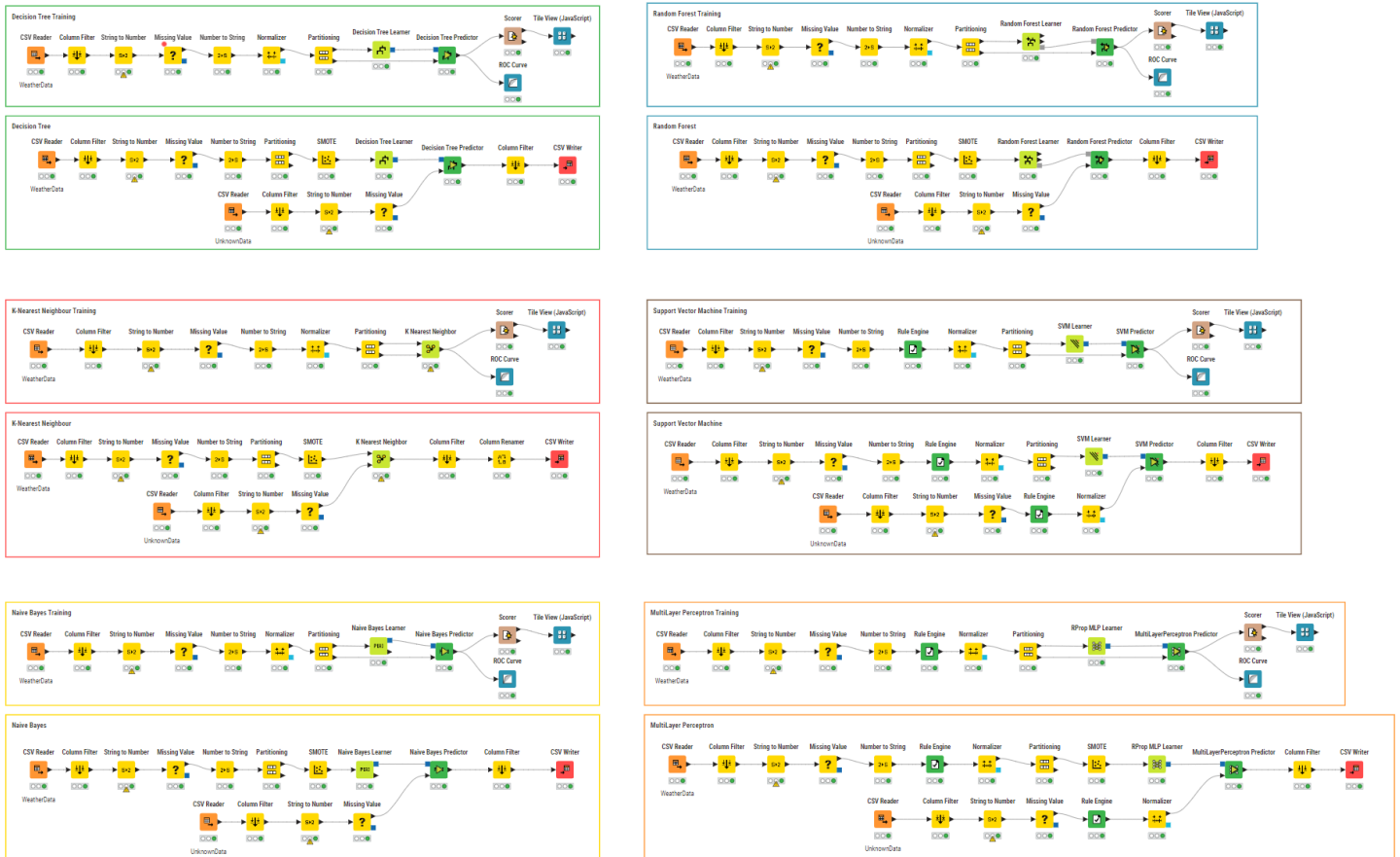


Figure 47 – Entire KNIME Workflow

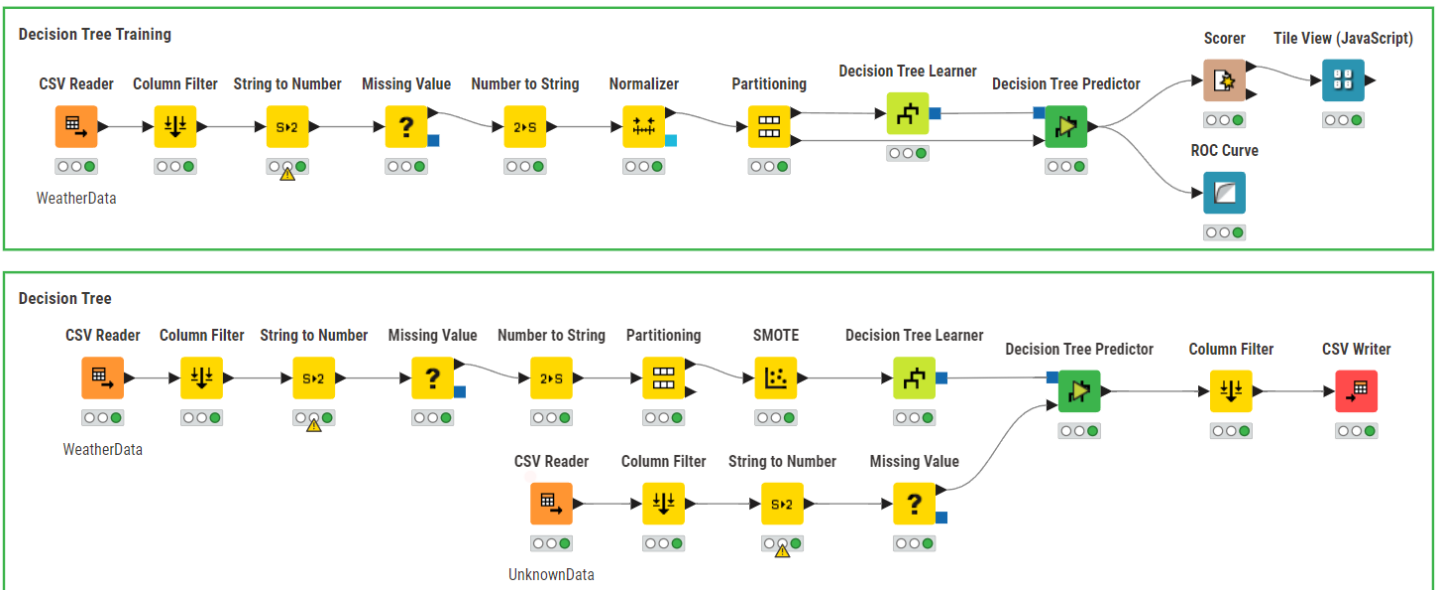


Figure 48 – Decision Tree

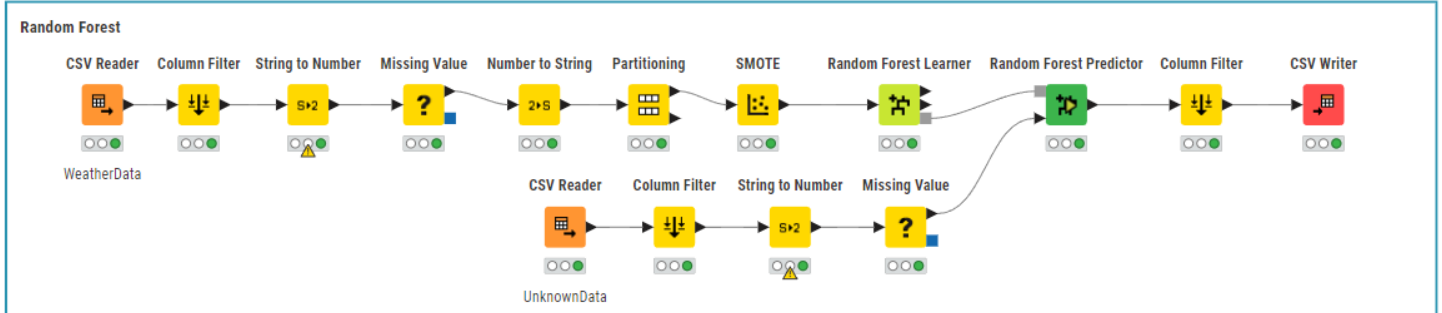
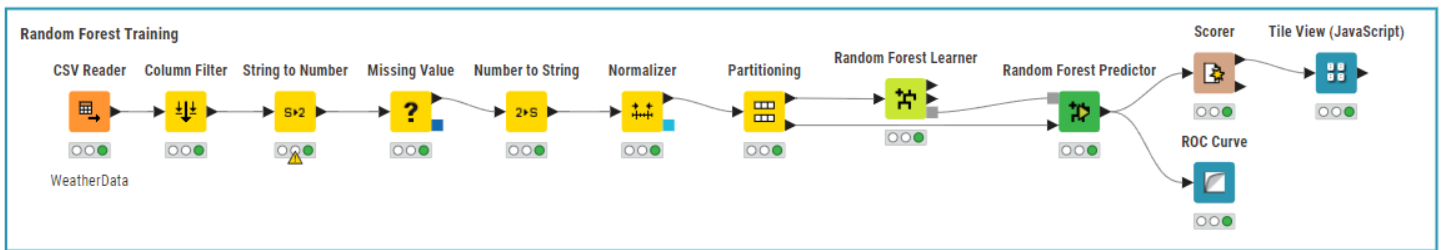


Figure 49 – Random Forest

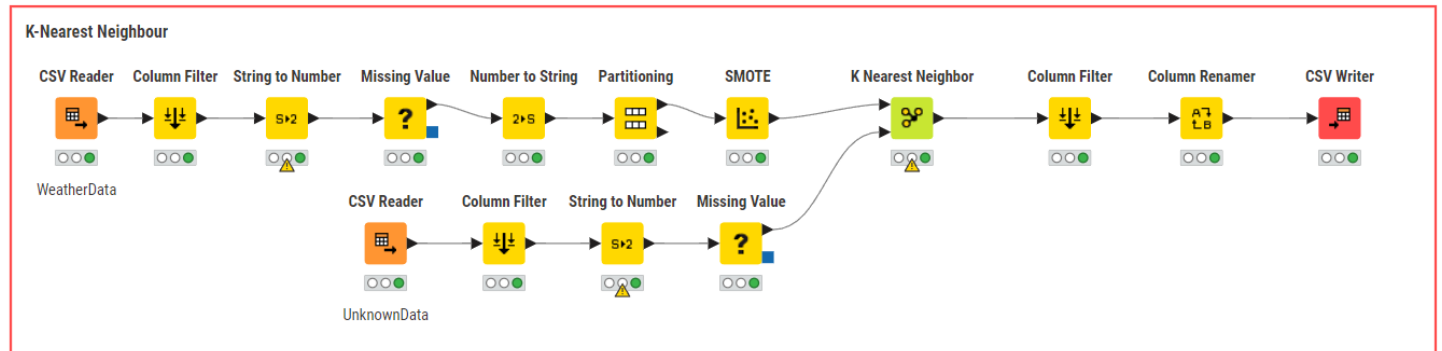
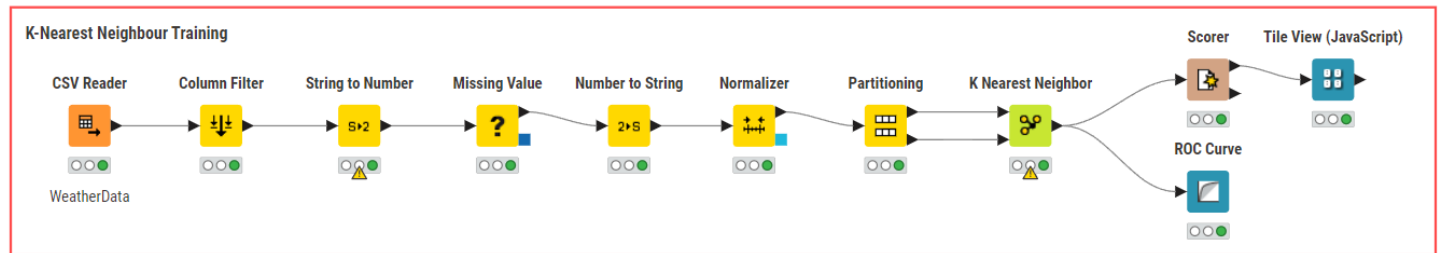


Figure 50 – KNN

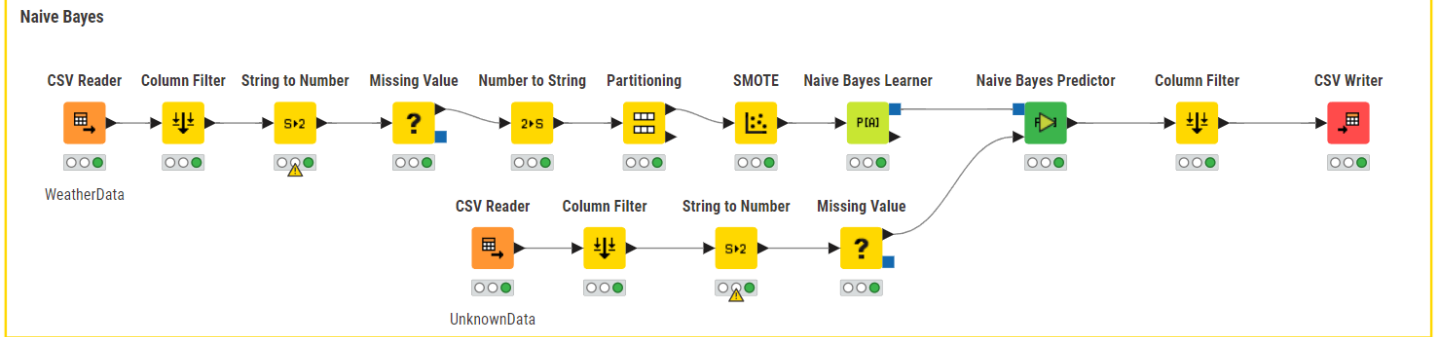
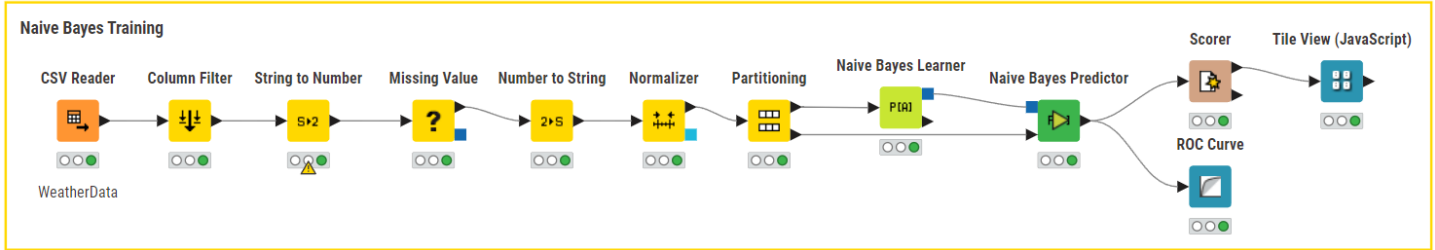


Figure 51 – Naïve Bayes

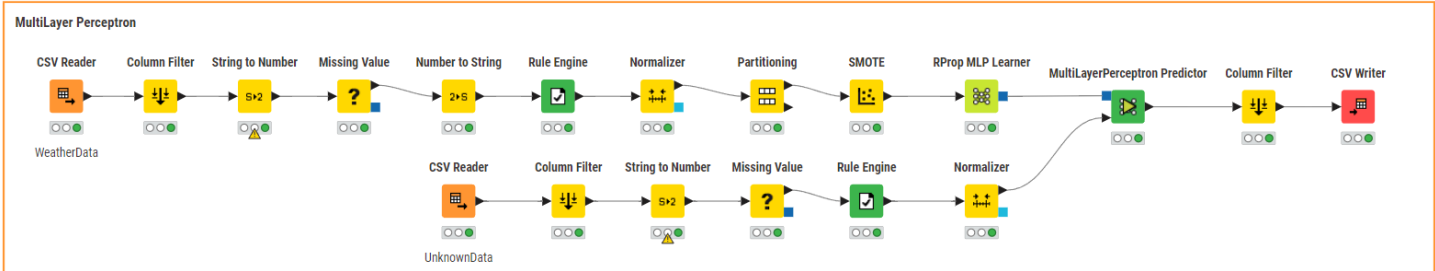
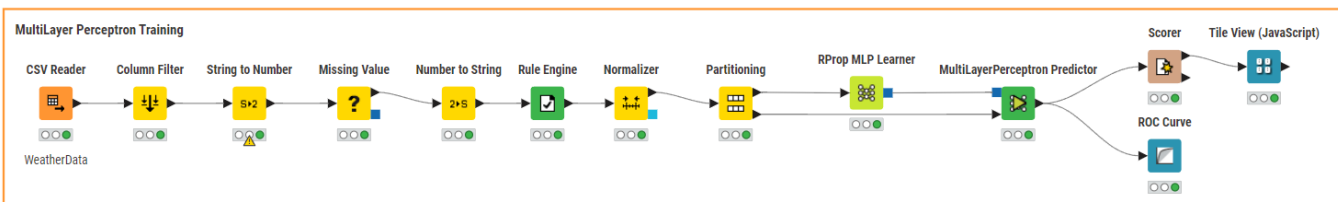


Figure 52 – MLP

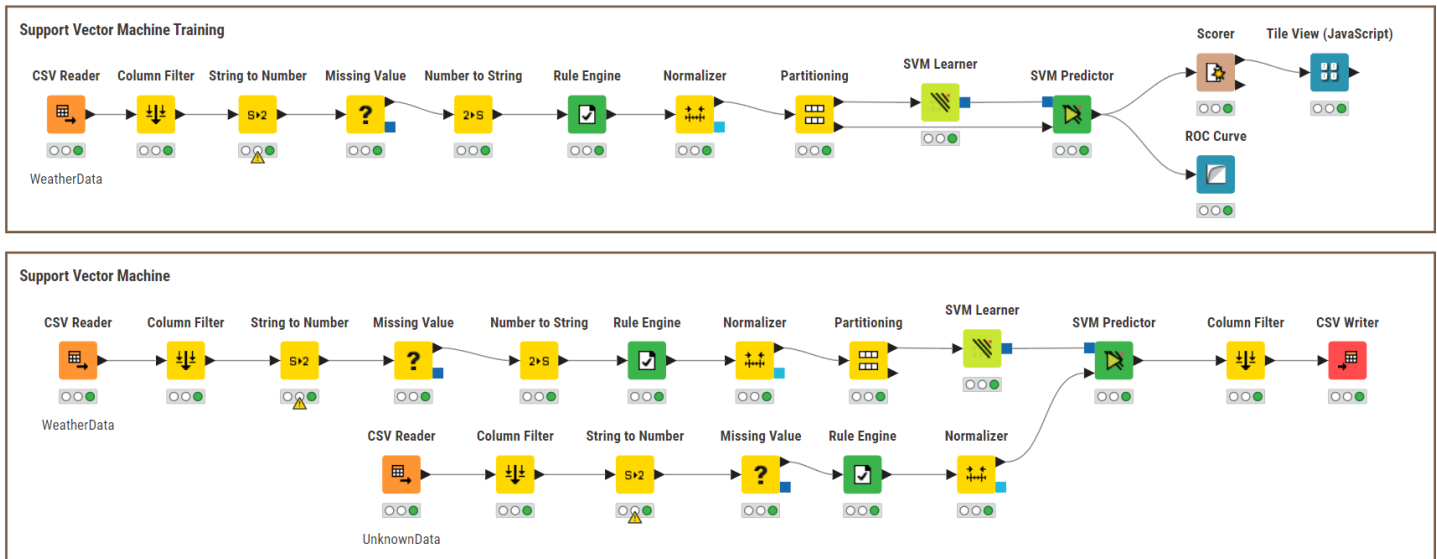


Figure 53 - SVM

```

Python > Python Workspace.py > ...
1  import pandas as pd
2
3  # Load the dataset.
4  data = pd.read_csv('E:\Assignments\Data Analytics - Assessment 3/Assignment3-WeatherData.csv')
5
6  # Check for missing values.
7  missing_values = data.isnull().sum()
8
9  # Find the column with the most missing values.
10 most_missing = missing_values.idxmax(), missing_values.max()
11
12 # Calculate the percentage of missing values for each column.
13 missing_percentage = (data.isnull().sum() / len(data)) * 100
14
15 print(most_missing)
16 print(missing_percentage)

```

Figure 54 – Python Code