Extensive Analysis - EDA + Preprocessing + FE + Modelling

"Some people walk in the rain, others just get wet."

The above quote belongs to **Roger Miller**. He was an American singer-songwriter, musician, and actor. He was widely known for his novelty songs and his chart-topping country and pop hits. He began his musical career as a songwriter in the late 1950s. He later began a recording career and reached the peak of his fame in the mid-1960s, continuing to record and tour into the 1990s.

(Source : https://en.wikipedia.org/wiki/Roger_Miller (https://en.wikipedia.org/wiki/Roger_Miller))

Rains are essential part of our lives. Clouds give the gift of rains to humans. Weather department tries to forecast when will it rain. So, I try to predict whether it will rain in Australia tomorrow or not.

I hope you find this kernel useful and your UPVOTES would be very much appreciated

Hence, in this kernel, I implement Logistic Regression with Python and Scikit-Learn and build a classifier to predict whether or not it will rain tomorrow in Australia. I train a binary classification model using Logistic Regression. I have used the **Rain in Australia** dataset for this project.

Table of Contents

The table of contents for this project is as follows:-

1. The problem statement

In this kernel, we will try to answer the question that whether or not it will rain tomorrow in Australia. We implement Logistic Regression with Python and Scikit-Learn.

To answer the question, we build a classifier to predict whether or not it will rain tomorrow in Australia. We train a binary classification model using Logistic Regression. I have used the **Rain in Australia** dataset for this project.

So, let's get started.

2. Import libraries

The first step in building the model is to import the necessary libraries.

In [1]:

```
# This Python 3 environment comes with many helpful analytics libraries ins
talled
# It is defined by the kaggle/python docker image: https://github.com/kaggl
e/docker-python
# For example, here's several helpful packages to load in

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# import libraries for plotting
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Input data files are available in the "../input/" directory.
# For example, running this (by clicking run or pressing Shift+Enter) will
 list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# Any results you write to the current directory are saved as output.
```

```
/kaggle/input/weather-dataset-rattle-package/weatherAUS.csv
```

In [2]:

```
import warnings

warnings.filterwarnings('ignore')
```

3. Import dataset

The next step is to import the dataset.

In [3]:

```
data = '/kaggle/input/weather-dataset-rattle-package/weatherAUS.csv'

df = pd.read_csv(data)
```

4. Exploratory data analysis

- We have imported the data.
- Now, its time to explore the data to gain insights about it.

View dimensions of dataset

In [4]:

```
df.shape
```

Out[4]:

```
(142193, 24)
```

We can see that there are 142193 instances and 24 variables in the data set.

Preview the dataset

In [5]:

```
df.head()
```

Out[5]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	Winc
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0

5 rows × 24 columns

View column names

In [6]:

```
col_names = df.columns

col_names
```

Out[6]:

```
Index(['Date', 'Location', 'MinTemp', 'MaxTemp', 'Rainfall', 'Evaporat
ion',
       'Sunshine', 'WindGustDir', 'WindGustSpeed', 'WindDir9am', 'Wind
Dir3pm',
       'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm',
       'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9a
m',
       'Temp3pm', 'RainToday', 'RISK_MM', 'RainTomorrow'],
      dtype='object')
```

Drop RISK_MM variable

It is given in the dataset description, that we should drop the `RISK_MM` feature variable from the dataset description. So, we should drop it as follows-

In [7]:

```
df.drop(['RISK_MM'], axis=1, inplace=True)
```

View summary of dataset

In [8]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 142193 entries, 0 to 142192
Data columns (total 23 columns):
Date             142193 non-null object
Location         142193 non-null object
MinTemp          141556 non-null float64
MaxTemp          141871 non-null float64
Rainfall         140787 non-null float64
Evaporation      81350 non-null float64
Sunshine         74377 non-null float64
WindGustDir      132863 non-null object
WindGustSpeed    132923 non-null float64
WindDir9am       132180 non-null object
WindDir3pm       138415 non-null object
WindSpeed9am     140845 non-null float64
WindSpeed3pm     139563 non-null float64
Humidity9am      140419 non-null float64
Humidity3pm      138583 non-null float64
Pressure9am      128179 non-null float64
Pressure3pm      128212 non-null float64
Cloud9am         88536 non-null float64
Cloud3pm         85099 non-null float64
Temp9am          141289 non-null float64
Temp3pm          139467 non-null float64
RainToday        140787 non-null object
RainTomorrow     142193 non-null object
dtypes: float64(16), object(7)
memory usage: 25.0+ MB
```

Comment

- We can see that the dataset contains mixture of categorical and numerical variables.
- Categorical variables have data type `object`.
- Numerical variables have data type `float64`.
- Also, there are some missing values in the dataset. We will explore it later.

View statistical properties of dataset

In [9]:

```
df.describe()
```

Out[9]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	W
count	141556.000000	141871.000000	140787.000000	81350.000000	74377.000000	1
mean	12.186400	23.226784	2.349974	5.469824	7.624853	3
std	6.403283	7.117618	8.465173	4.188537	3.781525	1
min	-8.500000	-4.800000	0.000000	0.000000	0.000000	6
25%	7.600000	17.900000	0.000000	2.600000	4.900000	3
50%	12.000000	22.600000	0.000000	4.800000	8.500000	3
75%	16.800000	28.200000	0.800000	7.400000	10.600000	4
max	33.900000	48.100000	371.000000	145.000000	14.500000	1

Important points to note

- The above command `df.describe()` helps us to view the statistical properties of numerical variables. It excludes character variables.
- If we want to view the statistical properties of character variables, we should run the following command -

```
df.describe(include=['object'])
```

- If we want to view the statistical properties of all the variables, we should run the following command -

```
df.describe(include='all')
```

5. Univariate Analysis

Explore `RainTomorrow` target variable

Check for missing values

In [10]:

```
df['RainTomorrow'].isnull().sum()
```

Out[10]:

```
0
```

We can see that there are no missing values in the `RainTomorrow` target variable.

View number of unique values

In [11]:
```
df['RainTomorrow'].nunique()
```

Out[11]:
```
2
```

We can see that the number of unique values in `RainTomorrow` variable is 2.

View the unique values

In [12]:
```
df['RainTomorrow'].unique()
```

Out[12]:
```
array(['No', 'Yes'], dtype=object)
```

The two unique values are `No` and `Yes`.

View the frequency distribution of values

In [13]:
```
df['RainTomorrow'].value_counts()
```

Out[13]:
```
No     110316
Yes     31877
Name: RainTomorrow, dtype: int64
```

View percentage of frequency distribution of values

In [14]:

```
df['RainTomorrow'].value_counts()/len(df)
```

Out[14]:

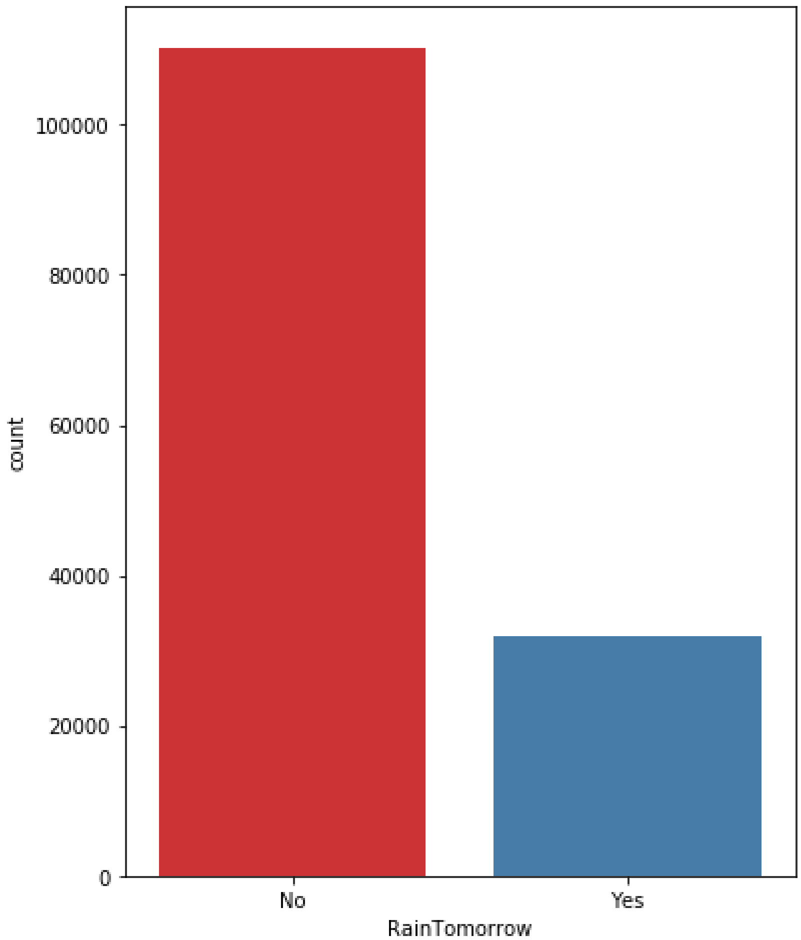
```
No     0.775819
Yes    0.224181
Name: RainTomorrow, dtype: float64
```

Comment

- We can see that out of the total number of `RainTomorrow` values, `No` appears 77.58% times and `Yes` appears 22.42% times.

Visualize frequency distribution of `RainTomorrow` variable

In [15]:
```
f, ax = plt.subplots(figsize=(6, 8))
ax = sns.countplot(x="RainTomorrow", data=df, palette="Set1")
plt.show()
```



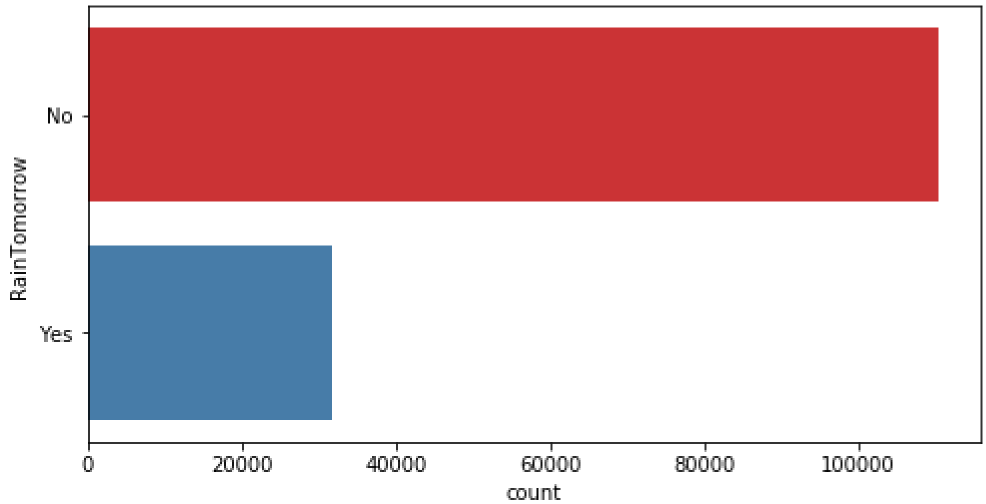
Interpretation

- The above univariate plot confirms our findings that -
  - The `No` variable have 110316 entries, and
  - The `Yes` variable have 31877 entries.

We can plot the bars horizontally as follows :

In [16]:

```
f, ax = plt.subplots(figsize=(8, 4))
ax = sns.countplot(y="RainTomorrow", data=df, palette="Set1")
plt.show()
```



Findings of Univariate Analysis

- The number of unique values in `RainTomorrow` variable is 2.
- The two unique values are `No` and `Yes`.
- Out of the total number of `RainTomorrow` values, `No` appears 77.58% times and `Yes` appears 22.42% times.
- The univariate plot confirms our findings that –
  - The `No` variable have 110316 entries, and
  - The `Yes` variable have 31877 entries.

6. Bivariate Analysis

Types of variables

In this section, I segregate the dataset into categorical and numerical variables. There are a mixture of categorical and numerical variables in the dataset. Categorical variables have data type object. Numerical variables have data type float64.

First of all, I will find categorical variables.

Explore Categorical Variables

In [17]:

```
# find categorical variables

categorical = [var for var in df.columns if df[var].dtype=='O']

print('There are {} categorical variables\n'.format(len(categorical)))

print('The categorical variables are :', categorical)
```

```
There are 7 categorical variables

The categorical variables are : ['Date', 'Location', 'WindGustDir', 'W
indDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']
```

In [18]:

```
# view the categorical variables

df[categorical].head()
```

Out[18]:

	Date	Location	WindGustDir	WindDir9am	WindDir3pm	RainToday	RainTomorrow
0	2008-12-01	Albury	W	W	WNW	No	No
1	2008-12-02	Albury	WNW	NNW	WSW	No	No
2	2008-12-03	Albury	WSW	W	WSW	No	No
3	2008-12-04	Albury	NE	SE	E	No	No
4	2008-12-05	Albury	W	ENE	NW	No	No

Summary of categorical variables

- There is a date variable. It is denoted by `Date` column.
- There are 6 categorical variables. These are given by `Location`, `WindGustDir`, `WindDir9am`, `WindDir3pm`, `RainToday` and `RainTomorrow`.
- There are two binary categorical variables - `RainToday` and `RainTomorrow`.
- `RainTomorrow` is the target variable.

Explore problems within categorical variables

First, I will explore the categorical variables.

Missing values in categorical variables

In [19]:

```
# check missing values in categorical variables

df[categorical].isnull().sum()
```

Out[19]:

```
Date                0
Location            0
WindGustDir      9330
WindDir9am      10013
WindDir3pm       3778
RainToday        1406
RainTomorrow        0
dtype: int64
```

In [20]:

```
# print categorical variables containing missing values

cat1 = [var for var in categorical if df[var].isnull().sum()!=0]

print(df[cat1].isnull().sum())
```

```
WindGustDir     9330
WindDir9am     10013
WindDir3pm      3778
RainToday       1406
dtype: int64
```

We can see that there are only 4 categorical variables in the dataset which contains missing values. These are `WindGustDir`, `WindDir9am`, `WindDir3pm` and `RainToday`.

Frequency count of categorical variables

Now, I will check the frequency counts of categorical variables.

In [21]:

```
# view frequency of categorical variables

for var in categorical:

    print(df[var].value_counts())
```

```
2014-07-06    49
2013-12-21    49
2014-03-25    49
2013-10-21    49
2013-04-28    49
              ..
2008-01-14     1
2008-01-10     1
2008-01-01     1
2007-11-15     1
2007-11-20     1
Name: Date, Length: 3436, dtype: int64
Canberra            3418
Sydney              3337
Perth               3193
Darwin              3192
Hobart              3188
Brisbane            3161
Adelaide            3090
Bendigo             3034
Townsville          3033
AliceSprings        3031
MountGambier        3030
Ballarat            3028
Launceston          3028
Albany              3016
Albury              3011
MelbourneAirport    3009
PerthAirport        3009
Mildura             3007
SydneyAirport       3005
Nuriootpa           3002
Sale                3000
Watsonia            2999
Tuggeranong         2998
Portland            2996
Woomera             2990
Cairns              2988
Cobar               2988
Wollongong          2983
```

```
GoldCoast           2980
WaggaWagga          2976
Penrith             2964
NorfolkIsland       2964
Newcastle           2955
SalmonGums          2955
CoffsHarbour        2953
Witchcliffe         2952
Richmond            2951
Dartmoor            2943
NorahHead           2929
BadgerysCreek       2928
MountGinini         2907
Moree               2854
Walpole             2819
PearceRAAF          2762
Williamtown         2553
Melbourne           2435
Nhil                1569
Katherine           1559
Uluru               1521
Name: Location, dtype: int64
W      9780
SE     9309
E      9071
N      9033
SSE    8993
S      8949
WSW    8901
SW     8797
SSW    8610
WNW    8066
NW     8003
ENE    7992
ESE    7305
NE     7060
NNW    6561
NNE    6433
Name: WindGustDir, dtype: int64
N      11393
SE      9162
```

```
E       9024
SSE     8966
NW      8552
S       8493
W       8260
SW      8237
NNE     7948
NNW     7840
ENE     7735
ESE     7558
NE      7527
SSW     7448
WNW     7194
WSW     6843
Name: WindDir9am, dtype: int64
SE     10663
W       9911
S       9598
WSW     9329
SW      9182
SSE     9142
N       8667
WNW     8656
NW      8468
ESE     8382
E       8342
NE      8164
SSW     8010
NNW     7733
ENE     7724
NNE     6444
Name: WindDir3pm, dtype: int64
No     109332
Yes     31455
Name: RainToday, dtype: int64
No     110316
Yes     31877
Name: RainTomorrow, dtype: int64
```

In [22]:

```
# view frequency distribution of categorical variables

for var in categorical:

    print(df[var].value_counts()/np.float(len(df)))
```

```
2014-07-06    0.000345
2013-12-21    0.000345
2014-03-25    0.000345
2013-10-21    0.000345
2013-04-28    0.000345
                ...
2008-01-14    0.000007
2008-01-10    0.000007
2008-01-01    0.000007
2007-11-15    0.000007
2007-11-20    0.000007
Name: Date, Length: 3436, dtype: float64
Canberra            0.024038
Sydney              0.023468
Perth               0.022455
Darwin              0.022448
Hobart              0.022420
Brisbane            0.022230
Adelaide            0.021731
Bendigo             0.021337
Townsville          0.021330
AliceSprings        0.021316
MountGambier        0.021309
Ballarat            0.021295
Launceston          0.021295
Albany              0.021211
Albury              0.021175
MelbourneAirport    0.021161
PerthAirport        0.021161
Mildura             0.021147
SydneyAirport       0.021133
Nuriootpa           0.021112
Sale                0.021098
Watsonia            0.021091
Tuggeranong         0.021084
Portland            0.021070
Woomera             0.021028
Cairns              0.021014
Cobar               0.021014
Wollongong          0.020979
```

```
GoldCoast          0.020957
WaggaWagga         0.020929
Penrith            0.020845
NorfolkIsland      0.020845
Newcastle          0.020782
SalmonGums         0.020782
CoffsHarbour       0.020768
Witchcliffe        0.020761
Richmond           0.020753
Dartmoor           0.020697
NorahHead          0.020599
BadgerysCreek      0.020592
MountGinini        0.020444
Moree              0.020071
Walpole            0.019825
PearceRAAF         0.019424
Williamtown        0.017954
Melbourne          0.017125
Nhil               0.011034
Katherine          0.010964
Uluru              0.010697
Name: Location, dtype: float64
W      0.068780
SE     0.065467
E      0.063794
N      0.063526
SSE    0.063245
S      0.062936
WSW    0.062598
SW     0.061867
SSW    0.060552
WNW    0.056726
NW     0.056283
ENE    0.056205
ESE    0.051374
NE     0.049651
NNW    0.046142
NNE    0.045241
Name: WindGustDir, dtype: float64
N      0.080123
SE     0.064434
```

```
E      0.063463
SSE    0.063055
NW     0.060144
S      0.059729
W      0.058090
SW     0.057928
NNE    0.055896
NNW    0.055136
ENE    0.054398
ESE    0.053153
NE     0.052935
SSW    0.052380
WNW    0.050593
WSW    0.048125
Name: WindDir9am, dtype: float64
SE     0.074990
W      0.069701
S      0.067500
WSW    0.065608
SW     0.064574
SSE    0.064293
N      0.060952
WNW    0.060875
NW     0.059553
ESE    0.058948
E      0.058667
NE     0.057415
SSW    0.056332
NNW    0.054384
ENE    0.054321
NNE    0.045319
Name: WindDir3pm, dtype: float64
No     0.768899
Yes    0.221213
Name: RainToday, dtype: float64
No     0.775819
Yes    0.224181
Name: RainTomorrow, dtype: float64
```

Number of labels: cardinality

The number of labels within a categorical variable is known as **cardinality**. A high number of labels within a variable is known as **high cardinality**. High cardinality may pose some serious problems in the machine learning model. So, I will check for high cardinality.

In [23]:

```
# check for cardinality in categorical variables

for var in categorical:

    print(var, ' contains ', len(df[var].unique()), ' labels')
```

```
Date  contains  3436  labels
Location  contains  49  labels
WindGustDir  contains  17  labels
WindDir9am  contains  17  labels
WindDir3pm  contains  17  labels
RainToday  contains  3  labels
RainTomorrow  contains  2  labels
```

We can see that there is a `Date` variable which needs to be preprocessed. I will do preprocessing in the following section.

All the other variables contain relatively smaller number of variables.

Feature Engineering of Date Variable

In [24]:

```
df['Date'].dtypes
```

Out[24]:

```
dtype('O')
```

We can see that the data type of `Date` variable is object. I will parse the date currently coded as object into datetime format.

In [25]:

```
# parse the dates, currently coded as strings, into datetime format

df['Date'] = pd.to_datetime(df['Date'])
```

In [26]:

```
# extract year from date

df['Year'] = df['Date'].dt.year

df['Year'].head()
```

Out[26]:

```
0    2008
1    2008
2    2008
3    2008
4    2008
Name: Year, dtype: int64
```

In [27]:

```
# extract month from date

df['Month'] = df['Date'].dt.month

df['Month'].head()
```

Out[27]:

```
0    12
1    12
2    12
3    12
4    12
Name: Month, dtype: int64
```

In [28]:

```python
# extract day from date

df['Day'] = df['Date'].dt.day

df['Day'].head()
```

Out[28]:

```
0    1
1    2
2    3
3    4
4    5
Name: Day, dtype: int64
```

In [29]:

```python
# again view the summary of dataset

df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 142193 entries, 0 to 142192
Data columns (total 26 columns):
Date             142193 non-null datetime64[ns]
Location         142193 non-null object
MinTemp          141556 non-null float64
MaxTemp          141871 non-null float64
Rainfall         140787 non-null float64
Evaporation      81350 non-null float64
Sunshine         74377 non-null float64
WindGustDir      132863 non-null object
WindGustSpeed    132923 non-null float64
WindDir9am       132180 non-null object
WindDir3pm       138415 non-null object
WindSpeed9am     140845 non-null float64
WindSpeed3pm     139563 non-null float64
Humidity9am      140419 non-null float64
Humidity3pm      138583 non-null float64
Pressure9am      128179 non-null float64
Pressure3pm      128212 non-null float64
Cloud9am         88536 non-null float64
Cloud3pm         85099 non-null float64
Temp9am          141289 non-null float64
Temp3pm          139467 non-null float64
RainToday        140787 non-null object
RainTomorrow     142193 non-null object
Year             142193 non-null int64
Month            142193 non-null int64
Day              142193 non-null int64
dtypes: datetime64[ns](1), float64(16), int64(3), object(6)
memory usage: 28.2+ MB
```

We can see that there are three additional columns created from `Date` variable. Now, I will drop the original `Date` variable from the dataset.

In [30]:

```
# drop the original Date variable

df.drop('Date', axis=1, inplace = True)
```

In [31]:

```
# preview the dataset again

df.head()
```

Out[31]:

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpe
0	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0
1	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0
2	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0
3	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0
4	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0

5 rows × 25 columns

Now, we can see that the `Date` variable has been removed from the dataset.

Explore Categorical Variables one by one

Now, I will explore the categorical variables one by one.

In [32]:

```
# find categorical variables

categorical = [var for var in df.columns if df[var].dtype=='O']

print('There are {} categorical variables\n'.format(len(categorical)))

print('The categorical variables are :', categorical)
```

```
There are 6 categorical variables

The categorical variables are : ['Location', 'WindGustDir', 'WindDir9a
m', 'WindDir3pm', 'RainToday', 'RainTomorrow']
```

We can see that there are 6 categorical variables in the dataset. The `Date` variable has been removed. First, I will check missing values in categorical variables.

In [33]:

```
# check for missing values in categorical variables

df[categorical].isnull().sum()
```

Out[33]:

```
Location            0
WindGustDir      9330
WindDir9am      10013
WindDir3pm       3778
RainToday        1406
RainTomorrow        0
dtype: int64
```

We can see that `WindGustDir`, `WindDir9am`, `WindDir3pm`, `RainToday` variables contain missing values. I will explore these variables one by one.

Explore `Location` variable

In [34]:

```
# print number of labels in Location variable

print('Location contains', len(df.Location.unique()), 'labels')
```

```
Location contains 49 labels
```

In [35]:

```
# check labels in location variable

df.Location.unique()
```

Out[35]:

```
array(['Albury', 'BadgerysCreek', 'Cobar', 'CoffsHarbour', 'Moree',
       'Newcastle', 'NorahHead', 'NorfolkIsland', 'Penrith', 'Richmon
d',
       'Sydney', 'SydneyAirport', 'WaggaWagga', 'Williamtown',
       'Wollongong', 'Canberra', 'Tuggeranong', 'MountGinini', 'Ballar
at',
       'Bendigo', 'Sale', 'MelbourneAirport', 'Melbourne', 'Mildura',
       'Nhil', 'Portland', 'Watsonia', 'Dartmoor', 'Brisbane', 'Cairn
s',
       'GoldCoast', 'Townsville', 'Adelaide', 'MountGambier', 'Nurioot
pa',
       'Woomera', 'Albany', 'Witchcliffe', 'PearceRAAF', 'PerthAirpor
t',
       'Perth', 'SalmonGums', 'Walpole', 'Hobart', 'Launceston',
       'AliceSprings', 'Darwin', 'Katherine', 'Uluru'], dtype=object)
```

In [36]:

```
# check frequency distribution of values in Location variable

df.Location.value_counts()
```

Out[36]:

```
Canberra            3418
Sydney              3337
Perth               3193
Darwin              3192
Hobart              3188
Brisbane            3161
Adelaide            3090
Bendigo             3034
Townsville          3033
AliceSprings        3031
MountGambier        3030
Ballarat            3028
Launceston          3028
Albany              3016
Albury              3011
MelbourneAirport    3009
PerthAirport        3009
Mildura             3007
SydneyAirport       3005
Nuriootpa           3002
Sale                3000
Watsonia            2999
Tuggeranong         2998
Portland            2996
Woomera             2990
Cairns              2988
Cobar               2988
Wollongong          2983
GoldCoast           2980
WaggaWagga          2976
Penrith             2964
NorfolkIsland       2964
Newcastle           2955
SalmonGums          2955
CoffsHarbour        2953
Witchcliffe         2952
Richmond            2951
Dartmoor            2943
NorahHead           2929
BadgerysCreek       2928
```

```
MountGinini         2907
Moree               2854
Walpole             2819
PearceRAAF          2762
Williamtown         2553
Melbourne           2435
Nhil                1569
Katherine           1559
Uluru               1521
Name: Location, dtype: int64
```

In [37]:

```
# let's do One Hot Encoding of Location variable
# get k-1 dummy variables after One Hot Encoding
# preview the dataset with head() method

pd.get_dummies(df.Location, drop_first=True).head()
```

Out[37]:

	Albany	Albury	AliceSprings	BadgerysCreek	Ballarat	Bendigo	Brisbane	Cairns	Canbe
0	0	1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	0	0
4	0	1	0	0	0	0	0	0	0

5 rows × 48 columns

Explore `WindGustDir` variable

In [38]:

```
# print number of labels in WindGustDir variable

print('WindGustDir contains', len(df['WindGustDir'].unique()), 'labels')
```

```
WindGustDir contains 17 labels
```

In [39]:

```
# check labels in WindGustDir variable

df['WindGustDir'].unique()
```

Out[39]:

```
array(['W', 'WNW', 'WSW', 'NE', 'NNW', 'N', 'NNE', 'SW', 'ENE', 'SSE',
       'S', 'NW', 'SE', 'ESE', nan, 'E', 'SSW'], dtype=object)
```

In [40]:

```
# check frequency distribution of values in WindGustDir variable

df.WindGustDir.value_counts()
```

Out[40]:

```
W      9780
SE     9309
E      9071
N      9033
SSE    8993
S      8949
WSW    8901
SW     8797
SSW    8610
WNW    8066
NW     8003
ENE    7992
ESE    7305
NE     7060
NNW    6561
NNE    6433
Name: WindGustDir, dtype: int64
```

In [41]:

```python
# let's do One Hot Encoding of WindGustDir variable
# get k-1 dummy variables after One Hot Encoding
# also add an additional dummy variable to indicate there was missing data
# preview the dataset with head() method

pd.get_dummies(df.WindGustDir, drop_first=True, dummy_na=True).head()
```

Out[41]:

	ENE	ESE	N	NE	NNE	NNW	NW	S	SE	SSE	SSW	SW	W	WNW	WSW	Na
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

In [42]:

```python
# sum the number of 1s per boolean variable over the rows of the dataset
# it will tell us how many observations we have for each category

pd.get_dummies(df.WindGustDir, drop_first=True, dummy_na=True).sum(axis=0
)
```

Out[42]:

```
ENE    7992
ESE    7305
N      9033
NE     7060
NNE    6433
NNW    6561
NW     8003
S      8949
SE     9309
SSE    8993
SSW    8610
SW     8797
W      9780
WNW    8066
WSW    8901
NaN    9330
dtype: int64
```

We can see that there are 9330 missing values in WindGustDir variable.

Explore `WindDir9am` variable

In [43]:

```python
# print number of labels in WindDir9am variable

print('WindDir9am contains', len(df['WindDir9am'].unique()), 'labels')
```

```
WindDir9am contains 17 labels
```

In [44]:

```
# check labels in WindDir9am variable

df['WindDir9am'].unique()
```

Out[44]:

```
array(['W', 'NNW', 'SE', 'ENE', 'SW', 'SSE', 'S', 'NE', nan, 'SSW',
'N',
       'WSW', 'ESE', 'E', 'NW', 'WNW', 'NNE'], dtype=object)
```

In [45]:

```
# check frequency distribution of values in WindDir9am variable

df['WindDir9am'].value_counts()
```

Out[45]:

```
N      11393
SE      9162
E       9024
SSE     8966
NW      8552
S       8493
W       8260
SW      8237
NNE     7948
NNW     7840
ENE     7735
ESE     7558
NE      7527
SSW     7448
WNW     7194
WSW     6843
Name: WindDir9am, dtype: int64
```

In [46]:

```
# let's do One Hot Encoding of WindDir9am variable
# get k-1 dummy variables after One Hot Encoding
# also add an additional dummy variable to indicate there was missing data
# preview the dataset with head() method

pd.get_dummies(df.WindDir9am, drop_first=True, dummy_na=True).head()
```

Out[46]:

	ENE	ESE	N	NE	NNE	NNW	NW	S	SE	SSE	SSW	SW	W	WNW	WSW	Na
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In [47]:

```
# sum the number of 1s per boolean variable over the rows of the dataset
# it will tell us how many observations we have for each category

pd.get_dummies(df.WindDir9am, drop_first=True, dummy_na=True).sum(axis=0)
```

Out[47]:

```
ENE     7735
ESE     7558
N      11393
NE      7527
NNE     7948
NNW     7840
NW      8552
S       8493
SE      9162
SSE     8966
SSW     7448
SW      8237
W       8260
WNW     7194
WSW     6843
NaN    10013
dtype: int64
```

We can see that there are 10013 missing values in the `WindDir9am` variable.

Explore `WindDir3pm` variable

In [48]:

```
# print number of labels in WindDir3pm variable

print('WindDir3pm contains', len(df['WindDir3pm'].unique()), 'labels')
```

```
WindDir3pm contains 17 labels
```

In [49]:

```
# check labels in WindDir3pm variable

df['WindDir3pm'].unique()
```

Out[49]:

```
array(['WNW', 'WSW', 'E', 'NW', 'W', 'SSE', 'ESE', 'ENE', 'NNW', 'SS
W',
       'SW', 'SE', 'N', 'S', 'NNE', nan, 'NE'], dtype=object)
```

In [50]:

```
# check frequency distribution of values in WindDir3pm variable

df['WindDir3pm'].value_counts()
```

Out[50]:

```
SE     10663
W       9911
S       9598
WSW     9329
SW      9182
SSE     9142
N       8667
WNW     8656
NW      8468
ESE     8382
E       8342
NE      8164
SSW     8010
NNW     7733
ENE     7724
NNE     6444
Name: WindDir3pm, dtype: int64
```

In [51]:

```python
# let's do One Hot Encoding of WindDir3pm variable
# get k-1 dummy variables after One Hot Encoding
# also add an additional dummy variable to indicate there was missing data
# preview the dataset with head() method

pd.get_dummies(df.WindDir3pm, drop_first=True, dummy_na=True).head()
```

Out[51]:

	ENE	ESE	N	NE	NNE	NNW	NW	S	SE	SSE	SSW	SW	W	WNW	WSW	Na
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

In [52]:

```
# sum the number of 1s per boolean variable over the rows of the dataset
# it will tell us how many observations we have for each category

pd.get_dummies(df.WindDir3pm, drop_first=True, dummy_na=True).sum(axis=0)
```

Out[52]:

```
ENE     7724
ESE     8382
N       8667
NE      8164
NNE     6444
NNW     7733
NW      8468
S       9598
SE     10663
SSE     9142
SSW     8010
SW      9182
W       9911
WNW     8656
WSW     9329
NaN     3778
dtype: int64
```

There are 3778 missing values in the `WindDir3pm` variable.

Explore `RainToday` variable

In [53]:

```
# print number of labels in RainToday variable

print('RainToday contains', len(df['RainToday'].unique()), 'labels')
```

```
RainToday contains 3 labels
```

In [54]:

```
# check labels in WindGustDir variable

df['RainToday'].unique()
```

Out[54]:

```
array(['No', 'Yes', nan], dtype=object)
```

In [55]:

```
# check frequency distribution of values in WindGustDir variable

df.RainToday.value_counts()
```

Out[55]:

```
No     109332
Yes     31455
Name: RainToday, dtype: int64
```

In [56]:

```
# let's do One Hot Encoding of RainToday variable
# get k-1 dummy variables after One Hot Encoding
# also add an additional dummy variable to indicate there was missing data
# preview the dataset with head() method

pd.get_dummies(df.RainToday, drop_first=True, dummy_na=True).head()
```

Out[56]:

	Yes	NaN
0	0	0
1	0	0
2	0	0
3	0	0
4	0	0

In [57]:

```
# sum the number of 1s per boolean variable over the rows of the dataset
# it will tell us how many observations we have for each category

pd.get_dummies(df.RainToday, drop_first=True, dummy_na=True).sum(axis=0)
```

Out[57]:

```
Yes    31455
NaN     1406
dtype: int64
```

There are 1406 missing values in the `RainToday` variable.

Explore Numerical Variables

In [58]:

```
# find numerical variables

numerical = [var for var in df.columns if df[var].dtype!='O']

print('There are {} numerical variables\n'.format(len(numerical)))

print('The numerical variables are :', numerical)
```

```
There are 19 numerical variables

The numerical variables are : ['MinTemp', 'MaxTemp', 'Rainfall', 'Evap
oration', 'Sunshine', 'WindGustSpeed', 'WindSpeed9am', 'WindSpeed3pm',
'Humidity9am', 'Humidity3pm', 'Pressure9am', 'Pressure3pm', 'Cloud9a
m', 'Cloud3pm', 'Temp9am', 'Temp3pm', 'Year', 'Month', 'Day']
```

In [59]:

```
# view the numerical variables

df[numerical].head()
```

Out[59]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	Wi
0	13.4	22.9	0.6	NaN	NaN	44.0	20.0	24
1	7.4	25.1	0.0	NaN	NaN	44.0	4.0	22
2	12.9	25.7	0.0	NaN	NaN	46.0	19.0	26
3	9.2	28.0	0.0	NaN	NaN	24.0	11.0	9.0
4	17.5	32.3	1.0	NaN	NaN	41.0	7.0	20

Summary of numerical variables

- There are 16 numerical variables.
- These are given by `MinTemp`, `MaxTemp`, `Rainfall`, `Evaporation`, `Sunshine`, `WindGustSpeed`, `WindSpeed9am`, `WindSpeed3pm`, `Humidity9am`, `Humidity3pm`, `Pressure9am`, `Pressure3pm`, `Cloud9am`, `Cloud3pm`, `Temp9am` and `Temp3pm`.
- All of the numerical variables are of continuous type.

Explore problems within numerical variables

Now, I will explore the numerical variables.

Missing values in numerical variables

In [60]:

```
# check missing values in numerical variables

df[numerical].isnull().sum()
```

Out[60]:

```
MinTemp            637
MaxTemp            322
Rainfall          1406
Evaporation      60843
Sunshine         67816
WindGustSpeed     9270
WindSpeed9am      1348
WindSpeed3pm      2630
Humidity9am       1774
Humidity3pm       3610
Pressure9am      14014
Pressure3pm      13981
Cloud9am         53657
Cloud3pm         57094
Temp9am            904
Temp3pm           2726
Year                 0
Month                0
Day                  0
dtype: int64
```

We can see that all the 16 numerical variables contain missing values.

Outliers in numerical variables

In [61]:

```
# view summary statistics in numerical variables

print(round(df[numerical].describe()),2)
```

```
        MinTemp   MaxTemp  Rainfall  Evaporation  Sunshine  WindGustSp
eed  \
count  141556.0  141871.0  140787.0      81350.0   74377.0       13292
3.0
mean       12.0      23.0       2.0          5.0       8.0           4
0.0
std         6.0       7.0       8.0          4.0       4.0           1
4.0
min        -8.0      -5.0       0.0          0.0       0.0
6.0
25%         8.0      18.0       0.0          3.0       5.0           3
1.0
50%        12.0      23.0       0.0          5.0       8.0           3
9.0
75%        17.0      28.0       1.0          7.0      11.0           4
8.0
max        34.0      48.0     371.0        145.0      14.0          13
5.0

       WindSpeed9am  WindSpeed3pm  Humidity9am  Humidity3pm  Pressure9
am  \
count      140845.0      139563.0     140419.0     138583.0     12817
9.0
mean           14.0          19.0         69.0         51.0       101
8.0
std             9.0           9.0         19.0         21.0
7.0
min             0.0           0.0          0.0          0.0        98
0.0
25%             7.0          13.0         57.0         37.0       101
3.0
50%            13.0          19.0         70.0         52.0       101
8.0
75%            19.0          24.0         83.0         66.0       102
2.0
max           130.0          87.0        100.0        100.0       104
1.0

       Pressure3pm  Cloud9am  Cloud3pm   Temp9am   Temp3pm      Year
\
```

```
count    128212.0   88536.0   85099.0  141289.0  139467.0  142193.0
mean       1015.0       4.0       5.0      17.0      22.0    2013.0
std           7.0       3.0       3.0       6.0       7.0       3.0
min         977.0       0.0       0.0      -7.0      -5.0    2007.0
25%        1010.0       1.0       2.0      12.0      17.0    2011.0
50%        1015.0       5.0       5.0      17.0      21.0    2013.0
75%        1020.0       7.0       7.0      22.0      26.0    2015.0
max        1040.0       9.0       9.0      40.0      47.0    2017.0

          Month       Day
count  142193.0  142193.0
mean        6.0      16.0
std         3.0       9.0
min         1.0       1.0
25%         3.0       8.0
50%         6.0      16.0
75%         9.0      23.0
max        12.0      31.0   2
```

On closer inspection, we can see that the `Rainfall`, `Evaporation`, `WindSpeed9am` and `WindSpeed3pm` columns may contain outliers.

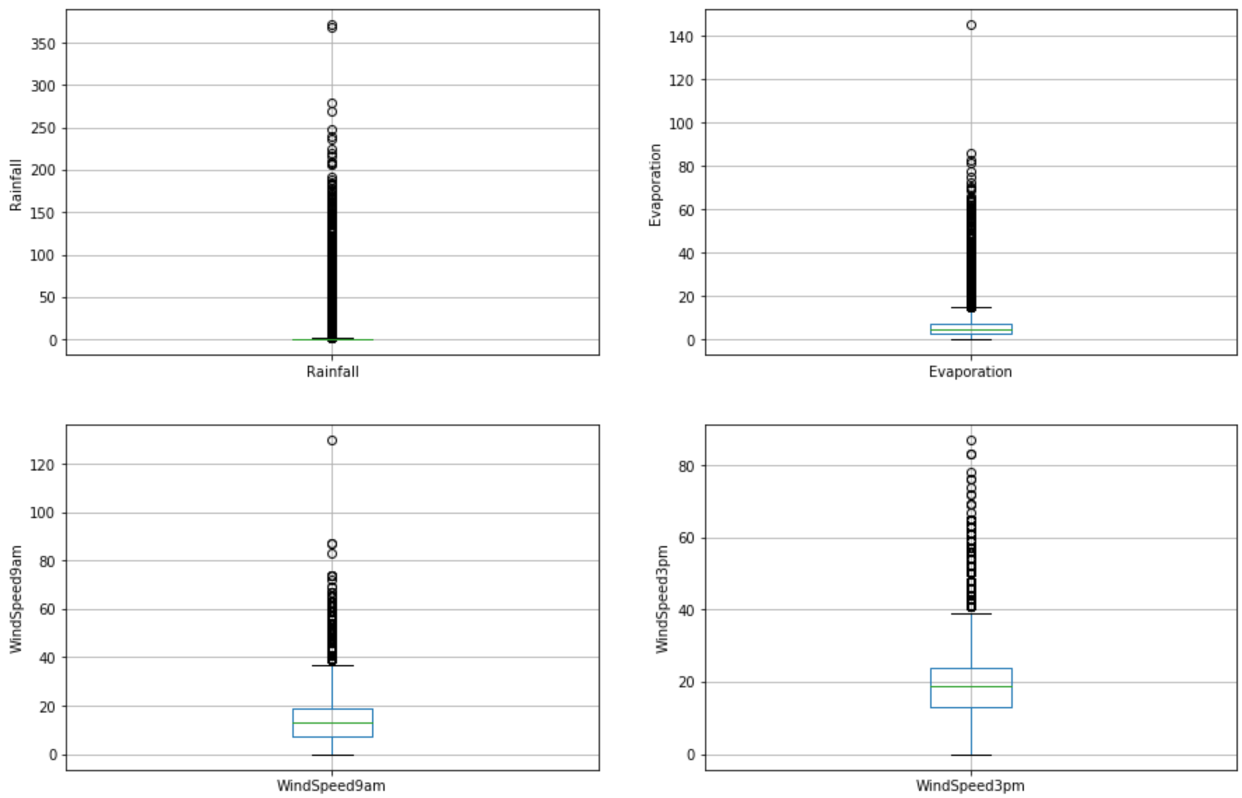
I will draw boxplots to visualise outliers in the above variables.

In [62]:

```
# draw boxplots to visualize outliers

plt.figure(figsize=(15,10))


plt.subplot(2, 2, 1)
fig = df.boxplot(column='Rainfall')
fig.set_title('')
fig.set_ylabel('Rainfall')


plt.subplot(2, 2, 2)
fig = df.boxplot(column='Evaporation')
fig.set_title('')
fig.set_ylabel('Evaporation')


plt.subplot(2, 2, 3)
fig = df.boxplot(column='WindSpeed9am')
fig.set_title('')
fig.set_ylabel('WindSpeed9am')


plt.subplot(2, 2, 4)
fig = df.boxplot(column='WindSpeed3pm')
fig.set_title('')
fig.set_ylabel('WindSpeed3pm')
```

Out[62]:

```
Text(0, 0.5, 'WindSpeed3pm')
```



The above boxplots confirm that there are lot of outliers in these variables.

Check the distribution of variables

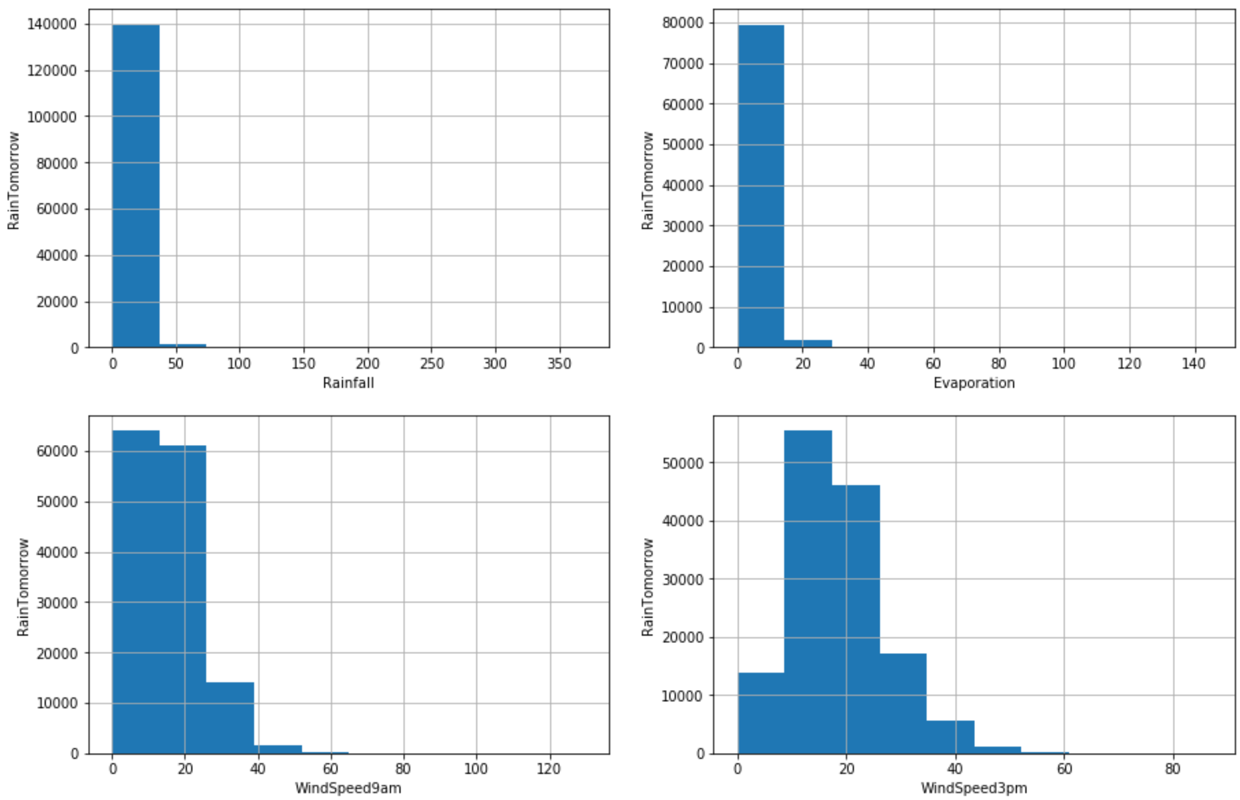
- Now, I will plot the histograms to check distributions to find out if they are normal or skewed.
- If the variable follows normal distribution, then I will do `Extreme Value Analysis` otherwise if they are skewed, I will find IQR (Interquantile range).

In [63]:

```
# plot histogram to check distribution

plt.figure(figsize=(15,10))


plt.subplot(2, 2, 1)
fig = df.Rainfall.hist(bins=10)
fig.set_xlabel('Rainfall')
fig.set_ylabel('RainTomorrow')


plt.subplot(2, 2, 2)
fig = df.Evaporation.hist(bins=10)
fig.set_xlabel('Evaporation')
fig.set_ylabel('RainTomorrow')


plt.subplot(2, 2, 3)
fig = df.WindSpeed9am.hist(bins=10)
fig.set_xlabel('WindSpeed9am')
fig.set_ylabel('RainTomorrow')


plt.subplot(2, 2, 4)
fig = df.WindSpeed3pm.hist(bins=10)
fig.set_xlabel('WindSpeed3pm')
fig.set_ylabel('RainTomorrow')
```

Out[63]:

```
Text(0, 0.5, 'RainTomorrow')
```



We can see that all the four variables are skewed. So, I will use interquantile range to find outliers.

In [64]:

```
# find outliers for Rainfall variable

IQR = df.Rainfall.quantile(0.75) - df.Rainfall.quantile(0.25)
Lower_fence = df.Rainfall.quantile(0.25) - (IQR * 3)
Upper_fence = df.Rainfall.quantile(0.75) + (IQR * 3)
print('Rainfall outliers are values < {lowerboundary} or > {upperboundar
y}'.format(lowerboundary=Lower_fence, upperboundary=Upper_fence))
```

```
Rainfall outliers are values < -2.4000000000000004 or > 3.2
```

For `Rainfall`, the minimum and maximum values are 0.0 and 371.0. So, the outliers are values > 3.2.

In [65]:

```
# find outliers for Evaporation variable

IQR = df.Evaporation.quantile(0.75) - df.Evaporation.quantile(0.25)
Lower_fence = df.Evaporation.quantile(0.25) - (IQR * 3)
Upper_fence = df.Evaporation.quantile(0.75) + (IQR * 3)
print('Evaporation outliers are values < {lowerboundary} or > {upperboundary}'.format(lowerboundary=Lower_fence, upperboundary=Upper_fence))
```

```
Evaporation outliers are values < -11.800000000000002 or > 21.800000000000004
```

For `Evaporation`, the minimum and maximum values are 0.0 and 145.0. So, the outliers are values > 21.8.

In [66]:

```
# find outliers for WindSpeed9am variable

IQR = df.WindSpeed9am.quantile(0.75) - df.WindSpeed9am.quantile(0.25)
Lower_fence = df.WindSpeed9am.quantile(0.25) - (IQR * 3)
Upper_fence = df.WindSpeed9am.quantile(0.75) + (IQR * 3)
print('WindSpeed9am outliers are values < {lowerboundary} or > {upperboundary}'.format(lowerboundary=Lower_fence, upperboundary=Upper_fence))
```

```
WindSpeed9am outliers are values < -29.0 or > 55.0
```

For `WindSpeed9am`, the minimum and maximum values are 0.0 and 130.0. So, the outliers are values > 55.0.

In [67]:
```
# find outliers for WindSpeed3pm variable

IQR = df.WindSpeed3pm.quantile(0.75) - df.WindSpeed3pm.quantile(0.25)
Lower_fence = df.WindSpeed3pm.quantile(0.25) - (IQR * 3)
Upper_fence = df.WindSpeed3pm.quantile(0.75) + (IQR * 3)
print('WindSpeed3pm outliers are values < {lowerboundary} or > {upperboundary}'.format(lowerboundary=Lower_fence, upperboundary=Upper_fence))
```

```
WindSpeed3pm outliers are values < -20.0 or > 57.0
```

For `WindSpeed3pm`, the minimum and maximum values are 0.0 and 87.0. So, the outliers are values > 57.0.

7. Multivariate Analysis

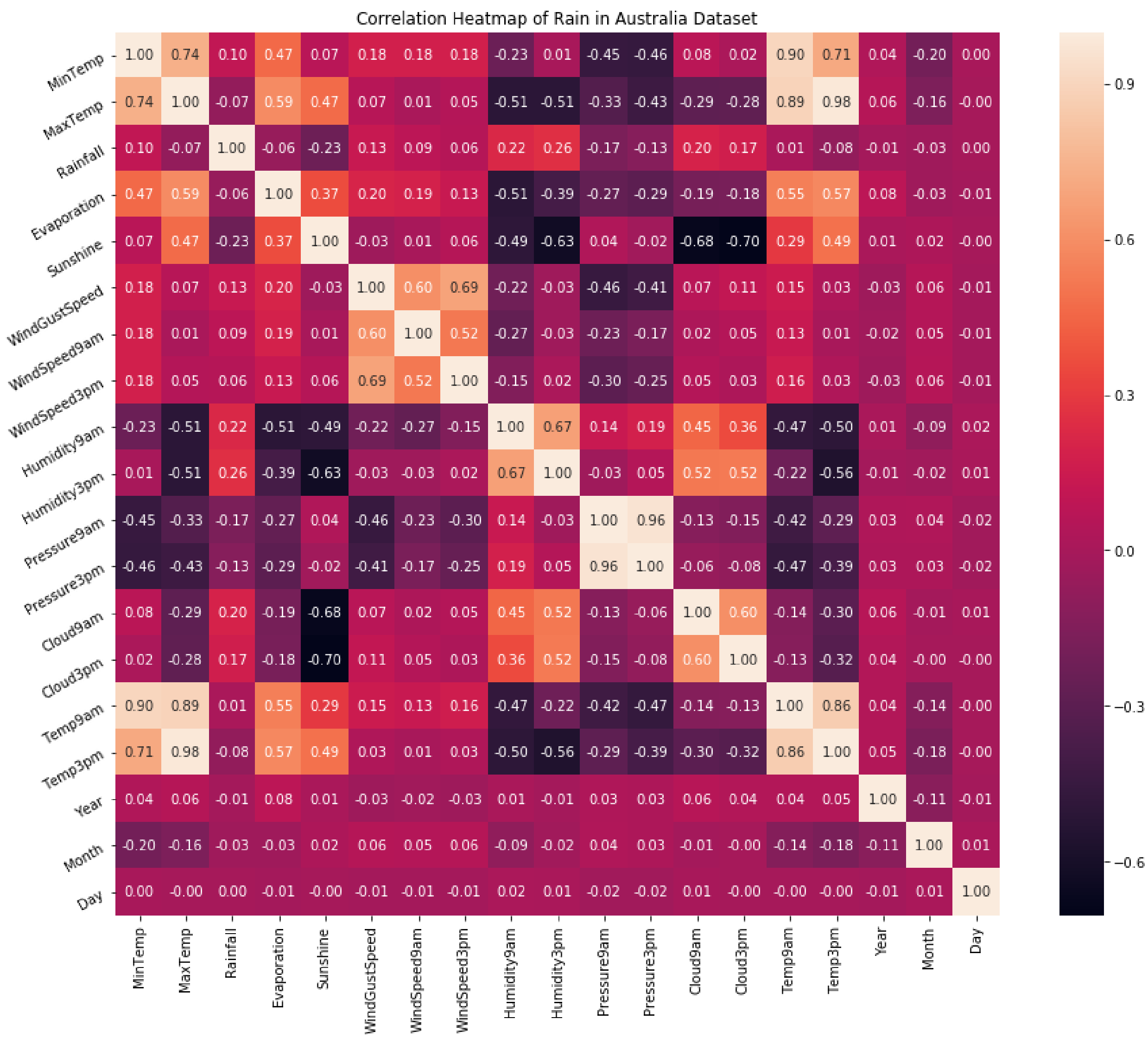
- An important step in EDA is to discover patterns and relationships between variables in the dataset.
- I will use heat map and pair plot to discover the patterns and relationships in the dataset.
- First of all, I will draw a heat map.

In [68]:
```
correlation = df.corr()
```

Heat Map

In [69]:

```python
plt.figure(figsize=(16,12))
plt.title('Correlation Heatmap of Rain in Australia Dataset')
ax = sns.heatmap(correlation, square=True, annot=True, fmt='.2f', linecol
or='white')
ax.set_xticklabels(ax.get_xticklabels(), rotation=90)
ax.set_yticklabels(ax.get_yticklabels(), rotation=30)
plt.show()
```



Interpretation

From the above correlation heat map, we can conclude that :-

- `MinTemp` and `MaxTemp` variables are highly positively correlated (correlation coefficient = 0.74).
- `MinTemp` and `Temp3pm` variables are also highly positively correlated (correlation coefficient = 0.71).
- `MinTemp` and `Temp9am` variables are strongly positively correlated (correlation coefficient = 0.90).
- `MaxTemp` and `Temp9am` variables are strongly positively correlated (correlation coefficient = 0.89).
- `MaxTemp` and `Temp3pm` variables are also strongly positively correlated (correlation coefficient = 0.98).
- `WindGustSpeed` and `WindSpeed3pm` variables are highly positively correlated (correlation coefficient = 0.69).
- `Pressure9am` and `Pressure3pm` variables are strongly positively correlated (correlation coefficient = 0.96).
- `Temp9am` and `Temp3pm` variables are strongly positively correlated (correlation coefficient = 0.86).

Pair Plot

First of all, I will define extract the variables which are highly positively correlated.

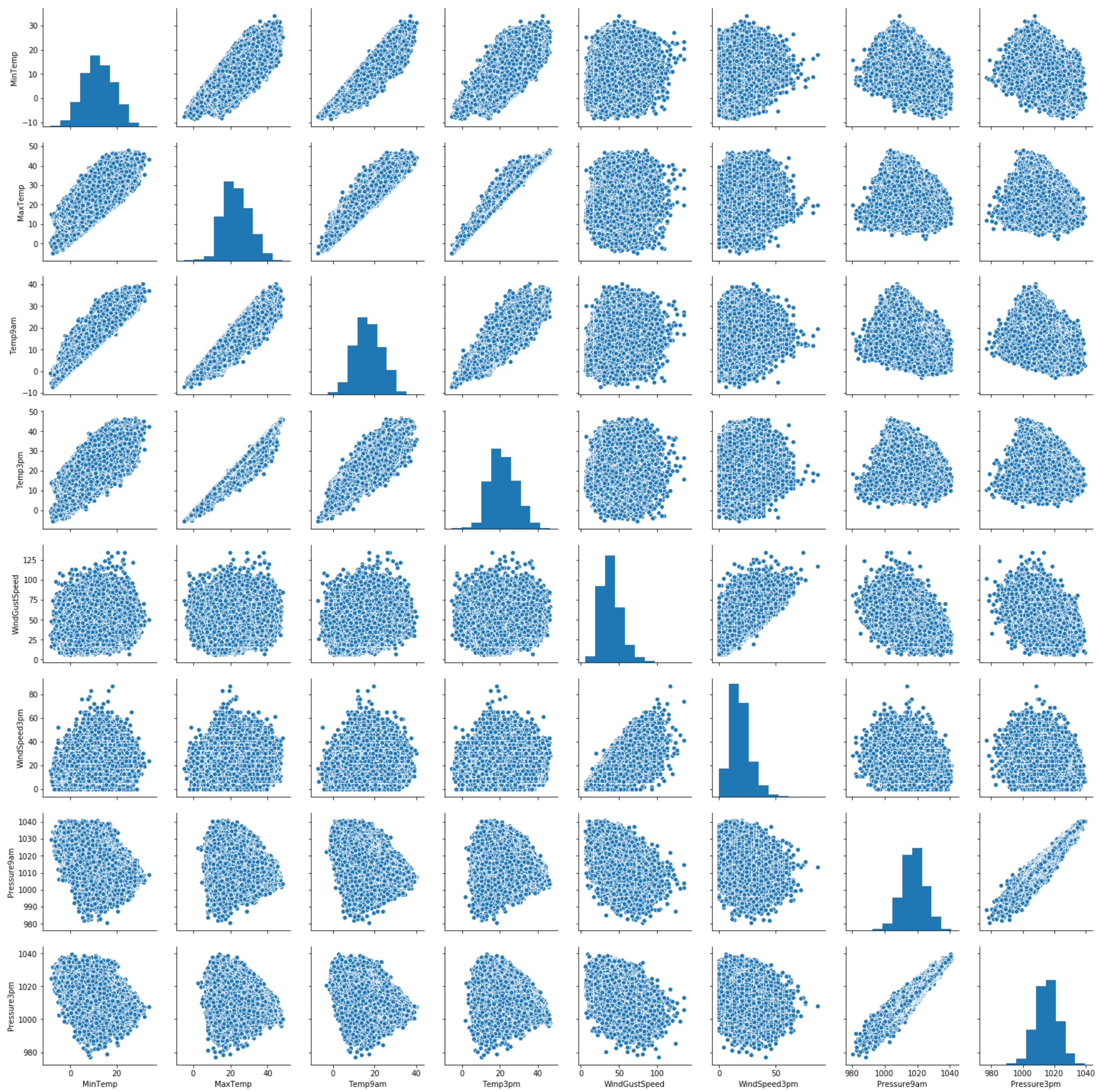
In [70]:

```
num_var = ['MinTemp', 'MaxTemp', 'Temp9am', 'Temp3pm', 'WindGustSpeed', 
'WindSpeed3pm', 'Pressure9am', 'Pressure3pm']
```

Now, I will draw pairplot to depict relationship between these variables.

In [71]:

```
sns.pairplot(df[num_var], kind='scatter', diag_kind='hist', palette='Rainbow')
plt.show()
```



Interpretation

- I have defined a variable `num_var` which consists of `MinTemp`, `MaxTemp`, `Temp9am`, `Temp3pm`, `WindGustSpeed`, `WindSpeed3pm`, `Pressure9am` and `Pressure3pm` variables.
- The above pair plot shows relationship between these variables.

8. Declare feature vector and target variable

In [72]:

```
X = df.drop(['RainTomorrow'], axis=1)

y = df['RainTomorrow']
```

9. Split data into separate training and test set

In [73]:

```
# split X and y into training and testing sets

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2
, random_state = 0)
```

In [74]:

```
# check the shape of X_train and X_test

X_train.shape, X_test.shape
```

Out[74]:

```
((113754, 24), (28439, 24))
```

10. Feature Engineering

Feature Engineering is the process of transforming raw data into useful features that help us to understand our model better and increase its predictive power. I will carry out feature engineering on different types of variables.

First, I will display the categorical and numerical variables again separately.

In [75]:

```
# check data types in X_train

X_train.dtypes
```

Out[75]:

```
Location          object
MinTemp          float64
MaxTemp          float64
Rainfall         float64
Evaporation      float64
Sunshine         float64
WindGustDir       object
WindGustSpeed    float64
WindDir9am        object
WindDir3pm        object
WindSpeed9am     float64
WindSpeed3pm     float64
Humidity9am      float64
Humidity3pm      float64
Pressure9am      float64
Pressure3pm      float64
Cloud9am         float64
Cloud3pm         float64
Temp9am          float64
Temp3pm          float64
RainToday         object
Year               int64
Month              int64
Day                int64
dtype: object
```

In [76]:

```
# display categorical variables

categorical = [col for col in X_train.columns if X_train[col].dtypes ==
'O']

categorical
```

Out[76]:

```
['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday']
```

In [77]:

```
# display numerical variables

numerical = [col for col in X_train.columns if X_train[col].dtypes != 'O'
]

numerical
```

Out[77]:

```
['MinTemp',
 'MaxTemp',
 'Rainfall',
 'Evaporation',
 'Sunshine',
 'WindGustSpeed',
 'WindSpeed9am',
 'WindSpeed3pm',
 'Humidity9am',
 'Humidity3pm',
 'Pressure9am',
 'Pressure3pm',
 'Cloud9am',
 'Cloud3pm',
 'Temp9am',
 'Temp3pm',
 'Year',
 'Month',
 'Day']
```

Engineering missing values in numerical variables

In [78]:

```
# check missing values in numerical variables in X_train

X_train[numerical].isnull().sum()
```

Out[78]:

```
MinTemp            495
MaxTemp            264
Rainfall          1139
Evaporation      48718
Sunshine         54314
WindGustSpeed     7367
WindSpeed9am      1086
WindSpeed3pm      2094
Humidity9am       1449
Humidity3pm       2890
Pressure9am      11212
Pressure3pm      11186
Cloud9am         43137
Cloud3pm         45768
Temp9am            740
Temp3pm           2171
Year                 0
Month                0
Day                  0
dtype: int64
```

In [79]:

```python
# check missing values in numerical variables in X_test

X_test[numerical].isnull().sum()
```

Out[79]:

```
MinTemp            142
MaxTemp             58
Rainfall           267
Evaporation      12125
Sunshine         13502
WindGustSpeed     1903
WindSpeed9am       262
WindSpeed3pm       536
Humidity9am        325
Humidity3pm        720
Pressure9am       2802
Pressure3pm       2795
Cloud9am         10520
Cloud3pm         11326
Temp9am            164
Temp3pm            555
Year                 0
Month                0
Day                  0
dtype: int64
```

In [80]:

```
# print percentage of missing values in the numerical variables in training set

for col in numerical:
    if X_train[col].isnull().mean()>0:
        print(col, round(X_train[col].isnull().mean(),4))
```

```
MinTemp 0.0044
MaxTemp 0.0023
Rainfall 0.01
Evaporation 0.4283
Sunshine 0.4775
WindGustSpeed 0.0648
WindSpeed9am 0.0095
WindSpeed3pm 0.0184
Humidity9am 0.0127
Humidity3pm 0.0254
Pressure9am 0.0986
Pressure3pm 0.0983
Cloud9am 0.3792
Cloud3pm 0.4023
Temp9am 0.0065
Temp3pm 0.0191
```

Assumption

I assume that the data are missing completely at random (MCAR). There are two methods which can be used to impute missing values. One is mean or median imputation and other one is random sample imputation. When there are outliers in the dataset, we should use median imputation. So, I will use median imputation because median imputation is robust to outliers.

I will impute missing values with the appropriate statistical measures of the data, in this case median. Imputation should be done over the training set, and then propagated to the test set. It means that the statistical measures to be used to fill missing values both in train and test set, should be extracted from the train set only. This is to avoid overfitting.

In [81]:

```
# impute missing values in X_train and X_test with respective column median in X_train

for df1 in [X_train, X_test]:
    for col in numerical:
        col_median=X_train[col].median()
        df1[col].fillna(col_median, inplace=True)
```

In [82]:

```
# check again missing values in numerical variables in X_train

X_train[numerical].isnull().sum()
```

Out[82]:

```
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustSpeed    0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
Year             0
Month            0
Day              0
dtype: int64
```

In [83]:

```
# check missing values in numerical variables in X_test

X_test[numerical].isnull().sum()
```

Out[83]:

```
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustSpeed    0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
Year             0
Month            0
Day              0
dtype: int64
```

Now, we can see that there are no missing values in the numerical columns of training and test set.

Engineering missing values in categorical variables

In [84]:

```
# print percentage of missing values in the categorical variables in traini
ng set

X_train[categorical].isnull().mean()
```

Out[84]:

```
Location       0.000000
WindGustDir    0.065114
WindDir9am     0.070134
WindDir3pm     0.026443
RainToday      0.010013
dtype: float64
```

In [85]:

```
# print categorical variables with missing data

for col in categorical:
    if X_train[col].isnull().mean()>0:
        print(col, (X_train[col].isnull().mean()))
```

```
WindGustDir 0.06511419378659213
WindDir9am 0.07013379749283542
WindDir3pm 0.026443026179299188
RainToday 0.01001283471350458
```

In [86]:

```
# impute missing categorical variables with most frequent value

for df2 in [X_train, X_test]:
    df2['WindGustDir'].fillna(X_train['WindGustDir'].mode()[0], inplace=T
rue)
    df2['WindDir9am'].fillna(X_train['WindDir9am'].mode()[0], inplace=Tru
e)
    df2['WindDir3pm'].fillna(X_train['WindDir3pm'].mode()[0], inplace=Tru
e)
    df2['RainToday'].fillna(X_train['RainToday'].mode()[0], inplace=True)
```

In [87]:

```
# check missing values in categorical variables in X_train

X_train[categorical].isnull().sum()
```

Out[87]:

```
Location       0
WindGustDir    0
WindDir9am     0
WindDir3pm     0
RainToday      0
dtype: int64
```

In [88]:

```
# check missing values in categorical variables in X_test

X_test[categorical].isnull().sum()
```

Out[88]:

```
Location       0
WindGustDir    0
WindDir9am     0
WindDir3pm     0
RainToday      0
dtype: int64
```

As a final check, I will check for missing values in X_train and X_test.

In [89]:

```
# check missing values in X_train

X_train.isnull().sum()
```

Out[89]:

```
Location         0
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustDir      0
WindGustSpeed    0
WindDir9am       0
WindDir3pm       0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
RainToday        0
Year             0
Month            0
Day              0
dtype: int64
```

In [90]:

```
# check missing values in X_test

X_test.isnull().sum()
```

Out[90]:

```
Location         0
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustDir      0
WindGustSpeed    0
WindDir9am       0
WindDir3pm       0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
RainToday        0
Year             0
Month            0
Day              0
dtype: int64
```

We can see that there are no missing values in X_train and X_test.

Engineering outliers in numerical variables

We have seen that the `Rainfall`, `Evaporation`, `WindSpeed9am` and `WindSpeed3pm` columns contain outliers. I will use top-coding approach to cap maximum values and remove outliers from the above variables.

In [91]:

```
def max_value(df3, variable, top):
    return np.where(df3[variable]>top, top, df3[variable])

for df3 in [X_train, X_test]:
    df3['Rainfall'] = max_value(df3, 'Rainfall', 3.2)
    df3['Evaporation'] = max_value(df3, 'Evaporation', 21.8)
    df3['WindSpeed9am'] = max_value(df3, 'WindSpeed9am', 55)
    df3['WindSpeed3pm'] = max_value(df3, 'WindSpeed3pm', 57)
```

In [92]:

```
X_train.Rainfall.max(), X_test.Rainfall.max()
```

Out[92]:

```
(3.2, 3.2)
```

In [93]:

```
X_train.Evaporation.max(), X_test.Evaporation.max()
```

Out[93]:

```
(21.8, 21.8)
```

In [94]:

```
X_train.WindSpeed9am.max(), X_test.WindSpeed9am.max()
```

Out[94]:

```
(55.0, 55.0)
```

In [95]:

```
X_train.WindSpeed3pm.max(), X_test.WindSpeed3pm.max()
```

Out[95]:

```
(57.0, 57.0)
```

In [96]:

```
X_train[numerical].describe()
```

Out[96]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
count	113754.000000	113754.000000	113754.000000	113754.000000	113754.000000
mean	12.193497	23.237216	0.675080	5.151606	8.041154
std	6.388279	7.094149	1.183837	2.823707	2.769480
min	-8.200000	-4.800000	0.000000	0.000000	0.000000
25%	7.600000	18.000000	0.000000	4.000000	8.200000
50%	12.000000	22.600000	0.000000	4.800000	8.500000
75%	16.800000	28.200000	0.600000	5.400000	8.700000
max	33.900000	48.100000	3.200000	21.800000	14.500000

We can now see that the outliers in `Rainfall`, `Evaporation`, `WindSpeed9am` and `WindSpeed3pm` columns are capped.

Encode categorical variables

In [97]:

```
# print categorical variables

categorical
```

Out[97]:

```
['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday']
```

In [98]:

```
X_train[categorical].head()
```

Out[98]:

	Location	WindGustDir	WindDir9am	WindDir3pm	RainToday
110803	Witchcliffe	S	SSE	S	No
87289	Cairns	ENE	SSE	SE	Yes
134949	AliceSprings	E	NE	N	No
85553	Cairns	ESE	SSE	E	No
16110	Newcastle	W	N	SE	No

In [99]:

```
# encode RainToday variable

import category_encoders as ce

encoder = ce.BinaryEncoder(cols=['RainToday'])

X_train = encoder.fit_transform(X_train)

X_test = encoder.transform(X_test)
```

In [100]:

```
X_train.head()
```

Out[100]:

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	Wi
110803	Witchcliffe	13.9	22.6	0.2	4.8	8.5	S	41.
87289	Cairns	22.4	29.4	2.0	6.0	6.3	ENE	33.
134949	AliceSprings	9.7	36.2	0.0	11.4	12.3	E	31.
85553	Cairns	20.5	30.1	0.0	8.8	11.1	ESE	37.
16110	Newcastle	16.8	29.2	0.0	4.8	8.5	W	39.

5 rows × 25 columns

We can see that two additional variables `RainToday_0` and `RainToday_1` are created from `RainToday` variable.

Now, I will create the `X_train` training set.

In [101]:

```
X_train = pd.concat([X_train[numerical], X_train[['RainToday_0', 'RainToday_1']],
                     pd.get_dummies(X_train.Location),
                     pd.get_dummies(X_train.WindGustDir),
                     pd.get_dummies(X_train.WindDir9am),
                     pd.get_dummies(X_train.WindDir3pm)], axis=1)
```

In [102]:

```
X_train.head()
```

Out[102]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am
110803	13.9	22.6	0.2	4.8	8.5	41.0	20.0
87289	22.4	29.4	2.0	6.0	6.3	33.0	7.0
134949	9.7	36.2	0.0	11.4	12.3	31.0	15.0
85553	20.5	30.1	0.0	8.8	11.1	37.0	22.0
16110	16.8	29.2	0.0	4.8	8.5	39.0	0.0

5 rows × 118 columns

Similarly, I will create the `X_test` testing set.

In [103]:

```
X_test = pd.concat([X_test[numerical], X_test[['RainToday_0', 'RainToday_1']],
                     pd.get_dummies(X_test.Location),
                     pd.get_dummies(X_test.WindGustDir),
                     pd.get_dummies(X_test.WindDir9am),
                     pd.get_dummies(X_test.WindDir3pm)], axis=1)
```

In [104]:

```
X_test.head()
```

Out[104]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9an
86232	17.4	29.0	0.0	3.6	11.1	33.0	11.0
57576	6.8	14.4	0.8	0.8	8.5	46.0	17.0
124071	10.1	15.4	3.2	4.8	8.5	31.0	13.0
117955	14.4	33.4	0.0	8.0	11.6	41.0	9.0
133468	6.8	14.3	3.2	0.2	7.3	28.0	15.0

5 rows × 118 columns

We now have training and testing set ready for model building. Before that, we should map all the feature variables onto the same scale. It is called `feature scaling`. I will do it as follows.

11. Feature Scaling

In [105]:

```
X_train.describe()
```

Out[105]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
count	113754.000000	113754.000000	113754.000000	113754.000000	113754.000000
mean	12.193497	23.237216	0.675080	5.151606	8.041154
std	6.388279	7.094149	1.183837	2.823707	2.769480
min	-8.200000	-4.800000	0.000000	0.000000	0.000000
25%	7.600000	18.000000	0.000000	4.000000	8.200000
50%	12.000000	22.600000	0.000000	4.800000	8.500000
75%	16.800000	28.200000	0.600000	5.400000	8.700000
max	33.900000	48.100000	3.200000	21.800000	14.500000

8 rows × 118 columns

In [106]:

```
cols = X_train.columns
```

In [107]:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

X_train = scaler.fit_transform(X_train)

X_test = scaler.transform(X_test)
```

In [108]:

```
X_train = pd.DataFrame(X_train, columns=[cols])
```

In [109]:

```
X_test = pd.DataFrame(X_test, columns=[cols])
```

In [110]:

```
X_train.describe()
```

Out[110]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
count	113754.000000	113754.000000	113754.000000	113754.000000	113754.000000
mean	0.484406	0.530004	0.210962	0.236312	0.554562
std	0.151741	0.134105	0.369949	0.129528	0.190999
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.375297	0.431002	0.000000	0.183486	0.565517
50%	0.479810	0.517958	0.000000	0.220183	0.586207
75%	0.593824	0.623819	0.187500	0.247706	0.600000
max	1.000000	1.000000	1.000000	1.000000	1.000000

8 rows × 118 columns

We now have `X_train` dataset ready to be fed into the Logistic Regression classifier. I will do it as follows.

12. Model training

In [111]:

```
# train a logistic regression model on the training set
from sklearn.linear_model import LogisticRegression


# instantiate the model
logreg = LogisticRegression(solver='liblinear', random_state=0)


# fit the model
logreg.fit(X_train, y_train)
```

Out[111]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept
=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, ver
bose=0,
                   warm_start=False)
```

13. Predict results

In [112]:

```
y_pred_test = logreg.predict(X_test)

y_pred_test
```

Out[112]:

```
array(['No', 'No', 'No', ..., 'No', 'No', 'Yes'], dtype=object)
```

predict_proba method

predict_proba method gives the probabilities for the target variable(0 and 1) in this case, in array form.

`0 is for probability of no rain` and `1 is for probability of rain.`

In [113]:

```
# probability of getting output as 0 - no rain

logreg.predict_proba(X_test)[:,0]
```

Out[113]:

```
array([0.91381393, 0.83549933, 0.8202694 , ..., 0.97674837, 0.7984706
,
       0.30735807])
```

In [114]:

```
# probability of getting output as 1 - rain

logreg.predict_proba(X_test)[:,1]
```

Out[114]:

```
array([0.08618607, 0.16450067, 0.1797306 , ..., 0.02325163, 0.2015294
,
       0.69264193])
```

14. Check accuracy score

In [115]:

```
from sklearn.metrics import accuracy_score

print('Model accuracy score: {0:0.4f}'. format(accuracy_score(y_test, y_p
red_test)))
```

```
Model accuracy score: 0.8501
```

Here, **y_test** are the true class labels and **y_pred_test** are the predicted class labels in the test-set.

Compare the train-set and test-set accuracy

Now, I will compare the train-set and test-set accuracy to check for overfitting.

In [116]:

```
y_pred_train = logreg.predict(X_train)

y_pred_train
```

Out[116]:

```
array(['No', 'No', 'No', ..., 'No', 'No', 'No'], dtype=object)
```

In [117]:

```
print('Training-set accuracy score: {0:0.4f}'. format(accuracy_score(y_tr
ain, y_pred_train)))
```

```
Training-set accuracy score: 0.8477
```

Check for overfitting and underfitting

In [118]:

```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg.score(X_train, y_train
)))

print('Test set score: {:.4f}'.format(logreg.score(X_test, y_test)))
```

```
Training set score: 0.8477
Test set score: 0.8501
```

The training-set accuracy score is 0.8476 while the test-set accuracy to be 0.8501. These two values are quite comparable. So, there is no question of overfitting.

In Logistic Regression, we use default value of C = 1. It provides good performance with approximately 85% accuracy on both the training and the test set. But the model performance on both the training and test set are very comparable. It is likely the case of underfitting.

I will increase C and fit a more flexible model.

In [119]:

```
# fit the Logsitic Regression model with C=100

# instantiate the model
logreg100 = LogisticRegression(C=100, solver='liblinear', random_state=0)


# fit the model
logreg100.fit(X_train, y_train)
```

Out[119]:

```
LogisticRegression(C=100, class_weight=None, dual=False, fit_intercept
=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, ver
bose=0,
                   warm_start=False)
```

In [120]:

```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg100.score(X_train, y_trai
n)))

print('Test set score: {:.4f}'.format(logreg100.score(X_test, y_test)))
```

```
Training set score: 0.8478
Test set score: 0.8506
```

We can see that, C=100 results in higher test set accuracy and also a slightly increased training set accuracy. So, we can conclude that a more complex model should perform better.

Now, I will investigate, what happens if we use more regularized model than the default value of C=1, by setting C=0.01.

In [121]:

```
# fit the Logsitic Regression model with C=001

# instantiate the model
logreg001 = LogisticRegression(C=0.01, solver='liblinear', random_state=0
)


# fit the model
logreg001.fit(X_train, y_train)
```

Out[121]:

```
LogisticRegression(C=0.01, class_weight=None, dual=False, fit_intercep
t=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, ver
bose=0,
                   warm_start=False)
```

In [122]:

```
# print the scores on training and test set

print('Training set score: {:.4f}'.format(logreg001.score(X_train, y_trai
n)))

print('Test set score: {:.4f}'.format(logreg001.score(X_test, y_test)))
```

```
Training set score: 0.8409
Test set score: 0.8448
```

So, if we use more regularized model by setting C=0.01, then both the training and test set accuracy decrease relative to the default parameters.

Compare model accuracy with null accuracy

So, the model accuracy is 0.8501. But, we cannot say that our model is very good based on the above accuracy. We must compare it with the **null accuracy**. Null accuracy is the accuracy that could be achieved by always predicting the most frequent class.

So, we should first check the class distribution in the test set.

In [123]:

```
# check class distribution in test set

y_test.value_counts()
```

Out[123]:

```
No     22067
Yes     6372
Name: RainTomorrow, dtype: int64
```

We can see that the occurences of most frequent class is 22067. So, we can calculate null accuracy by dividing 22067 by total number of occurences.

In [124]:

```
# check null accuracy score

null_accuracy = (22067/(22067+6372))

print('Null accuracy score: {0:0.4f}'. format(null_accuracy))
```

```
Null accuracy score: 0.7759
```

Interpretation

We can see that our model accuracy score is 0.8501 but null accuracy score is 0.7759. So, we can conclude that our Logistic Regression model is doing a very good job in predicting the class labels.

Interpretation

Now, based on the above analysis we can conclude that our classification model accuracy is very good. Our model is doing a very good job in terms of predicting the class labels.

But, it does not give the underlying distribution of values. Also, it does not tell anything about the type of errors our classifer is making.

We have another tool called `Confusion matrix` that comes to our rescue.

15. Confusion matrix

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called **Type I error.**

False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called **Type II error.**

These four outcomes are summarized in a confusion matrix given below.

In [125]:

```python
# Print the Confusion Matrix and slice it into four pieces

from sklearn.metrics import confusion_matrix

cm = confusion_matrix(y_test, y_pred_test)

print('Confusion matrix\n\n', cm)

print('\nTrue Positives(TP) = ', cm[0,0])

print('\nTrue Negatives(TN) = ', cm[1,1])

print('\nFalse Positives(FP) = ', cm[0,1])

print('\nFalse Negatives(FN) = ', cm[1,0])
```

```
Confusion matrix

 [[20892  1175]
 [ 3088  3284]]

True Positives(TP) =  20892

True Negatives(TN) =  3284

False Positives(FP) =  1175

False Negatives(FN) =  3088
```

The confusion matrix shows `20892 + 3285 = 24177 correct predictions` and `3087 + 1175 = 4262 incorrect predictions`.

In this case, we have

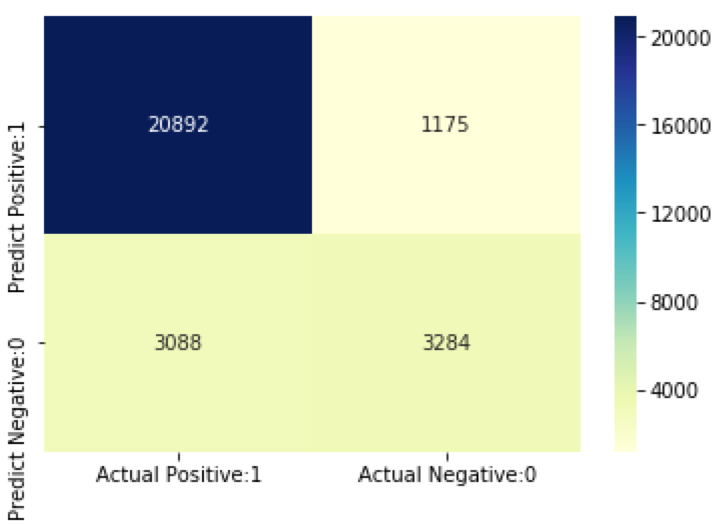
- `True Positives` (Actual Positive:1 and Predict Positive:1) - 20892
- `True Negatives` (Actual Negative:0 and Predict Negative:0) - 3285
- `False Positives` (Actual Negative:0 but Predict Positive:1) - 1175 `(Type I error)`
- `False Negatives` (Actual Positive:1 but Predict Negative:0) - 3087 `(Type II error)`

In [126]:

```
# visualize confusion matrix with seaborn heatmap

cm_matrix = pd.DataFrame(data=cm, columns=['Actual Positive:1', 'Actual Negative:0'],
                                 index=['Predict Positive:1', 'Predict Negative:0'])

sns.heatmap(cm_matrix, annot=True, fmt='d', cmap='YlGnBu')
```

Out[126]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7ffa3c199668>
```



16. Classification Metrices

Classification Report

Classification report is another way to evaluate the classification model performance. It displays the **precision**, **recall**, **f1** and **support** scores for the model. I have described these terms in later.

We can print a classification report as follows:-

In [127]:

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred_test))
```

```
              precision    recall  f1-score   support

          No       0.87      0.95      0.91     22067
         Yes       0.74      0.52      0.61      6372

    accuracy                           0.85     28439
   macro avg       0.80      0.73      0.76     28439
weighted avg       0.84      0.85      0.84     28439
```

Classification Accuracy

In [128]:

```
TP = cm[0,0]
TN = cm[1,1]
FP = cm[0,1]
FN = cm[1,0]
```

In [129]:

```
# print classification accuracy

classification_accuracy = (TP + TN) / float(TP + TN + FP + FN)

print('Classification accuracy : {0:0.4f}'.format(classification_accuracy
))
```

```
Classification accuracy : 0.8501
```

Classification Error

In [130]:

```
# print classification error

classification_error = (FP + FN) / float(TP + TN + FP + FN)

print('Classification error : {0:0.4f}'.format(classification_error))
```

```
Classification error : 0.1499
```

Precision

Precision can be defined as the percentage of correctly predicted positive outcomes out of all the predicted positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true and false positives (TP + FP).

So, **Precision** identifies the proportion of correctly predicted positive outcome. It is more concerned with the positive class than the negative class.

Mathematically, precision can be defined as the ratio of `TP to (TP + FP).`

In [131]:

```
# print precision score

precision = TP / float(TP + FP)


print('Precision : {0:0.4f}'.format(precision))
```

```
Precision : 0.9468
```

Recall

Recall can be defined as the percentage of correctly predicted positive outcomes out of all the actual positive outcomes. It can be given as the ratio of true positives (TP) to the sum of true positives and false negatives (TP + FN). **Recall** is also called **Sensitivity**.

Recall identifies the proportion of correctly predicted actual positives.

Mathematically, recall can be given as the ratio of `TP to (TP + FN).`

In [132]:

```
recall = TP / float(TP + FN)

print('Recall or Sensitivity : {0:0.4f}'.format(recall))
```

```
Recall or Sensitivity : 0.8712
```

True Positive Rate

True Positive Rate is synonymous with **Recall**.

In [133]:

```
true_positive_rate = TP / float(TP + FN)


print('True Positive Rate : {0:0.4f}'.format(true_positive_rate))
```

```
True Positive Rate : 0.8712
```

False Positive Rate

In [134]:

```
false_positive_rate = FP / float(FP + TN)


print('False Positive Rate : {0:0.4f}'.format(false_positive_rate))
```

```
False Positive Rate : 0.2635
```

Specificity

In [135]:

```
specificity = TN / (TN + FP)

print('Specificity : {0:0.4f}'.format(specificity))
```

```
Specificity : 0.7365
```

f1-score

f1-score is the weighted harmonic mean of precision and recall. The best possible **f1-score** would be 1.0 and the worst would be 0.0. **f1-score** is the harmonic mean of precision and recall. So, **f1-score** is always lower than accuracy measures as they embed precision and recall into their computation. The weighted average of `f1-score` should be used to compare classifier models, not global accuracy.

Support

Support is the actual number of occurrences of the class in our dataset.

17. Adjusting the threshold level

In [136]:

```
# print the first 10 predicted probabilities of two classes- 0 and 1

y_pred_prob = logreg.predict_proba(X_test)[0:10]

y_pred_prob
```

Out[136]:

```
array([[0.91381393, 0.08618607],
       [0.83549933, 0.16450067],
       [0.8202694 , 0.1797306 ],
       [0.99025597, 0.00974403],
       [0.95726079, 0.04273921],
       [0.97993207, 0.02006793],
       [0.17830442, 0.82169558],
       [0.23461305, 0.76538695],
       [0.9004727 , 0.0995273 ],
       [0.8548867 , 0.1451133 ]])
```

Observations

- In each row, the numbers sum to 1.
- There are 2 columns which correspond to 2 classes - 0 and 1.
    - Class 0 - predicted probability that there is no rain tomorrow.
    - Class 1 - predicted probability that there is rain tomorrow.
- Importance of predicted probabilities
    - We can rank the observations by probability of rain or no rain.
- predict_proba process
    - Predicts the probabilities
    - Choose the class with the highest probability
- Classification threshold level
    - There is a classification threshold level of 0.5.
    - Class 1 - probability of rain is predicted if probability > 0.5.
    - Class 0 - probability of no rain is predicted if probability < 0.5.

In [137]:

```
# store the probabilities in dataframe

y_pred_prob_df = pd.DataFrame(data=y_pred_prob, columns=['Prob of - No ra
in tomorrow (0)', 'Prob of - Rain tomorrow (1)'])

y_pred_prob_df
```

Out[137]:

	Prob of - No rain tomorrow (0)	Prob of - Rain tomorrow (1)
0	0.913814	0.086186
1	0.835499	0.164501
2	0.820269	0.179731
3	0.990256	0.009744
4	0.957261	0.042739
5	0.979932	0.020068
6	0.178304	0.821696
7	0.234613	0.765387
8	0.900473	0.099527
9	0.854887	0.145113

In [138]:

```
# print the first 10 predicted probabilities for class 1 - Probability of r
ain

logreg.predict_proba(X_test)[0:10, 1]
```

Out[138]:

```
array([0.08618607, 0.16450067, 0.1797306 , 0.00974403, 0.04273921,
       0.02006793, 0.82169558, 0.76538695, 0.0995273 , 0.1451133 ])
```

In [139]:

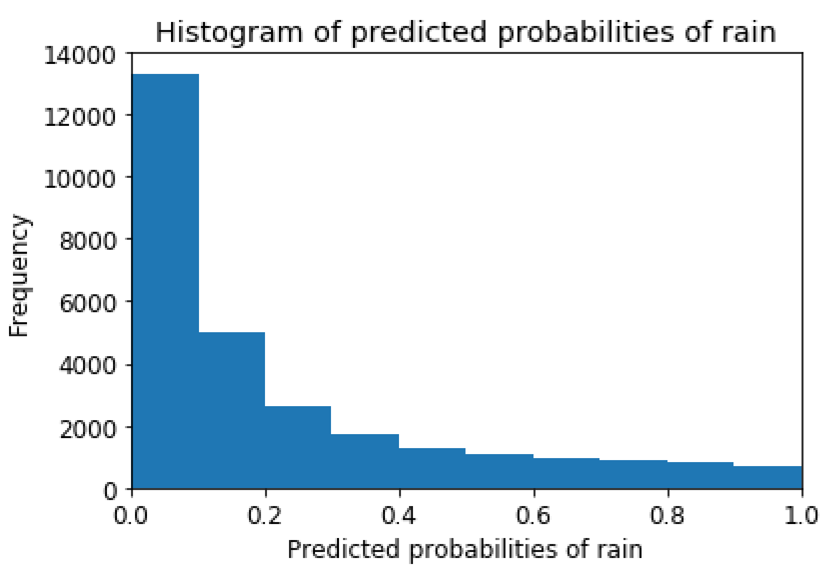
```
# store the predicted probabilities for class 1 - Probability of rain

y_pred1 = logreg.predict_proba(X_test)[:, 1]
```

In [140]:

```
# plot histogram of predicted probabilities


# adjust the font size
plt.rcParams['font.size'] = 12


# plot histogram with 10 bins
plt.hist(y_pred1, bins = 10)


# set the title of predicted probabilities
plt.title('Histogram of predicted probabilities of rain')


# set the x-axis limit
plt.xlim(0,1)


# set the title
plt.xlabel('Predicted probabilities of rain')
plt.ylabel('Frequency')
```

Out[140]:

```
Text(0, 0.5, 'Frequency')
```



Observations

- We can see that the above histogram is highly positive skewed.
- The first column tell us that there are approximately 15000 observations with probability between 0.0 and 0.1.
- There are small number of observations with probability > 0.5.
- So, these small number of observations predict that there will be rain tomorrow.
- Majority of observations predict that there will be no rain tomorrow.

Lower the threshold

In [141]:

```
from sklearn.preprocessing import binarize

for i in range(1,5):

    cm1=0

    y_pred1 = logreg.predict_proba(X_test)[:,1]

    y_pred1 = y_pred1.reshape(-1,1)

    y_pred2 = binarize(y_pred1, i/10)

    y_pred2 = np.where(y_pred2 == 1, 'Yes', 'No')

    cm1 = confusion_matrix(y_test, y_pred2)

    print ('With',i/10,'threshold the Confusion Matrix is ','\n\n',cm1,'
\n\n',

            'with',cm1[0,0]+cm1[1,1],'correct predictions, ', '\n\n',

            cm1[0,1],'Type I errors( False Positives), ','\n\n',

            cm1[1,0],'Type II errors( False Negatives), ','\n\n',

           'Accuracy score: ', (accuracy_score(y_test, y_pred2)), '\n\n',

           'Sensitivity: ',cm1[1,1]/(float(cm1[1,1]+cm1[1,0])), '\n\n',

           'Specificity: ',cm1[0,0]/(float(cm1[0,0]+cm1[0,1])),'\n\n',

            '====================================================', '\n\n
')
```

```
With 0.1 threshold the Confusion Matrix is

 [[12725  9342]
 [  547  5825]]

 with 18550 correct predictions,

 9342 Type I errors( False Positives),

 547 Type II errors( False Negatives),

 Accuracy score:  0.6522732866837793

 Sensitivity:  0.9141556811048337

 Specificity:  0.5766529206507455

 ====================================================


With 0.2 threshold the Confusion Matrix is

 [[17068  4999]
 [ 1233  5139]]

 with 22207 correct predictions,

 4999 Type I errors( False Positives),

 1233 Type II errors( False Negatives),

 Accuracy score:  0.7808643060585815

 Sensitivity:  0.806497175141243

 Specificity:  0.7734626365160647

 ====================================================
```

```
With 0.3 threshold the Confusion Matrix is

 [[19081  2986]
 [ 1873  4499]]

 with 23580 correct predictions,

 2986 Type I errors( False Positives),

 1873 Type II errors( False Negatives),

 Accuracy score:  0.8291430781673055

 Sensitivity:  0.7060577526679221

 Specificity:  0.8646848234920923

 ====================================================


With 0.4 threshold the Confusion Matrix is

 [[20191  1876]
 [ 2517  3855]]

 with 24046 correct predictions,

 1876 Type I errors( False Positives),

 2517 Type II errors( False Negatives),

 Accuracy score:  0.845529027040332

 Sensitivity:  0.6049905838041432

 Specificity:  0.9149861784565188

 ====================================================
```

Comments

- In binary problems, the threshold of 0.5 is used by default to convert predicted probabilities into class predictions.
- Threshold can be adjusted to increase sensitivity or specificity.
- Sensitivity and specificity have an inverse relationship. Increasing one would always decrease the other and vice versa.
- We can see that increasing the threshold level results in increased accuracy.
- Adjusting the threshold level should be one of the last step you do in the model-building process.

18. ROC - AUC

ROC Curve

Another tool to measure the classification model performance visually is **ROC Curve**. ROC Curve stands for **Receiver Operating Characteristic Curve**. An **ROC Curve** is a plot which shows the performance of a classification model at various classification threshold levels.

The **ROC Curve** plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)** at various threshold levels.

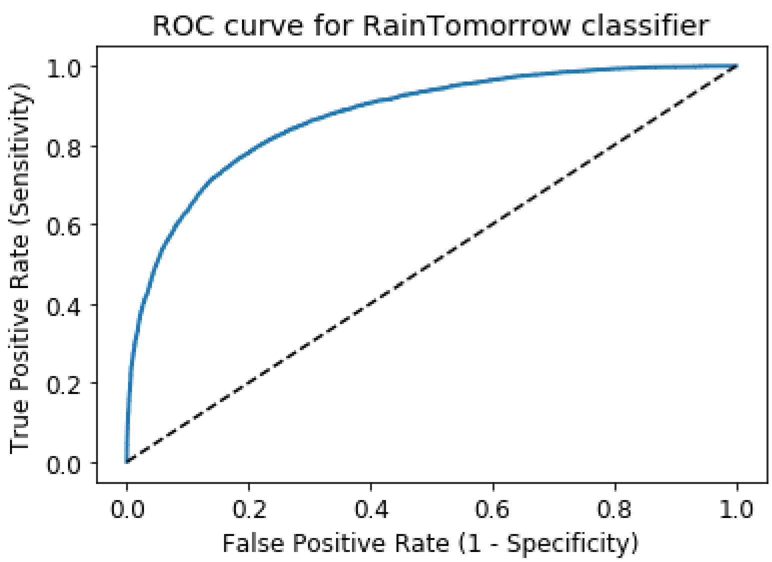
True Positive Rate (TPR) is also called **Recall**. It is defined as the ratio of `TP to (TP + FN).`

False Positive Rate (FPR) is defined as the ratio of `FP to (FP + TN).`

In the ROC Curve, we will focus on the TPR (True Positive Rate) and FPR (False Positive Rate) of a single point. This will give us the general performance of the ROC curve which consists of the TPR and FPR at various threshold levels. So, an ROC Curve plots TPR vs FPR at different classification threshold levels. If we lower the threshold levels, it may result in more items being classified as positve. It will increase both True Positives (TP) and False Positives (FP).

In [142]:

```
# plot ROC Curve

from sklearn.metrics import roc_curve

fpr, tpr, thresholds = roc_curve(y_test, y_pred1, pos_label = 'Yes')

plt.figure(figsize=(6,4))

plt.plot(fpr, tpr, linewidth=2)

plt.plot([0,1], [0,1], 'k--' )

plt.rcParams['font.size'] = 12

plt.title('ROC curve for RainTomorrow classifier')

plt.xlabel('False Positive Rate (1 - Specificity)')

plt.ylabel('True Positive Rate (Sensitivity)')

plt.show()
```



ROC curve help us to choose a threshold level that balances sensitivity and specificity for a particular context.

ROC AUC

ROC AUC stands for **Receiver Operating Characteristic - Area Under Curve**. It is a technique to compare classifier performance. In this technique, we measure the `area under the curve (AUC)`. 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.

So, **ROC AUC** is the percentage of the ROC plot that is underneath the curve.

In [143]:

```
# compute ROC AUC

from sklearn.metrics import roc_auc_score

ROC_AUC = roc_auc_score(y_test, y_pred1)

print('ROC AUC : {:.4f}'.format(ROC_AUC))
```

```
ROC AUC : 0.8729
```

Comments

- ROC AUC is a single number summary of classifier performance. The higher the value, the better the classifier.
- ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a good job in predicting whether it will rain tomorrow or not.

In [144]:

```
# calculate cross-validated ROC AUC

from sklearn.model_selection import cross_val_score

Cross_validated_ROC_AUC = cross_val_score(logreg, X_train, y_train, cv=5,
scoring='roc_auc').mean()

print('Cross validated ROC AUC : {:.4f}'.format(Cross_validated_ROC_AUC))
```

```
Cross validated ROC AUC : 0.8695
```

Model evaluation and improvement

In this section, I will employ several techniques to improve the model performance. I will discuss 3 techniques which are used in practice for performance improvement. These are `recursive feature elimination`, `k-fold cross validation` and `hyperparameter optimization using GridSearchCV`.

19. Recursive Feature Elimination with Cross Validation

`Recursive feature elimination (RFE)` is a feature selection technique that helps us to select best features from the given number of features. At first, the model is built on all the given features. Then, it removes the least useful predictor and build the model again. This process is repeated until all the unimportant features are removed from the model.

`Recursive Feature Elimination with Cross-Validated (RFECV) feature selection` technique selects the best subset of features for the estimator by removing 0 to N features iteratively using recursive feature elimination. Then it selects the best subset based on the accuracy or cross-validation score or roc-auc of the model. Recursive feature elimination technique eliminates n features from a model by fitting the model multiple times and at each step, removing the weakest features.

I will use this technique to select best features from this model.

In [145]:

```
from sklearn.feature_selection import RFECV

rfecv = RFECV(estimator=logreg, step=1, cv=5, scoring='accuracy')

rfecv = rfecv.fit(X_train, y_train)
```

In [146]:

```
print("Optimal number of features : %d" % rfecv.n_features_)
```

```
Optimal number of features : 111
```

In [147]:

```
# transform the training data

X_train_rfecv = rfecv.transform(X_train)



# train classifier

logreg.fit(X_train_rfecv, y_train)
```

Out[147]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept
=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='liblinear', tol=0.0001, ver
bose=0,
                   warm_start=False)
```

In [148]:

```
# test classifier on test data

X_test_rfecv = rfecv.transform(X_test)

y_pred_rfecv = logreg.predict(X_test_rfecv)
```

In [149]:

```
# print mean accuracy on transformed test data and labels

print ("Classifier score: {:.4f}".format(logreg.score(X_test_rfecv,y_test
)))
```

```
Classifier score: 0.8499
```

Our original model accuracy score is 0.8501 whereas accuracy score after RFECV is 0.8500. So, we can obtain approximately similar accuracy but with reduced or optimal set of features.

Confusion-matrix revisited

I will again plot the confusion-matrix for this model to get an idea of errors our model is making.

In [150]:

```
from sklearn.metrics import confusion_matrix

cm1 = confusion_matrix(y_test, y_pred_rfecv)

print('Confusion matrix\n\n', cm1)

print('\nTrue Positives(TP1) = ', cm1[0,0])

print('\nTrue Negatives(TN1) = ', cm1[1,1])

print('\nFalse Positives(FP1) = ', cm1[0,1])

print('\nFalse Negatives(FN1) = ', cm1[1,0])
```

```
Confusion matrix

 [[20892  1175]
 [ 3093  3279]]

True Positives(TP1) =  20892

True Negatives(TN1) =  3279

False Positives(FP1) =  1175

False Negatives(FN1) =  3093
```

We can see that in the original model, we have FP = 1175 whereas FP1 = 1174. So, we get approximately same number of false positives. Also, FN = 3087 whereas FN1 = 3091. So, we get slightly higher false negatives.

20. k-Fold Cross Validation

In [151]:

```
# Applying 5-Fold Cross Validation

from sklearn.model_selection import cross_val_score

scores = cross_val_score(logreg, X_train, y_train, cv = 5, scoring='accuracy')

print('Cross-validation scores:{}'.format(scores))
```

```
Cross-validation scores:[0.84690783 0.84624852 0.84633642 0.84963298
0.84773626]
```

We can summarize the cross-validation accuracy by calculating its mean.

In [152]:

```
# compute Average cross-validation score

print('Average cross-validation score: {:.4f}'.format(scores.mean()))
```

```
Average cross-validation score: 0.8474
```

Our, original model score is found to be 0.8476. The average cross-validation score is 0.8474. So, we can conclude that cross-validation does not result in performance improvement.

21. Hyperparameter Optimization using GridSearch CV

In [153]:

```
from sklearn.model_selection import GridSearchCV


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



grid_search = GridSearchCV(estimator = logreg,
                           param_grid = parameters,
                           scoring = 'accuracy',
                           cv = 5,
                           verbose=0)


grid_search.fit(X_train, y_train)
```

Out[153]:

```
GridSearchCV(cv=5, error_score='raise-deprecating',
             estimator=LogisticRegression(C=1.0, class_weight=None, du
al=False,
                                          fit_intercept=True,
                                          intercept_scaling=1, l1_rati
o=None,
                                          max_iter=100, multi_class='w
arn',
                                          n_jobs=None, penalty='l2',
                                          random_state=0, solver='libl
inear',
                                          tol=0.0001, verbose=0,
                                          warm_start=False),
             iid='warn', n_jobs=None,
             param_grid=[{'penalty': ['l1', 'l2']}, {'C': [1, 10, 100,
1000]}],
             pre_dispatch='2*n_jobs', refit=True, return_train_score=F
alse,
             scoring='accuracy', verbose=0)
```

In [154]:

```python
# examine the best model

# best score achieved during the GridSearchCV
print('GridSearch CV best score : {:.4f}\n\n'.format(grid_search.best_sco
re_))

# print parameters that give the best results
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_searc
h.best_estimator_))
```

```
GridSearch CV best score : 0.8474


Parameters that give the best results :

 {'penalty': 'l1'}


Estimator that was chosen by the search :

 LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercep
t=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l1',
                   random_state=0, solver='liblinear', tol=0.0001, ver
bose=0,
                   warm_start=False)
```

In [155]:

```
# calculate GridSearch CV score on test set

print('GridSearch CV score on test set: {0:0.4f}'.format(grid_search.score(X_test, y_test)))
```

```
GridSearch CV score on test set: 0.8507
```

Comments

- Our original model test accuracy is 0.8501 while GridSearch CV accuracy is 0.8507.
- We can see that GridSearch CV improve the performance for this particular model.

22. Results and Conclusion

1. The logistic regression model accuracy score is 0.8501. So, the model does a very good job in predicting whether or not it will rain tomorrow in Australia.
2. Small number of observations predict that there will be rain tomorrow. Majority of observations predict that there will be no rain tomorrow.
3. The model shows no signs of overfitting.
4. Increasing the value of C results in higher test set accuracy and also a slightly increased training set accuracy. So, we can conclude that a more complex model should perform better.
5. Increasing the threshold level results in increased accuracy.
6. ROC AUC of our model approaches towards 1. So, we can conclude that our classifier does a good job in predicting whether it will rain tomorrow or not.
7. Our original model accuracy score is 0.8501 whereas accuracy score after RFECV is 0.8500. So, we can obtain approximately similar accuracy but with reduced set of features.
8. In the original model, we have FP = 1175 whereas FP1 = 1174. So, we get approximately same number of false positives. Also, FN = 3087 whereas FN1 = 3091. So, we get slighly higher false negatives.
9. Our, original model score is found to be 0.8476. The average cross-validation score is 0.8474. So, we can conclude that cross-validation does not result in performance improvement.
10. Our original model test accuracy is 0.8501 while GridSearch CV accuracy is 0.8507. We can see that GridSearch CV improve the performance for this particular model.

23. References

The work done in this project is inspired from the following books and websites:-

1. Hands on Machine Learning with Scikit-Learn and Tensorflow by Aurélién Géron.
2. Introduction to Machine Learning with Python by Andreas C Muller and Sarah Guido.
3. Udemy course – Feature Engineering for Machine Learning by Soledad Galli.

Thank you for reading this kernel. I hope you enjoyed it.

Your comments and feedback are most welcome.

Go to the Top of the Page (https://www.kaggle.com/prashant111/extensive-analysis-eda-fe-modelling#0)