University of Reading

Department of Computer Science

CS3AI18: Predicting Rainfall in Australia

Savia Laloukiotou

Student Number: 28017133
Module Convenor: Dr Yevgeniya Kovalchuk

March 7, 2022

Contents

Abstract

Rainfall forecast is a challenging endeavour. It plays a critical role in decision making and the organisation of sustainable water resource systems. Although meteorological systems produce short-term rainfall forecasts, long-term rainfall forecasting is difficult and fraught with uncertainty due to a variety of causes. Finding the optimum approach to predict the complex rainfall process in areas like Australia, where the climate is erratic, is a major challenge. The aim of this project is to build a machine learning model to address the problem of rainfall prediction. The results demonstrate that the Random Forest algorithm outperforms Logistic Regression in predicting rainfall.

Chapter 1

Background

Rainfall prediction remains a major concern, attracting the attention of governments, businesses, risk management organisations, and researchers. Rainfall is a climatic component that has an impact on a variety of human activities, including agriculture, building, power generation, forestry, and tourism (WHO, 2003). Its forecasting is crucial in as it is closely linked to severe natural events including landslides, flooding, mass movements, and avalanches.
The continent is subjected to a wide range of rainfall patterns, including floods and droughts (Sharmila, 2020). Developing a successful rainfall prediction model allows to take preventive and mitigating measures against these natural occurrences (Nicholls, 2001).

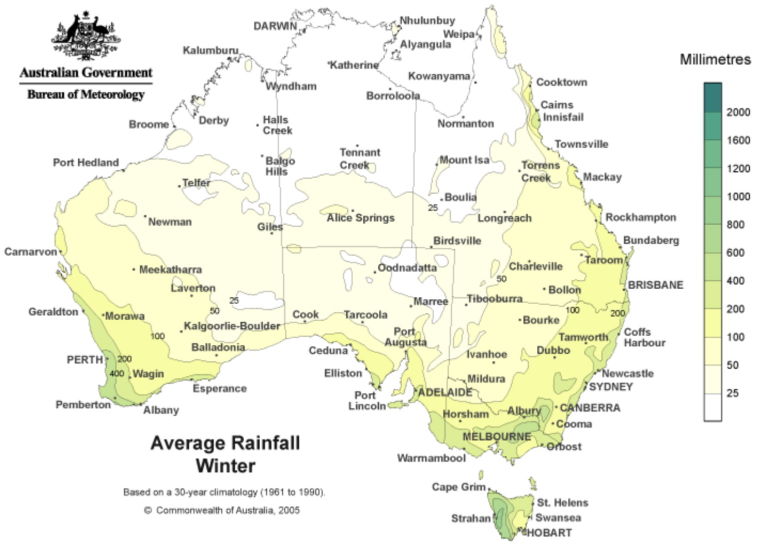


Figure 1.1: Average Winter Rainfall in Australia based on 30 year climatology.

Chapter 2

Dataset

The dataset used contains 10 years' worth of daily weather observations from multiple Australian weather stations (*Australian Government Bureau of Meteorology*, n.d.) and consists of 145460 records and 23 attributes. Table 2.1 shows a description of the attributes included. RainTomorrow is the target variable.

Name	Description
Date	The date of observation
Location	Location name of the weather station
MinTemp	The minimum temperature in degrees celsius
MaxTemp	The maximum temperature in degrees celsius
Rainfall	The amount of rainfall recorded for the day in mm
Evaporation	The Class A pan evaporation (mm) in the 24 hours to 9am
Sunshine	The number of hours of bright sunshine in the day
WindGustDir	The direction of the strongest wind gust in the 24 hours to midnight
WindGustSpeed	The speed (km/h) of the strongest wind gust in the 24 hours to midnight
WindDir9am	Direction of the wind at 9am
WindDir3pm	Direction of the wind at 3pm
WindSpeed9am	Wind speed (km/hr) averaged over 10 minutes prior to 9am
WindSpeed3pm	Wind speed (km/hr) averaged over 10 minutes prior to 3pm
Humidity9am	Humidity (percent) at 9am
Humidity3pm	Humidity (percent) at 3pm
Pressure9am	Atmospheric pressure (hpa) reduced to mean sea level at 9am
Pressure3pm	Atmospheric pressure (hpa) reduced to mean sea level at 3pm
Cloud9am	Fraction of sky obscured by cloud (in "oktas": eighths) at 9am
Cloud3pm	Fraction of sky obscured by cloud (in "oktas": eighths) at 3pm
Temp9am	Temperature (degrees C) at 9am
Temp3pm	Temperature (degrees C) at 3pm
RainToday	"Yes" if precipitation (mm) in the 24 hours to 9am exceeds 1mm, otherwise "No"
RainTomorrow	"Yes" if precipitation (mm) in the next day exceeds 1mm, otherwise "No"

Table 2.1: Names and description of the attributes.

Chapter 3

Machine Learning Model

3.1 Solution approach

Sharma et al. (2021) used the Decision Tree algorithm to construct a rainfall forecast model with an accuracy of 81%. Gupta and Ghose (2015) used the KNN (K NearestNeighbors) and Naive Bayes algorithms, with 80.7% and 78.9% accuracy respectively. However, none of the aforementioned papers mention the F1 score or other metrics. As described in the following chapters, the accuracy of a model is not always a good measure of its performance. In this approach, Random Forest and Logistic Regression algorithms are going to be implemented to test whether they can achieve improved results.

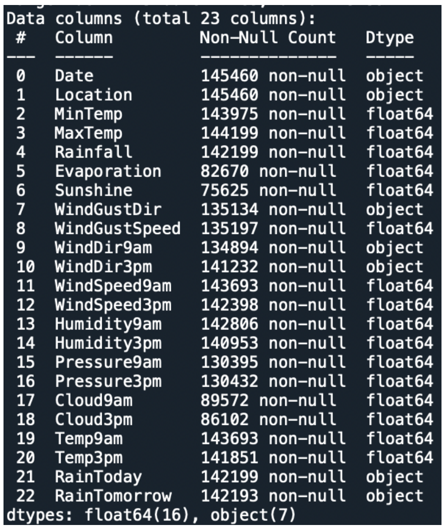
3.2 Data Preprocessing and Visualisation

Using the info() and describe() methods (Listing 3.1) we can retrieve useful insights about our data. The info() method returns the non-null count and type of each variable (Figure 3.1) and describe() returns a statistical summary of the numerical variables (Figure 3.2).

```
#%% VIEW THE SHAPE AND HAVE A QUICK LOOK AT THE DATA

print(f'The number of rows are {df.shape[0] } and the number of columns
    are {df.shape[1]}')
pd.set_option('display.max_columns', None) # to view all the columns
print(df.head()) # to have a quick look at the data

#%%
print(df.info()) # print a summary of the variables in the dataframe

#%%
print(df.describe()) # view a statistics summary of the numeric data
```

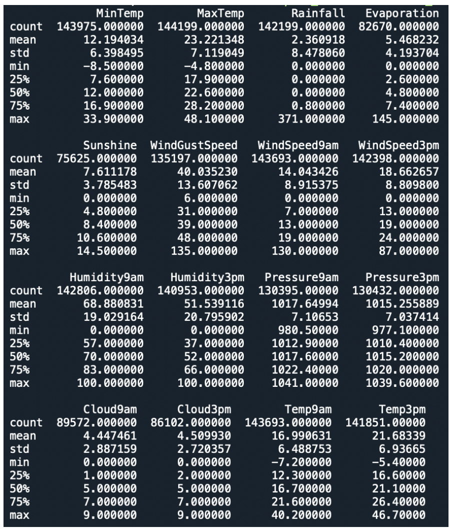
Listing 3.1: Information and statistical summary of the dataset

We can see that the dataset contains both categorical and numerical variables. Categorical variables have data type object and numerical variables have data type float64. It also appears that some columns contain a high amount of missing values. From the statistical summary, we can see that some variables have a higher standard deviation value which indicates that their data points are dispersed. A large difference between the 75th percentile and the maximum value of some variables indicates the presence of potential outliers (e.g., the max Rainfall value is 371.0 despite the 75th percentile being 0.8). The same goes for the difference between the



```
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           145460 non-null  object
 1   Location       145460 non-null  object
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object
 10  WindDir3pm     141232 non-null  object
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object
 22  RainTomorrow   142193 non-null  object
dtypes: float64(16), object(7)
```

Figure 3.1: Information about the attributes.



```
             MinTemp        MaxTemp       Rainfall   Evaporation
count  143975.000000  144199.000000  142199.000000  82670.000000
mean       12.194034      23.221348       2.360918      5.468232
std         6.398495       7.119049       8.478060      4.193704
min        -8.500000      -4.800000       0.000000      0.000000
25%         7.600000      17.900000       0.000000      2.600000
50%        12.000000      22.600000       0.000000      4.800000
75%        16.900000      28.200000       0.800000      7.400000
max        33.900000      48.100000     371.000000    145.000000

           Sunshine  WindGustSpeed   WindSpeed9am   WindSpeed3pm
count  75625.000000  135197.000000  143693.000000  142398.000000
mean       7.611178      40.035230      14.043426      18.662657
std        3.785483      13.607062       8.915375       8.809800
min        0.000000       6.000000       0.000000       0.000000
25%        4.800000      31.000000       7.000000      13.000000
50%        8.400000      39.000000      13.000000      19.000000
75%       10.600000      48.000000      19.000000      24.000000
max       14.500000     135.000000     130.000000      87.000000

         Humidity9am    Humidity3pm   Pressure9am    Pressure3pm
count  142806.000000  140953.000000  130395.00000  130432.000000
mean       68.880831      51.539116    1017.64994    1015.255889
std        19.029164      20.795902       7.10653       7.037414
min         0.000000       0.000000     980.50000     977.100000
25%        57.000000      37.000000    1012.90000    1010.400000
50%        70.000000      52.000000    1017.60000    1015.200000
75%        83.000000      66.000000    1022.40000    1020.000000
max       100.000000     100.000000    1041.00000    1039.600000

           Cloud9am      Cloud3pm        Temp9am       Temp3pm
count  89572.000000  86102.000000  143693.000000  141851.00000
mean       4.447461      4.509930      16.990631      21.68339
std        2.887159      2.720357       6.488753       6.93665
min        0.000000      0.000000      -7.200000      -5.40000
25%        1.000000      2.000000      12.300000      16.60000
50%        5.000000      5.000000      16.700000      21.10000
75%        7.000000      7.000000      21.600000      26.40000
max        9.000000      9.000000      40.200000      46.70000
```

Figure 3.2: Statistical description of the numerical attributes.

minimum value and the 25th percentile. To further investigate how the numerical variables are distributed, we plot their histograms (Figure 3.3). The histograms show us that some variables follow normal distribution, (e.g. Humidity3pm, MinTemp, Temp9am), while some are highly skewed (e.g. Evaporation and Rainfall).

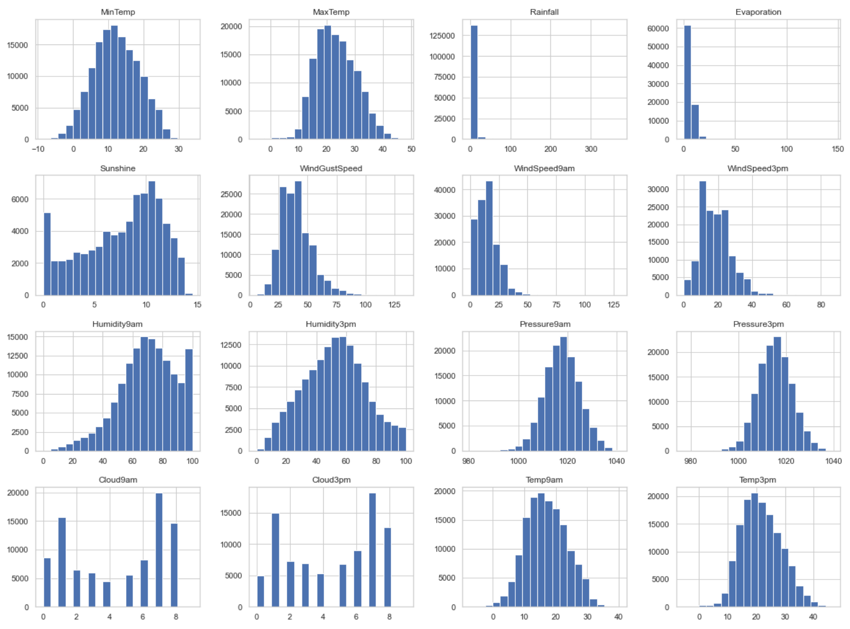


Figure 3.3: Distribution of the numerical variables.

We now investigate the correlation between the variables. In order to do so, we plot a correlation heatmap (Figure 3.4).

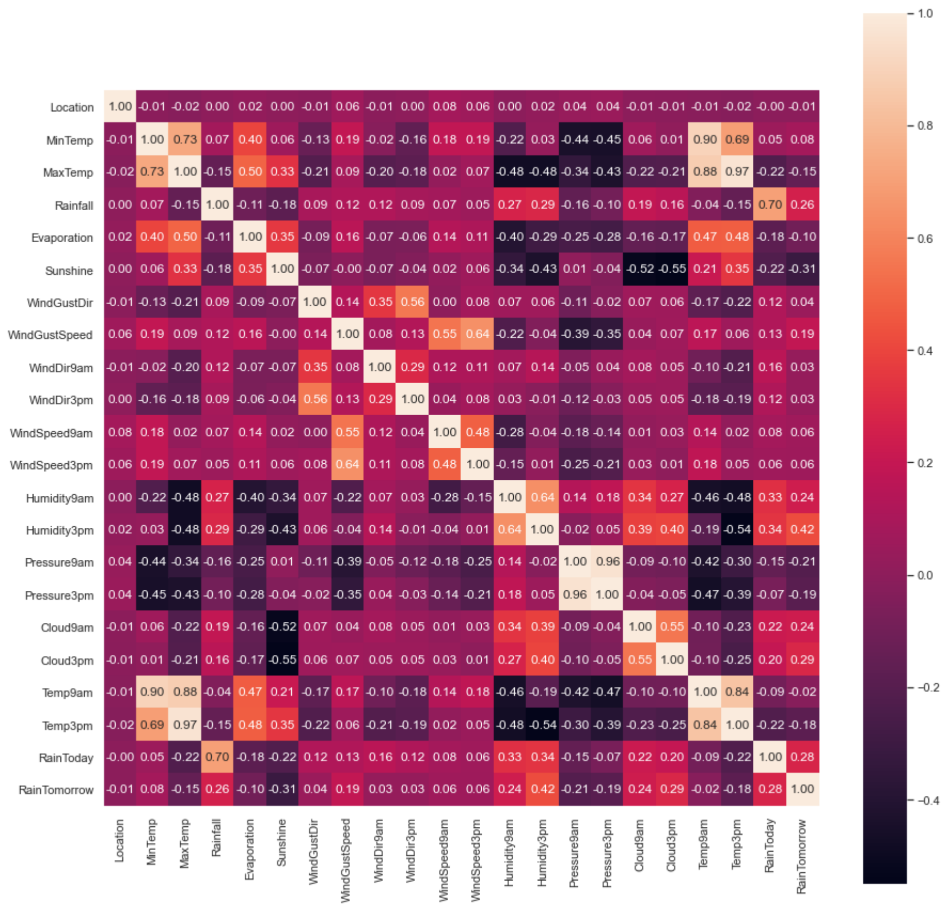


Figure 3.4: Variable correlation heatmap.

The feature pairs listed below are highly correlated:

- MaxTemp and MinTemp
- Pressure9am and Pressure3pm
- Temp9am and Temp3pm
- Evaporation and MaxTemp
- MaxTemp and Temp3pm

Strong correlated features (either positive or negative) have intuitive reasons for being so. However, by looking at the pair diagram below (Figure 3.5), we may learn more about the pairwise correlation between these features. The "yes" and "no" clusters of RainTomorrow are plainly visible. There is only a small amount of overlap.

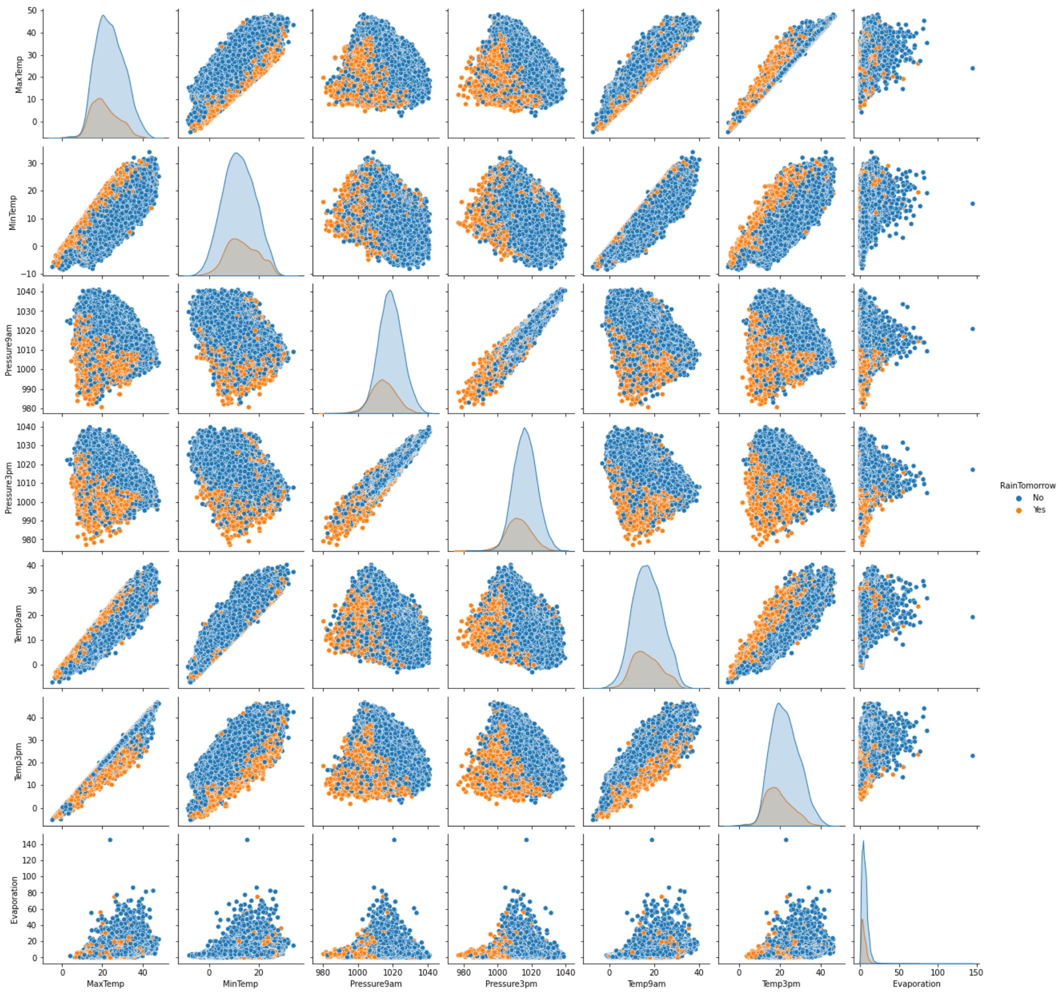


Figure 3.5: Pairplots of the variables.

Next, we have a closer look at the balance of the class variable RainTomorrow. As shown in Figure 3.6, the target class is imbalanced.

Balance of positive and negative classes:
0: 79.4%
1: 20.6%

Standard classifier algorithms have a bias towards classes which have large number of instances. The features of the minority class are treated as noise and are often ignored. Thus, there is a high probability of misclassification of the minority class as compared to the majority class (AnalyticsVidhya, 2017). In our case, if the classifier always predicts No rainfall, it will be correct around 80% of the time although in practice it is performing no better than an unskilled majority class classifier. This is also known as the Accuracy Paradox (MachineLearningMastery, 2021).

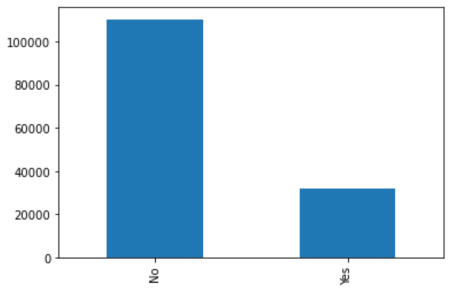


Figure 3.6: Target Class Distribution.

Then, we check for missing values in our dataset. When checking for missing values in categorical variables, checking for unique values and their count is essential to spot irrelevant entries as well (e.g. null values can be encoded as "?").

```python
df_cat= df.select_dtypes(exclude=['float']) # include only object type
    variables
for col in df_cat.columns:
    print(df_cat[col].unique()) # to print categories name only
    print(df_cat[col].value_counts()) # to print count of every category
```

Listing 3.2: Printing unique values and value counts of categorical variables

The output of the code in Listing 3.2 showed that missing values are encoded as "nan". To visualize them better, we plot a heatmap (Figure 3.7).

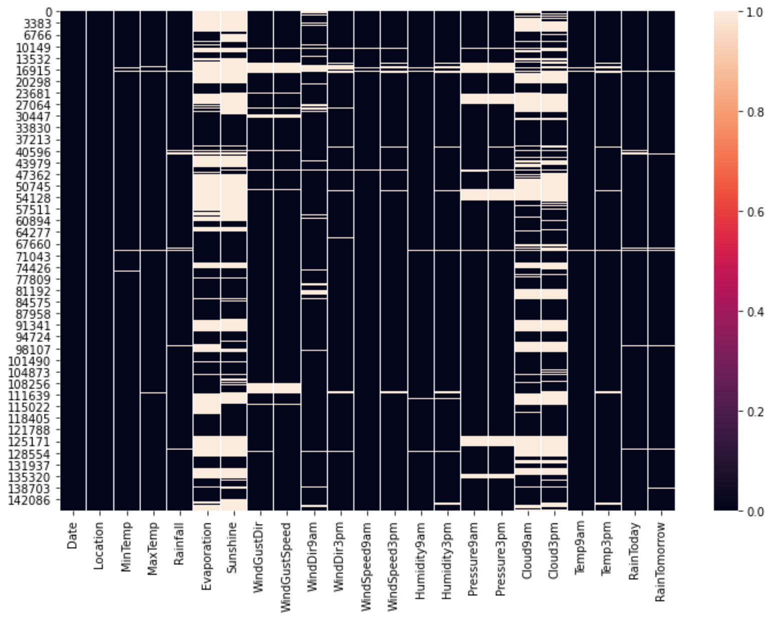


Figure 3.7: Missing values heatmap.

It is clear that 'Evaporation', 'Sunshine', 'Cloud9am' and 'Cloud3pm' are the features that contain the most missing values. To handle missing values throughout all the columns, we first split the features to two lists, Numerical and Categorical. Then, using the fillna() method we replace the null values with the median value for numerical, and the mode value for categorical variables (Listing 3.3). The median is more robust (less sensitive to outliers in the data) than the mean (Wand, n.d.).

```python
# List of categorical variables
categorical = [i for i in df.columns if df[i].dtypes == 'O']
# List of numerical variables
numerical = [i for i in df.columns if i not in categorical]
print('Categorical:\n', categorical , '\n\n', 'Numerical:\n', numerical)

#%%
# function to replace null numerical values
def replace_numerical(df):
    for col in numerical: # for all numerical columns
        df[col] = df[col].fillna(df[col].median()) # replace missing value with the median
    return df

# function to replace null categorical values
def replace_categorical(df):
    for col in categorical:  # for all categorical columns
        df[col] = df[col].fillna(df[col].mode()[0]) # replace missing value with mode

    return df

#%% REPLACE NUMERICAL AND CATEGORICAL VALUES
df = replace_numerical(df)
df = replace_categorical(df)
```

Listing 3.3: Replacing missing values

After replacing the missing values, we plot the missing values heatmap again (Figure 3.8) and observe that all null values have been eliminated.

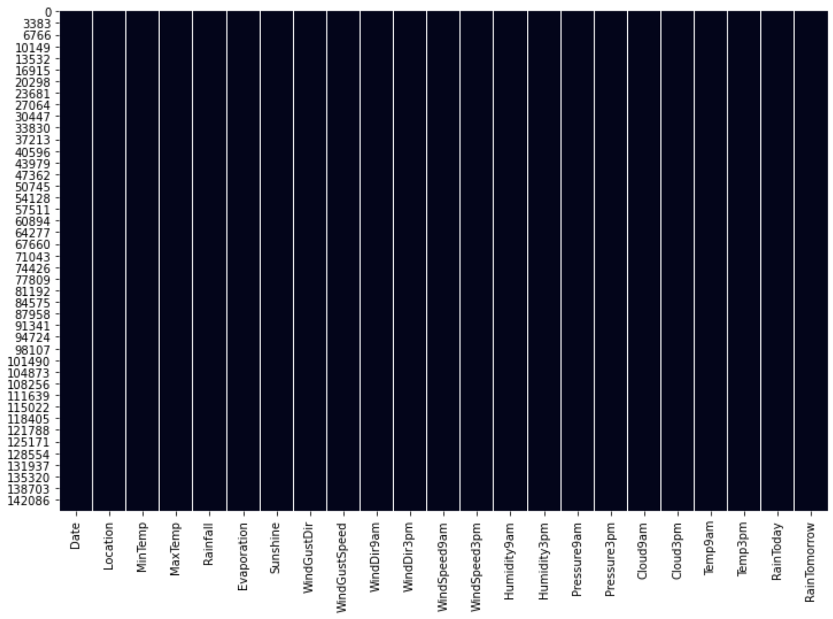


Figure 3.8: Missing values heatmap after replacement.

Machine learning models require all input and output variables to be numeric. This means that categorical variables must be encoded. The "YES" and "NO" variables (i.e. RainToday and RainTomorrow) are encoded with 1 and 0. (Listing 3.4).

```python
#%% REPLACE YES OR NO VALUES WITH 1 AND 0
df = df.replace({'RainTomorrow': {'Yes': 1, 'No': 0}})
df = df.replace({'RainToday': {'Yes': 1, 'No': 0}})
```

Listing 3.4: Encoding RainToday and RainTomorrow Variables

The 'Date' variable is dropped as it does not contribute to our analysis, and the rest of the categorical variables are encoded using the label encoder (Listing 3.5).

```python
#%% DROP THE COLUMN DATE
df.drop('Date', axis=1, inplace = True)

#%% ENCODE THE CATEGORICAL VALUES
le = LabelEncoder() # creating instance of labelencoder
# Assigning numerical values to the columns
df['Location'] = le.fit_transform(df['Location'])
df['WindDir9am'] = le.fit_transform(df['WindDir9am'])
df['WindDir3pm'] = le.fit_transform(df['WindDir3pm'])
df['WindGustDir'] = le.fit_transform(df['WindGustDir'])
```

Listing 3.5: Encoding the categorical variables

As discussed earlier, the dataset contains potential outliers (data points that differ significantly from other observations). A suitable way to visualize them, is by plotting boxplots for the numerical variables.

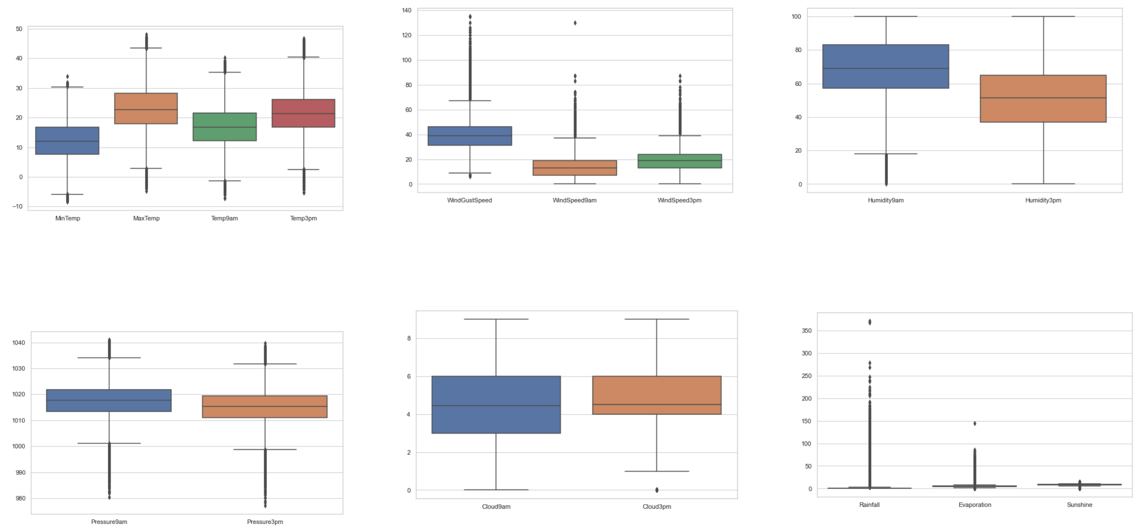


Figure 3.10: Boxplots of numerical variables before outlier treatment

Figure 3.10 confirms the presence of outliers. We can handle these outliers by using the z-score method. Z score tells how many standard deviations away a data point is from the mean. If the z score of a data point is more than 3, it indicates that the data point is quite different from the other data points. As shown in Listing 3.6, we keep all the observations with a z-score less than 3. We use the absolute value of the z-score as it can also be negative. We also print the shape of the dataset before and after outlier treatment.

```
#%% HANDLE OUTLIERS AND PRINT SHAPE BEFORE AND AFTER

print('Shape of DataFrame Before Removing Outliers', df.shape )
df = df[(np.abs(stats.zscore(df)) < 3).all(axis=1)] # keep observations with z-score absolute value less than 3
print('Shape of DataFrame After Removing Outliers', df.shape )
```

Listing 3.6: Outlier treatment using z-score from scipy's stats module

We plot the boxplots again and we observe that outlier presence has been significantly improved (Figure 3.12).

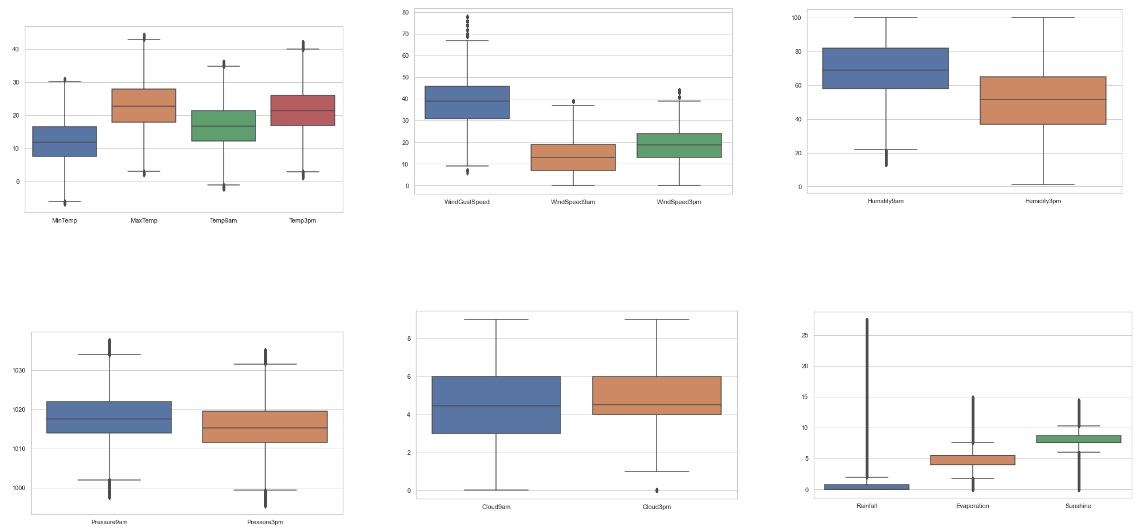


Figure 3.12: Boxplots of numerical variables after outlier treatment

The next step is data standardization (Listing 3.7). Standardization scales each input variable separately by subtracting the mean (called centering) and dividing by the standard deviation to shift the distribution to have a mean of zero and a standard deviation of one (Brownlee, 2020). The fit method is calculating the mean and variance of each of the features present in our data. The transform method is transforming all the features using the respective mean and variance. If we perform the fit method on our test data as well, we will get a new mean and variance for each feature, as well as allowing our model to learn about our test data (Khanna, 2020). This results to what is called data leakage (sharing the information between the test and training data sets).

```
#%% SCALING THE DATA
# create and fit the scaler on training set
sc=StandardScaler()
X_train=sc.fit_transform(X_train)
X_test=sc.transform(X_test) # transform the test set
```

Listing 3.7: Data Standardization.

As shown earlier in Figure 3.6, the class variable is imbalanced. One approach to addressing imbalanced datasets is to oversample the minority class. The most widely used approach to synthesizing new examples is called the Synthetic Minority Oversampling TEchnique, or SMOTE (Brownlee, 2021*b*). To avoid data leakage, SMOTE is only applied on the training set (Listing 3.8). We print the number of samples of the class variable before and after applying SMOTE.

Before SMOTE: 0: 86697, 1: 22503
After SMOTE: 0: 86697, 1: 86697

```python
#%% OVERSAMPLE THE MINORITY CLASS AND PRINT THE NEW RATIO
# print number of samples before smote
counter = Counter(y_train)
print('Before SMOTE: ', counter)

# create the SMOTE oversampler
smt = SMOTE()

# fit oversampler on training set
X_smote, y_smote = smt.fit_resample(X_train, y_train)

# print number of samples after smote
counter = Counter(y_smote)
print('After SMOTE: ', counter)
```

Listing 3.8: Oversampling the minority class.

3.3 Model Training and Evaluation

To train and evaluate our model, we create the following functions:

The fit_evaluate() function (Listing 3.9) fits the model, predicts the labels, calculates training and testing time, prints the classification report and plots the confusion matrix.

```python
# Function for fitting and evaluating the models
def fit_evaluate(model, x_train, y_train, x_test, y_test):
    train_start = time.time() # model training starts
    model.fit(x_train, y_train) # the model is fitted on the training set
    train_stop = time.time() # model training stops
    test_start=time.time() # testing starts
    y_pred = model.predict(x_test) # model predicts the labels of the features in the test set
    test_stop=time.time() # testing stops
    print("Training time: ", train_stop - train_start) # print training time
    print("Testing time: ", test_stop - test_start) # print testing time
    print(classification_report(y_test, y_pred)) # print classification report

    conf_matrix=confusion_matrix(y_test, y_pred) # calculate confusion matrix
    ax = sns.heatmap(conf_matrix, annot=True, fmt = "g")
    # plot the confusion matrix
    ax.set_title('Confusion Matrix\n\n');
    ax.set_xlabel('\nPredicted Values')
    ax.set_ylabel('Actual Values ');


    ## Display the visualization of the Confusion Matrix.
    plt.show()
```

Listing 3.9: The fit_evaluate() function.

The cross_validate() function (Listing 3.10) calculates the 5-fold cross validation scores of the model, and prints their mean and standard deviation. This gives a more efficient use of the data as every observation is used for both training and testing. Five fold cross validation was used instead of ten-fold to reduce the complexity.

```python
def cross_validate(model, x_train, y_train):
    scores=cross_val_score(model, x_train, y_train, cv=5) # calculate
   cross validation scores
    print("Cross Validation Scores:\n")
    print("Mean score: ", scores.mean()) # print the mean
    print("Standard deviation: ", scores.std()) # print standard deviation
    return scores
```

Listing 3.10: The cross_validate() function.

The ROC_AUC() function (Brownlee, 2021*a*) calculates and plots the ROC (Receiver Operating Characteristics) Curve of the model (Listing 3.11). ROC is a probability curve and AUC (Area under the curve) represents the degree or measure of separability. It tells how much the model is capable of distinguishing between classes. A perfect classifier will have a ROC AUC equal to 1, whereas a purely random classifier will have a ROC AUC equal to 0.5.

```python
def ROC_AUC(model):
    ns_probs = [0 for _ in range(len(y_test))] # generate a no skill
   prediction
    lr_probs = model.predict_proba(X_test) # predict probabilities
    lr_probs = lr_probs[:, 1] # keep probabilities for the positive
   outcome only
    # calculate scores
    ns_auc = roc_auc_score(y_test, ns_probs)
    lr_auc = roc_auc_score(y_test, lr_probs)

    # summarize scores
    print('No Skill: ROC AUC=%.3f' % (ns_auc))
    print('Model: ROC AUC=%.3f' % (lr_auc))

    # calculate roc curves
    ns_fpr, ns_tpr, _ = roc_curve(y_test, ns_probs)
    lr_fpr, lr_tpr, _ = roc_curve(y_test, lr_probs)

    # plot the roc curve for the model
    plt.plot(ns_fpr, ns_tpr, linestyle='--', label='No Skill')
    plt.plot(lr_fpr, lr_tpr, marker='.', label='Model')

    # axis labels
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    # show the legend
    plt.legend()
    # show the plot
    plt.show()
```

Listing 3.11: ROC_AUC() function.

Next, we call the above functions to fit and evaluate our model. The algorithms used are Logistic Regression and Random Forest. Firstly, the models are fitted on the imbalanced data (Listing 3.12).

```python
#%% ---- LOGISTIC REGRESSION -------

logreg = LogisticRegression(random_state=42,max_iter=700) # create the
   classifier
fit_evaluate(logreg, X_train, y_train, X_test, y_test) # fit and evaluate
   the model
```

```
scores_logreg=cross_validate(logreg,X_train, y_train) # print cross
    validation scores

#%% -------- RANDOM FOREST -----------

rfc=RandomForestClassifier() # create the classifier
fit_evaluate(rfc, X_train, y_train, X_test, y_test) # fit and evaluate the
     model
scores_rfc=cross_validate(rfc,X_train, y_train) # print cross validation
    scores
```

Listing 3.12: Fitting and evaluating the models with imbalanced data.

Next, the oversampled data is used as input (Listing 3.13)

```
#%%  -------- LOGISTIC REGRESSION ON OVERSAMPLED DATA --------

logreg_res=LogisticRegression(random_state=42,max_iter=700) # create the
    classifier
fit_evaluate(logreg_res, X_smote, y_smote, X_test, y_test) # fit and
    evaluate the model
log_res_scores=cross_validate(logreg_res,X_smote, y_smote) # print cross
    validation scores

#%% --------- RANDOM FOREST ON OVERSAMPLED DATA ---------
rfc_res=RandomForestClassifier() # create the classifier
fit_evaluate(rfc_res, X_smote, y_smote, X_test, y_test) # fit and evaluate
     the model
rfc_res_scores=cross_validate(rfc_res,X_smote, y_smote) # print cross
    validation scores
```

Listing 3.13: Fitting and evaluating the models with balanced data.

Multicollinearity occurs when features are highly correlated with one or more of the other features in the dataset. PCA (Principal Component Analysis) takes advantage of multicollinearity and combines the highly correlated variables into a set of uncorrelated variables. Therefore, PCA can effectively eliminate multicollinearity between features. We first need to find the number of principal components that capture an adequate percentage of information (Listing 3.14).

```
# Create and run a PCA
pca = PCA(n_components=None)
x_scaled = sc.fit_transform(X)
pca.fit(x_scaled) # fit the pca to scales x values (X is already scaled)

# Get the eigenvalues
print("Eigenvalues:")
print(pca.explained_variance_)
print()

# Get explained variances
print("Variances (Percentage):")
print(pca.explained_variance_ratio_ * 100)
print()

# Make the scree plot
plt.plot(np.cumsum(pca.explained_variance_ratio_ * 100))
plt.xlabel("Number of components (Dimensions)")
plt.ylabel("Explained variance (%)")
```

Listing 3.14: Running PCA to find optimal number of components.

Figure 3.13 shows the number of components and the percentage of their explained variance.

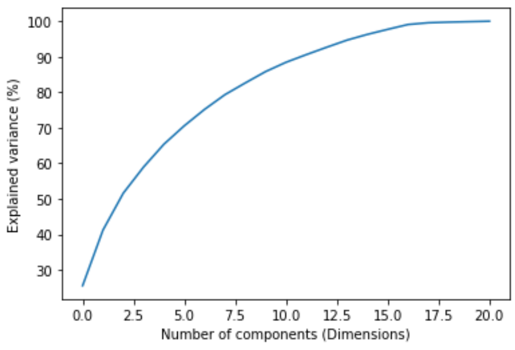


Figure 3.13: Number of components and the percentage of their explained variance.

We can see that we can capture around 90% of the information by using 11 principal components. We run PCA using n_components=11 and create a new dataframe that contains these 11 components. We then split this dataset to training and test sets (Listing 3.15).

```
# run pca again with 11 principal components
x_scaled = sc.fit_transform(X)
pca = PCA(n_components=11)
X_pca = pca.fit_transform(x_scaled)

# Get the transformed dataset
X_pca = pd.DataFrame(X_pca)
print(X_pca.head())
print("\nSize: ")
print(X_pca.shape) # print the size of the dataframe
#%%
# split PCA dataset to train and test sets
x_train_pca, x_test_pca, y_train_pca, y_test_pca = train_test_split(X_pca,
    y, test_size=0.20, shuffle=True, random_state=2)
```

Listing 3.15: Running PCA with 11 principal components and creating the new dataframe.

We then train our model using the training set that contains the principal components and evaluate it (Listing 3.16).

```
#%% -------- LOGISTIC REGRESSION WITH PRINCIPAL COMPONENTS  -------
logreg_pca = LogisticRegression(random_state=42,max_iter=700) # create the
    classifier
fit_evaluate(logreg_pca, x_train_pca, y_train_pca, x_test_pca, y_test_pca)
    # fit and evaluate the model
log_pca=cross_validate(logreg_pca,x_train_pca, y_train_pca) # print cross
   validation scores

#%% ------- RANDOM FOREST WITH PRINCIPAL COMPONENTS ------
rfc_pca=RandomForestClassifier() # create the classifier
fit_evaluate(rfc_pca, x_train_pca, y_train_pca, x_test_pca, y_test_pca) #
   fit and evaluate the model
rfc_pca=cross_validate(rfc_pca,x_train_pca, y_train_pca) # print cross
   validation scores
```

Listing 3.16: Running PCA with 11 principal components.

As shown in Listing 3.17, to tune the hyperparameters and improve model performance, we run GridSearchCV for Logistic Regression and RandomizedSearchCV for Random Forest (GridSearchCV with Random Forest has a high computational time).

```
#%% RANDOM FOREST WITH RANDOMIZED SEARCH CV
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 10, stop = 80, num =
    10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [2,4]
# Minimum number of samples required to split a node
min_samples_split = [2, 5]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2]
# Method of selecting samples for training each tree
bootstrap = [True, False]

# Create the param grid
param_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}

# create RandomizedSearchCV instance
rf_RandomGrid = RandomizedSearchCV(estimator = rfc, param_distributions =
    param_grid, cv = 10, verbose=2, n_jobs = 4)

rf_RandomGrid.fit(X_train, y_train) # fit to training sets
#%%
print(rf_RandomGrid.best_params_) # print the best parameters
```

Listing 3.17: Random Forest Hyperparameter Tuning with RandomizedSearchCV.

The output parameters are: n_estimators=72, min_samples_split=5, min_samples_leaf=1, max_features='sqrt', max_depth=4, bootstrap='True'.

Listing 3.18 shows the GridSearchCV performed for Logistic Regression. The parameters for the search are saved in a configuration file.

```
#%% LOGISTIC REGRESSION GRIDSEARCH CV
# load contents from config file that contains the parameters
file="params.config"
contents=open(file).read()
parameters=eval(contents)
# create gridsearch cv instance
grid_search = GridSearchCV(estimator = logreg,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 5,
                           verbose=0)

# fit the gridsearchcv to the training set
grid_search.fit(X_train, y_train)

#%%

```

```python
print('GridSearch CV best score : {:.4f}\n\n'.format(grid_search.
    best_score_))
print('Parameters that give the best results :','\n\n', (grid_search.
    best_params_))
# print estimator that was chosen by the GridSearch
print('\n\nEstimator that was chosen by the search :','\n\n', (grid_search
    .best_estimator_))
```

Listing 3.18: Logistic Regression Hyperparameter Tuning with GridSearchCV.

```
{
'penalty':['l1','l2'],
'C':[1, 10, 100, 1000]
}
```

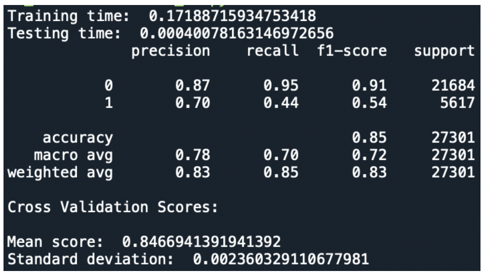
Listing 3.19: params.config.

The output parameters are: C=1, penalty: 'l2'. We then train both models using the hyperparameters acquired from the search.

```python
#%% ----- TUNED LOGISTIC REGRESSION ------
log_tuned=LogisticRegression(C=100, max_iter=700, random_state=42)
fit_evaluate(log_tuned, X_train, y_train, X_test, y_test)
scores_log_tuned=cross_validate(log_tuned,X_train, y_train)

#%% TUNED RANDOM FOREST
rfc_tuned=RandomForestClassifier(n_estimators=72, min_samples_split=5,
    min_samples_leaf=1, max_features='sqrt', max_depth=4, bootstrap='True')
fit_evaluate(rfc_tuned, X_train, y_train, X_test, y_test)
scores_rfc_tuned=cross_validate(rfc_tuned,X_train, y_train)
```

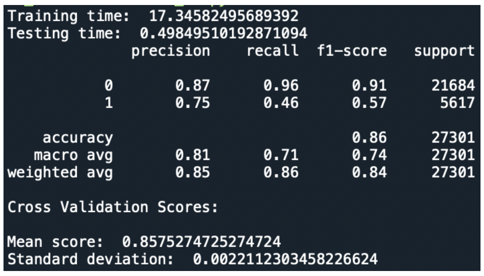
Listing 3.20: Train models with tuned hyperparameters.

The results of our models are demonstrated below:



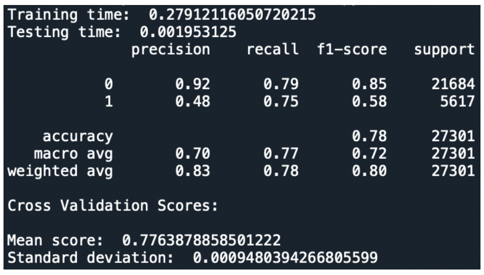
```
Training time:  0.17188715934753418
Testing time:  0.00040078163146972656
              precision    recall  f1-score   support

           0       0.87      0.95      0.91     21684
           1       0.70      0.44      0.54      5617

    accuracy                           0.85     27301
   macro avg       0.78      0.70      0.72     27301
weighted avg       0.83      0.85      0.83     27301

Cross Validation Scores:

Mean score:  0.8466941391941392
Standard deviation:  0.002360329110677981
```

Figure 3.14: Default Logistic Regression.



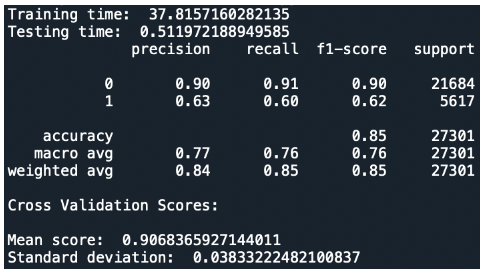
```
Training time:  17.34582495689392
Testing time:  0.49849510192871094
              precision    recall  f1-score   support

           0       0.87      0.96      0.91     21684
           1       0.75      0.46      0.57      5617

    accuracy                           0.86     27301
   macro avg       0.81      0.71      0.74     27301
weighted avg       0.85      0.86      0.84     27301

Cross Validation Scores:

Mean score:  0.8575274725274724
Standard deviation:  0.0022112303458226624
```

Figure 3.15: Default Random Forest.



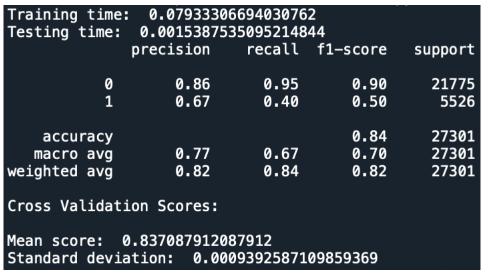
```
Training time:  0.27912116050720215
Testing time:  0.001953125
              precision    recall  f1-score   support

           0       0.92      0.79      0.85     21684
           1       0.48      0.75      0.58      5617

    accuracy                           0.78     27301
   macro avg       0.70      0.77      0.72     27301
weighted avg       0.83      0.78      0.80     27301

Cross Validation Scores:

Mean score:  0.7763878858501222
Standard deviation:  0.0009480394266805599
```

Figure 3.16: Oversampled Logistic Regression.



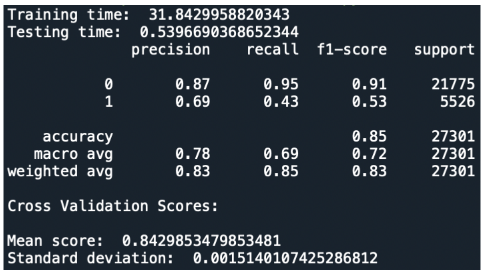
```
Training time:  37.8157160282135
Testing time:  0.511972188949585
              precision    recall  f1-score   support

           0       0.90      0.91      0.90     21684
           1       0.63      0.60      0.62      5617

    accuracy                           0.85     27301
   macro avg       0.77      0.76      0.76     27301
weighted avg       0.84      0.85      0.85     27301

Cross Validation Scores:

Mean score:  0.9068365927144011
Standard deviation:  0.03833222482100837
```

Figure 3.17: Oversampled Random Forest.



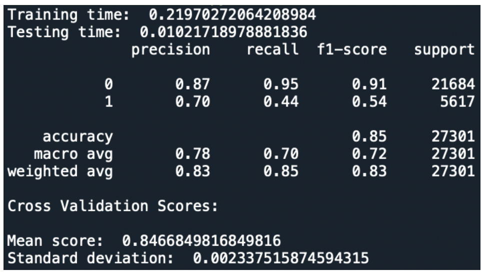
```
Training time:  0.07933306694030762
Testing time:  0.0015387535095214844
              precision    recall  f1-score   support

           0       0.86      0.95      0.90     21775
           1       0.67      0.40      0.50      5526

    accuracy                           0.84     27301
   macro avg       0.77      0.67      0.70     27301
weighted avg       0.82      0.84      0.82     27301

Cross Validation Scores:

Mean score:  0.837087912087912
Standard deviation:  0.0009392587109859369
```

Figure 3.18: PCA Logistic Regression.



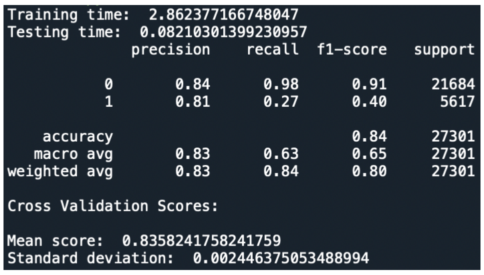
```
Training time:  31.8429958820343
Testing time:  0.5396690368652344
              precision    recall  f1-score   support

           0       0.87      0.95      0.91     21775
           1       0.69      0.43      0.53      5526

    accuracy                           0.85     27301
   macro avg       0.78      0.69      0.72     27301
weighted avg       0.83      0.85      0.83     27301

Cross Validation Scores:

Mean score:  0.8429853479853481
Standard deviation:  0.0015140107425286812
```

Figure 3.19: PCA Random Forest.



```
Training time:  0.21970272064208984
Testing time:  0.01021718978881836
              precision    recall  f1-score   support

           0       0.87      0.95      0.91     21684
           1       0.70      0.44      0.54      5617

    accuracy                           0.85     27301
   macro avg       0.78      0.70      0.72     27301
weighted avg       0.83      0.85      0.83     27301

Cross Validation Scores:

Mean score:  0.8466849816849816
Standard deviation:  0.002337515874594315
```

Figure 3.20: Tuned Logistic Regression.



```
Training time:  2.862377166748047
Testing time:  0.08210301399230957
              precision    recall  f1-score   support

           0       0.84      0.98      0.91     21684
           1       0.81      0.27      0.40      5617

    accuracy                           0.84     27301
   macro avg       0.83      0.63      0.65     27301
weighted avg       0.83      0.84      0.80     27301

Cross Validation Scores:

Mean score:  0.8358241758241759
Standard deviation:  0.002446375053488994
```

Figure 3.21: Tuned Random Forest.

3.4 Results

The results conclude the following:

- Oversampling produced a better F1 score compared to an imbalanced dataset in Logistic Regression, but significantly decreased the accuracy. For Random Forest, oversampling increased the F1 score without significantly reducing the accuracy.
- Compared to the default models, PCA improved the training time of Logistic Regression without reducing prediction capacity drastically, but increased computational time for Random Forest without producing better results.
- Tuning Logistic Regression produced the exact same results as the default model but decreased training time. Tuning Random Forest had decreased training time but did not bring significant gains that justified the loss of prediction capacity.
- All the cross validation scores are similar to the accuracies in the classification report. This means that all the models are stable. The small standard deviation of the scores also verifies this.

We come to the conclusion that the best Logistic Regression model is Tuned Logistic Regression and the best Random Forest model is the oversampled model. We now need to compare these two models. To do that, we have a look at their confusion matrices.

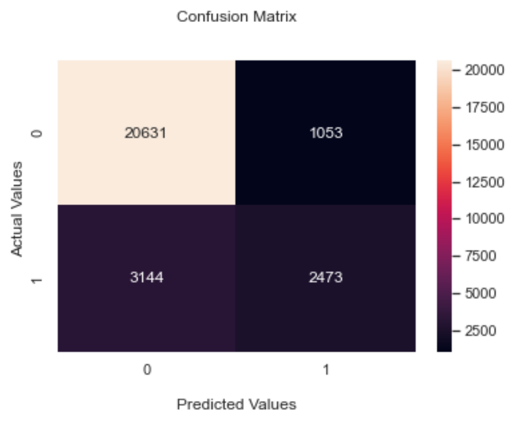


Figure 3.22: Logistic Regression Confusion Matrix.

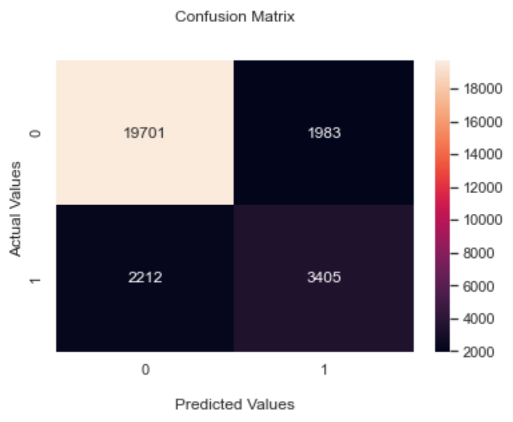


Figure 3.23: Random Forest Confusion Matrix.

We can see that Logistic Regression predicts the negative class (No rainfall) better than Random Forest, while Random Forest predicts the positive class (Rainfall) more accurately. Here we need to think about the importance of correct classification. In this case, True Positives are more important than True Negatives as heavy rainfall in Australia is a bigger danger than an unexpected drought (Reuters, 2022). Considering this, the Random Forest model seems to be a better choice. To have a better look at the models' performance, we investigate the ROC curve and the AUC.

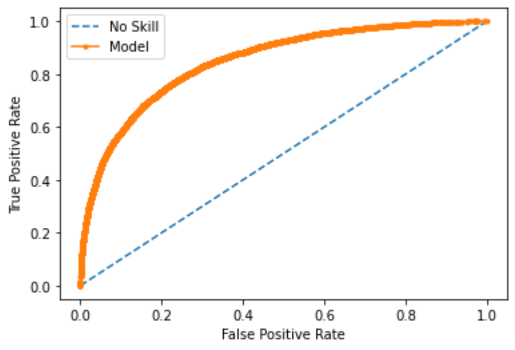


Figure 3.24: Logistic Regression ROC-AUC.

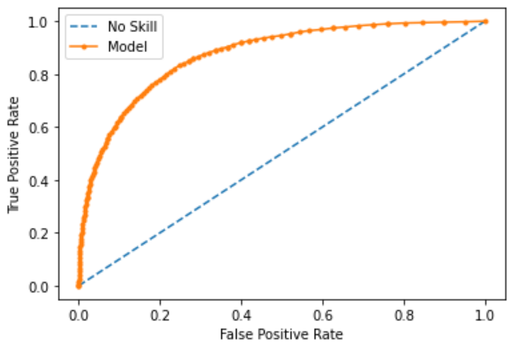


Figure 3.25: Random Forest ROC-AUC.

Since we can not tell a difference visually, we have a look at the metrics.

No Skill predictor: ROC AUC=0.500
Logistic Regression: ROC AUC=0.850
Random Forest: ROC AUC=0.874

Finally, we plot the cross validation scores of the two models using a box plot (Figure 3.26).

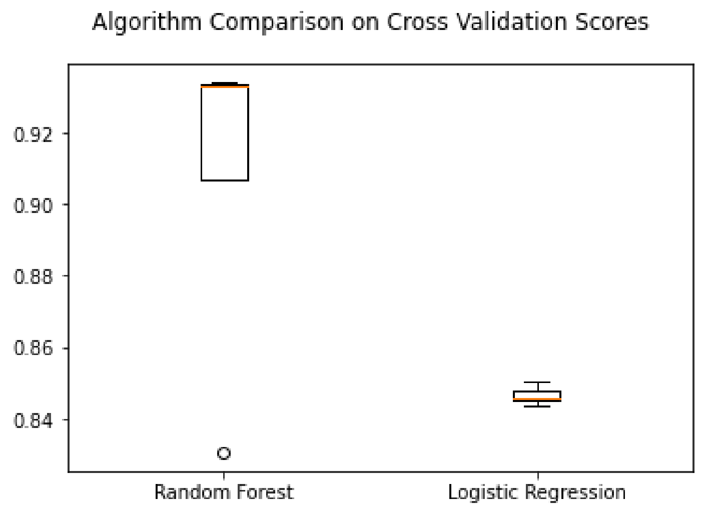


Figure 3.26: Cross validation scores.

The Logistic Regression model appears to be more stable, but Random Forest achieved significantly higher scores. However, we observe one score that is far off. Printing the cross validation scores, we see a lower score of 0.83 which is still a decent score.

Random forest scores: [0.83064679 0.90683122 0.93393696 0.93295654 0.93376204]

Chapter 4

Conclusions and Future Work

Based on the above performance results, the Random Forest algorithm trained on oversampled data outperformed any Logistic Regression model, with an accuracy of 85% and average cross validation score 90%. This score also outperforms the accuracies achieved by researchers that used Naive Bayes and Knn algorithms. Different approaches can be taken in the future in an attempt to achieve better results. Instead of feature extraction (PCA), feature selection can be implemented instead (e.g. SelectKBest). Also, the treatment of outliers can be revised. Some outliers can be legitimate observations and it is important to investigate their nature. For the evaluation of the models, it is also possible to check for overfitting by having a look at the accuracy scores over the training and test sets. Lastly, we can also test the ability of deep learning models to solving the problem of rainfall prediction.

References

AnalyticsVidhya (2017), 'Class imbalance: Handling imbalanced data using python'.
URL: *https://www.analyticsvidhya.com/blog/2017/03/imbalanced-data-classification/*

Australian Government Bureau of Meteorology (n.d.).
URL: *http://www.bom.gov.au/*

Brownlee, J. (2020), 'How to use standardscaler and minmaxscaler transforms in python'.
URL: *https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/*

Brownlee, J. (2021*a*), 'How to use roc curves and precision-recall curves for classification in python'.
URL: *https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/*

Brownlee, J. (2021*b*), 'Smote for imbalanced classification with python'.
URL: *https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/*

Gupta, D. and Ghose, U. (2015), A comparative study of classification algorithms for forecasting rainfall, *in* '2015 4th International Conference on Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions)', pp. 1–6.

Khanna, C. (2020), 'What and why behind fit$_{t}ransform()vstransform()inscikit - learnnbsp;!'$.
URL:*https://towardsdatascience.com/what-and-why-behind-fit-transform-vs-transform-in-scikit-learn-78f915c*

MachineLearningMastery (2021), 'Failure of classification accuracy for imbalanced class distributions'.
URL: *https://machinelearningmastery.com/failure-of-accuracy-for-imbalanced-class-distributions/*

Nicholls, N. (2001), 'Atmospheric and climatic hazards: Improved monitoring and prediction for disaster mitigation. natural hazards'.

Reuters (2022), 'Sydney faces more rain as death toll from australian floods rises'.
URL: *https://edition.cnn.com/2022/03/06/asia/sydney-australia-flood-intl-hnk/index.html*

Sharma, K. K., Verdon-Kidd, D. C. and Magee, A. D. (2021), 'A decision tree approach to identify predictors of extreme rainfall events – a case study for the fiji islands', *Weather and Climate Extremes* **34**, 100405.
URL: *https://www.sciencedirect.com/science/article/pii/S221209472100092X*

Sharmila (2020), 'Mechanisms of multiyear variations of northern australia wet-season rainfall'.

Wand (n.d.).
URL: *https://wand.net.nz/pubs/22/html/node34.html*

WHO (2003), ‘Climate change and human health: Risks and re- sponses.’.