Extensive Analysis - EDA + FE + Modelling

Objective - Use a classification model to predict whether it will rain tomorrow or not, using the weather dataset from the previous 10 years

In [ ]:

Importing Libraries

In [1]:
```
import pandas as pd
import numpy as np
import seaborn as sns
import sklearn
import matplotlib.pyplot as plt
import matplotlib as mp
from plotly import express as px
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
```

Loading Dataset

In [2]:
```
rain_df = pd.read_csv('weatherAUS.csv')
```

In [3]:
```
# Checking NULL rows for our target variable
rain_df.isna().sum()
```

Out[3]:
```
Date                0
Location            0
MinTemp          1485
MaxTemp          1261
Rainfall         3261
Evaporation     62790
Sunshine        69835
WindGustDir     10326
WindGustSpeed   10263
WindDir9am      10566
WindDir3pm       4228
WindSpeed9am     1767
WindSpeed3pm     3062
Humidity9am      2654
Humidity3pm      4507
Pressure9am     15065
Pressure3pm     15028
Cloud9am        55888
Cloud3pm        59358
Temp9am          1767
Temp3pm          3609
RainToday        3261
RainTomorrow     3267
dtype: int64
```

In [4]:
```
# We can see that about 3000 rows dont contain values for our Target Variable, so we cannot use these rows.
# In addition to this, if values for "RainToday" are also missing, then we will disregard these values also because
# this factor seems influential on our target variable.
rain_df.dropna(subset=['RainToday','RainTomorrow'],inplace=True)
```

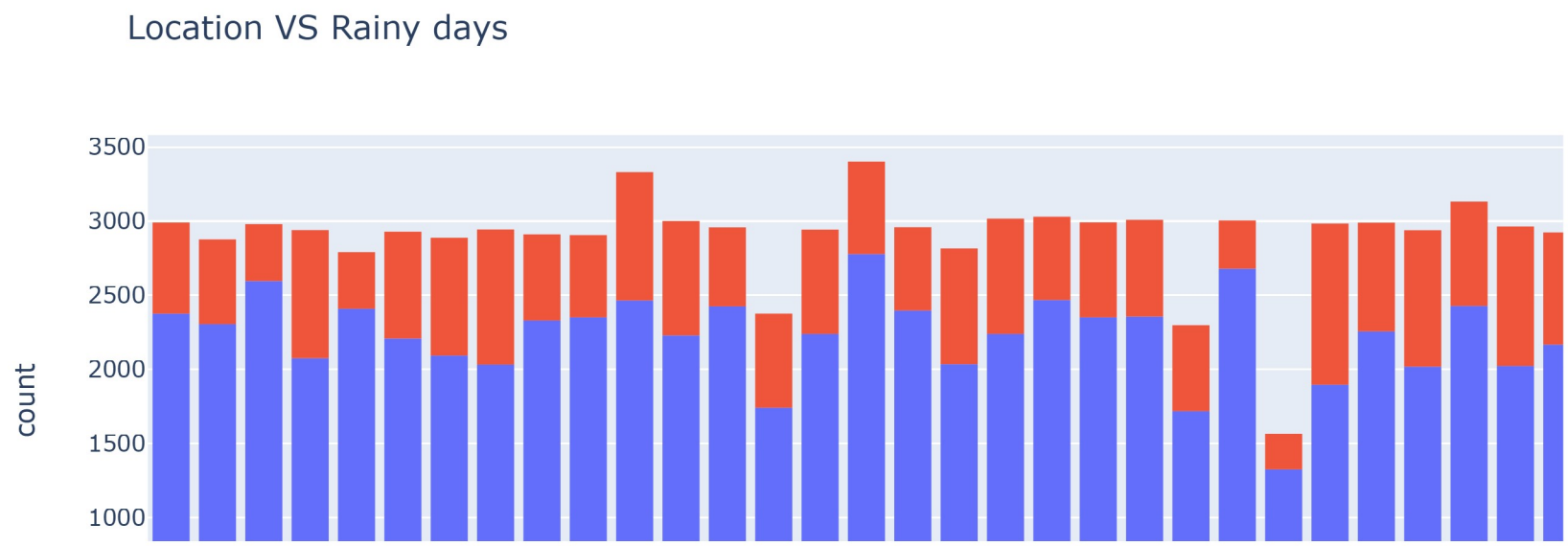
In [5]:
```
rain_df.isna().sum()
```

Out[5]:
```
Date                0
Location            0
MinTemp           468
MaxTemp           307
Rainfall            0
Evaporation     59694
Sunshine        66805
WindGustDir      9163
WindGustSpeed    9105
WindDir9am       9660
WindDir3pm       3670
WindSpeed9am     1055
WindSpeed3pm     2531
Humidity9am      1517
Humidity3pm      3501
Pressure9am     13743
Pressure3pm     13769
Cloud9am        52625
Cloud3pm        56094
Temp9am           656
Temp3pm          2624
RainToday           0
RainTomorrow        0
dtype: int64
```

In [ ]:

Exploratory Data Analysis

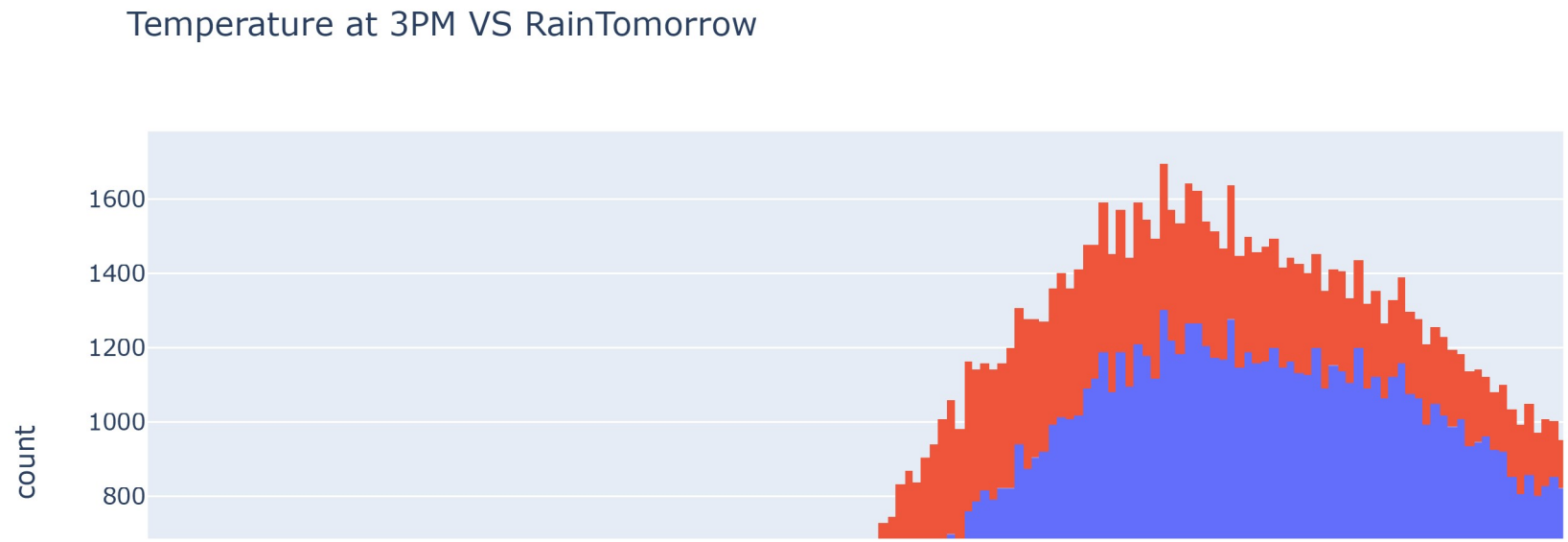
In [6]:
```
# Checking number of locations and their count of values
px.histogram(rain_df,x="Location",title='Location VS Rainy days',color='RainToday')
```



EDA - Inference of Location

In [7]:
```
# We can see that we have 48 unique locations, with close to 3000 data points for each location. Some of these locat
# don't have exactly 3000 data points. This could be because their setup might have been installed at a later time pe

# We can also see that for most of the locations, roughly 20% of the days had rain, and 80% didn't have rain, regardl
# location, except some exceptions. We can conclude that Location is definitely an important factor, because the quan
# of rain is dependent on location.
```

In [8]:
```
# Checking correlation of 'Temperature3PM' with our Target Variable
px.histogram(rain_df,x='Temp3pm',title='Temperature at 3PM VS RainTomorrow',color='RainTomorrow')
```



EDA - Inference of Temperature at 3PM

In [9]:
```
# From this Gaussian Distribution curve, we can infer that low temperature at 3PM directly correlates with RainTomorr
# If temperature at 3PM is low, then we can expect rain tomorrow - this is because we can see that
# the cases of "lowTemp yesRain" > "highTemp yesRain"
# So if temperature is lower, it is more likely to rain tomorrow.
```

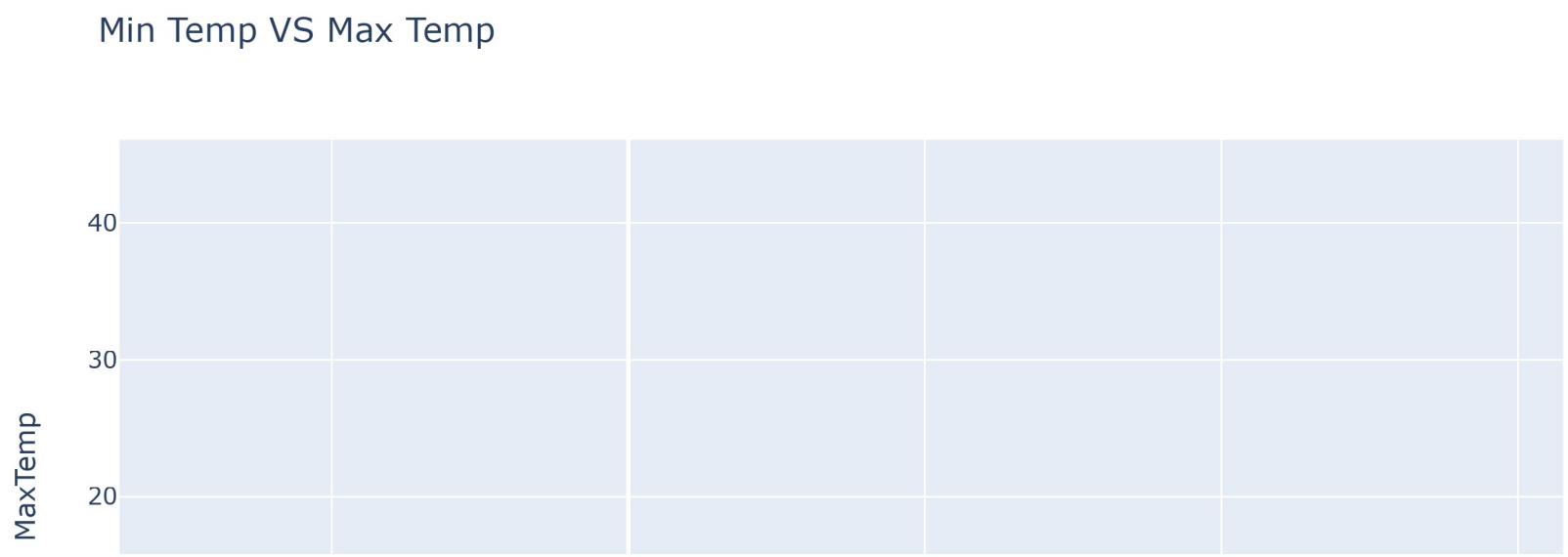
In [10]:
```
# Checking relation between RainToday VS RainTomorrow
px.histogram(rain_df,x="RainTomorrow",color="RainToday",title="RainTomorrow VS RainToday")
```

EDA - Inference of RainToday VS RainTomorrow

In [11]:
```
# We can see that we have class imbalance here, since about 78% of datapoints have a target value of 'No'.

# We can also infer that when 'RainToday' = 'No', most of 'RainTomorrow' = 'No' also.
# This means that is it easier to predict 'RainTomorrow = No' IF 'RainToday' is already 'No'.

# However, when 'rainToday = Yes', then there becomes a 50% chance that 'RainTomorrow = Yes'.

# So it is easy to predict rainTomorrow = No if rainToday is already No but it is not  as easy to predict if rainTomo
# when rainToday = Yes
# => If 'RainToday = No', then we can be confident that rainTomorrow = No only
```

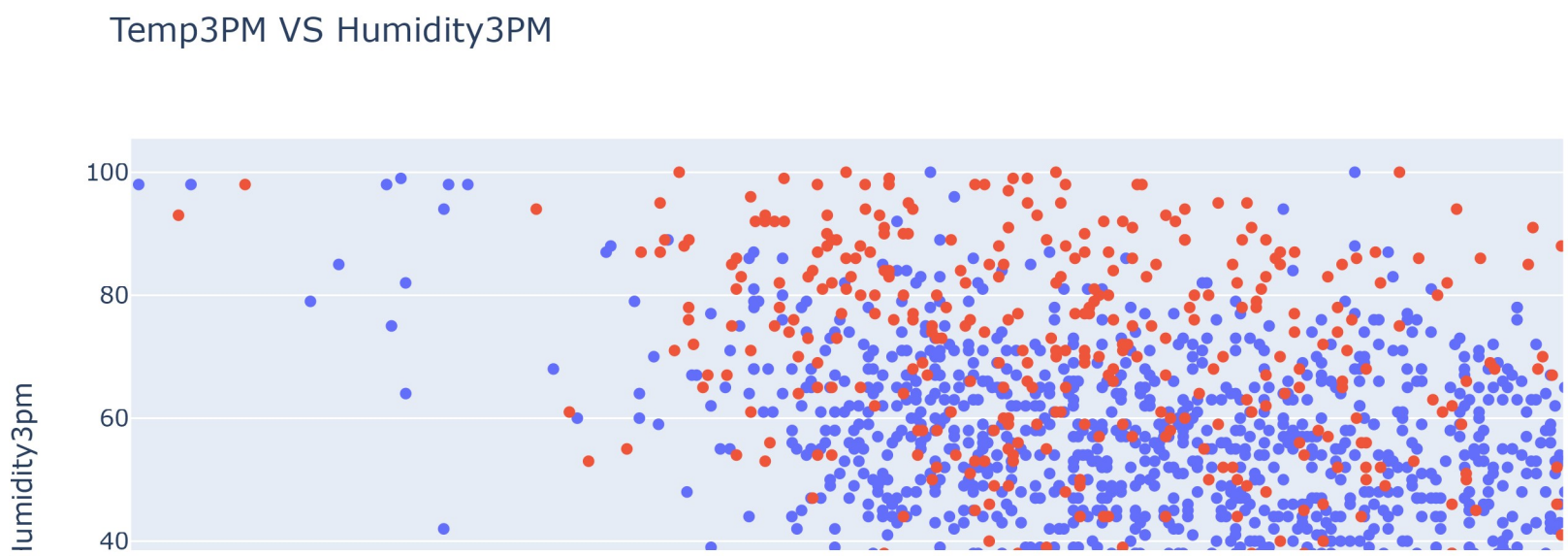
EDA - Checking whether Minimum and Maximum Temperature has any effect on weather conditions

In [12]:
```
px.scatter(rain_df.sample(2000),title="Min Temp VS Max Temp",x="MinTemp",y="MaxTemp",color="RainToday")
# Even though this graph seems erratic initially, we can infer that rainToday = Yes is more prominent in cases where
# difference between Minimum and Maximum temperature is low
# => When the variation in temperature of a day is small (and Minimum temperature is low), 'RainToday = Yes' is more
```



EDA - Checking whether Temperature_3PM and Humidity_3PM effects RainTomorrow

In [13]:
```
px.strip(rain_df.sample(2000),title="Temp3PM VS Humidity3PM",x='Temp3pm',y='Humidity3pm',color="RainTomorrow")
# We can infer that since most of the 'RainTomorrow = Yes' points are in the high humidity and low temperature regio
# so these factors influence RainTomorrow.
# => High Humidity and Low Temperature cause RainTomorrow more than their counterparts
```



Splitting dataset into Train, Validation and Test

In [14]:
```
from sklearn.model_selection import train_test_split
```

In [15]:
```
#train_val_df, test_df = train_test_split(rain_df, test_size = 0.2,random_state=42) # extracting test dataframe with
#train_df, val_df = train_test_split(train_val_df, test_size = 0.25,random_state=42) # extracting validation datafra
```

In [16]:
```
#print("TrainingDataFrame: ",train_df.shape)
#print("TestDataFrame: ",test_df.shape)
#print("ValidationDataFrame: ",val_df.shape)
```

In [17]:
```
# The above apprach is good, but when we work with Time Series Data, i.e., data that spans some time (for eg from 20
# to keep early years for training set, then 1-1.5 years for validation set and the previous year for test set -> th
# so, our model should not have access to data from future to predict something for today
# also, it is good practice to use the immediately last year as the test set, because it is the most relevant dataset
```

In [18]:
```
year = pd.to_datetime(rain_df.Date).dt.year
train_df = rain_df[year < 2015] # Training Set - all the data before 2015
val_df = rain_df[year == 2015] # Validation Set - all the data for 2015
test_df = rain_df[year > 2015] # Testing Se - all the data after 2015

print("TrainingDataFrame: ",train_df.shape)
print("TestDataFrame: ",test_df.shape)
print("ValidationDataFrame: ",val_df.shape)
```

```
TrainingDataFrame:  (97988, 23)
TestDataFrame:  (25710, 23)
ValidationDataFrame:  (17089, 23)
```

In [19]:
```
# ------------------------------------------------------------ #
```

Feature Engineering

In [20]:
```
# Making list of input and output columns - Segregating X and Y values
input_cols = list(train_df.columns)[1:-1] # picking all columns except first and last - first is date which can be ig
target_col = 'RainTomorrow'
```

In [21]:
```
# creating copy dataframe with all X variables, and another dataframe with the Y column for Training Set

train_inputs = train_df[input_cols].copy()
train_targets = train_df[target_col].copy()
```

In [22]:
```
# creating copy dataframe with all X variables, and another dataframe with the Y column for Testing Set

test_input = test_df[input_cols].copy()
test_targets = test_df[target_col].copy()
```

In [23]:
```
# creating copy dataframe with all X variables, and another dataframe with the Y column for Validation Set

val_input = val_df[input_cols].copy()
val_targets = val_df[target_col].copy()
```

In [24]:
```
# Recognizing categorical and numerical data

numeric_cols = train_inputs.select_dtypes(include=np.number).columns.tolist()
categorical_cols = train_inputs.select_dtypes('object').columns.tolist()
```

In [25]:
```
# checking number of unique categories in each categorical variable

train_inputs[categorical_cols].nunique()
```

Out[25]:
```
Location       49
WindGustDir    16
WindDir9am     16
WindDir3pm     16
RainToday       2
dtype: int64
```

Data Processing

Imputing Missing Numerical Data

In [26]:
```
# ML cannot work with missing numerical data - so we need to substitute the missing numerical values with some alterr
# i.e., we have to perform imputation - we do this by substituting the values with the average (mean) of other values
# We can also discard the NaN rows but by so doing, we may lose out on a lot of data - if a lot of rows contain NaN v
# in case of data where average can effect the mean (for eg salary distribution),
# it is better to impute using the median instead of the mean
```

Imputing Values

In [ ]:
```

```

In [27]:
```
imputer = SimpleImputer(strategy="mean")
```

In [28]:
```
imputer.fit(rain_df[numeric_cols])
```

Out[28]:
```
SimpleImputer()
```

In [29]:
```
# replacing the NaNs with the averages in each dataframe

train_inputs[numeric_cols] = imputer.transform(train_inputs[numeric_cols])
val_input[numeric_cols] = imputer.transform(val_input[numeric_cols])
test_input[numeric_cols] = imputer.transform(test_input[numeric_cols])
```

In [30]:
```
# Imputation is now complete - none of the columns now contain any NaN values
```

Scaling Numerical Data

In [31]:
```
# In a real world dataset, the numerical columns have values that have varying ranges
# for example, Rainfall goes from 0.0001 to 1000+ while temperature goes from 0 to 50.
# Columns with such high varying values tend to dominate the loss function and columns with higher values and higher
# i.e., their weights changes a lot during optimization
# but columns with smaller values and/or smaller ranges - their weights dont change as much during optimization
# so to level the playing field, we scale all the numerical values between [0,1]
```

In [32]:
```
# MinMaxScaler does 2 things:
# a. it recognizes the min and max of each numerical column
# b. it scales each numerical column down to the range [0,1] based on the min and max values
```

In [33]:
```
scaler = MinMaxScaler()
scaler.fit(rain_df[numeric_cols])
```

Out[33]:
```
MinMaxScaler()
```

In [34]:
```
train_inputs[numeric_cols] = scaler.transform(train_inputs[numeric_cols])
val_input[numeric_cols] = scaler.transform(val_input[numeric_cols])
test_input[numeric_cols] = scaler.transform(test_input[numeric_cols])
```

Categorical data encoding using One-Hot Encoding

In [35]:
```
# We use One Hot Encoding here - since 49 categorical values (of Location) is not high enough
```

In [36]:
```
rain_df[categorical_cols].nunique()
```

Out[36]:
```
Location       49
WindGustDir    16
WindDir9am     16
WindDir3pm     16
RainToday       2
dtype: int64
```

In [37]:
```
encoder = OneHotEncoder(sparse=False,handle_unknown='ignore') # dont create Sparse matrix # ignore new unknown catego
```

In [38]:
```
# identifying the different categories present in the data
encoder.fit(rain_df[categorical_cols])
# now we have identified the different categories present in each categorical column
```

Out[38]:
```
OneHotEncoder(handle_unknown='ignore', sparse=False)
```

In [39]:
```
# generating column names for each generated category using get_feature_names_out function

encoded_cols = list(encoder.get_feature_names_out(categorical_cols))
print(encoded_cols)
```

```
['Location_Adelaide', 'Location_Albany', 'Location_Albury', 'Location_AliceSprings', 'Location_BadgerysCreek', 'Loca
tion_Ballarat', 'Location_Bendigo', 'Location_Brisbane', 'Location_Cairns', 'Location_Canberra', 'Location_Cobar', '
Location_CoffsHarbour', 'Location_Dartmoor', 'Location_Darwin', 'Location_GoldCoast', 'Location_Hobart', 'Location_K
atherine', 'Location_Launceston', 'Location_Melbourne', 'Location_MelbourneAirport', 'Location_Mildura', 'Location_M
oree', 'Location_MountGambier', 'Location_MountGinini', 'Location_Newcastle', 'Location_Nhil', 'Location_NorahHead',
'Location_NorfolkIsland', 'Location_Nuriootpa', 'Location_PearceRAAF', 'Location_Penrith', 'Location_Perth', 'Locati
on_PerthAirport', 'Location_Portland', 'Location_Richmond', 'Location_Sale', 'Location_SalmonGums', 'Location_Sydney
', 'Location_SydneyAirport', 'Location_Townsville', 'Location_Tuggeranong', 'Location_Uluru', 'Location_WaggaWagga',
'Location_Walpole', 'Location_Watsonia', 'Location_Williamtown', 'Location_Witchcliffe', 'Location_Wollongong', 'Loc
ation_Woomera', 'WindGustDir_E', 'WindGustDir_ENE', 'WindGustDir_ESE', 'WindGustDir_N', 'WindGustDir_NE', 'WindGustD
ir_NNE', 'WindGustDir_NNW', 'WindGustDir_NW', 'WindGustDir_S', 'WindGustDir_SE', 'WindGustDir_SSE', 'WindGustDir_SSW
', 'WindGustDir_SW', 'WindGustDir_W', 'WindGustDir_WNW', 'WindGustDir_WSW', 'WindGustDir_nan', 'WindDir9am_E', 'Wind
Dir9am_ENE', 'WindDir9am_ESE', 'WindDir9am_N', 'WindDir9am_NE', 'WindDir9am_NNE', 'WindDir9am_NNW', 'WindDir9am_NW',
'WindDir9am_S', 'WindDir9am_SE', 'WindDir9am_SSE', 'WindDir9am_SSW', 'WindDir9am_SW', 'WindDir9am_W', 'WindDir9am_WN
W', 'WindDir9am_WSW', 'WindDir9am_nan', 'WindDir3pm_E', 'WindDir3pm_ENE', 'WindDir3pm_ESE', 'WindDir3pm_N', 'WindDir
3pm_NE', 'WindDir3pm_NNE', 'WindDir3pm_NNW', 'WindDir3pm_NW', 'WindDir3pm_S', 'WindDir3pm_SE', 'WindDir3pm_SSE', 'Wi
ndDir3pm_SSW', 'WindDir3pm_SW', 'WindDir3pm_W', 'WindDir3pm_WNW', 'WindDir3pm_WSW', 'WindDir3pm_nan', 'RainToday_No
', 'RainToday_Yes']
```

In [40]:
```
# using encoder.transform to actually create the encoded columns in the dataframe
train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols])
val_input[encoded_cols] = encoder.transform(val_input[categorical_cols])
test_input[encoded_cols] = encoder.transform(test_input[categorical_cols])
# now we have gotten our one-hot encoded vectors in a NumPy array
```

```
C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:2: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:2: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:2: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:2: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:3: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:3: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:3: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:3: PerformanceWarning:
```

```
DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:4: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:4: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:4: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`

C:\Users\sahil\AppData\Local\Temp\ipykernel_30748\1149358685.py:4: PerformanceWarning:

DataFrame is highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor per
formance.  Consider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `
newframe = frame.copy()`
```

In [41]:

```
# now categorical encoding is done
```

Saving pre-processed intermediate outputs to local disk using Parquet Format

In [42]:

```
# parquet format has the benefit that it saves the current state of our variables and dataframe
# and we can simply load them into the new notebook without needing to go through the pre-pocessing stages again
```

In [43]:
```
# saving training, validation, testing dataframe
train_inputs.to_parquet('train_inputs.parquet')
val_input.to_parquet('val_input.parquet')
test_input.to_parquet('test_input.parquet')
```

In [44]:
```
# saving targets by storing as DataFrame
pd.DataFrame(train_targets).to_parquet('train_targets.parquet')
pd.DataFrame(val_targets).to_parquet('val_targets.parquet')
pd.DataFrame(test_targets).to_parquet('test_targets.parquet')
```

In [45]:
```
# we can now read the data back using pd.read_parquet
train_inputs = pd.read_parquet('train_inputs.parquet')
val_input = pd.read_parquet('val_input.parquet')
test_input = pd.read_parquet('test_input.parquet')

train_targets = pd.read_parquet('train_targets.parquet')[target_col]
val_targets = pd.read_parquet('val_targets.parquet')[target_col]
test_targets = pd.read_parquet('test_targets.parquet')[target_col]
```

Training the logistic regression model

In [46]:
```
# Logistic Regression is a commonly used technique for solving Binary Classification problems. We follow the followi
# a. Take linear combination of input features
# b. apply the sigmoid function to attain a number between 0 and 1 - this represents the probability of Y being 1("Ye
# c. instead of RMSE, the cross entropy loss function is used to evaluate the results as:
# c.i. if Y = 1, we want the prob to be as high as possible
# c.ii. if Y = 0, we want the prob to be as low as possible
# the cross entropy loss function penalizes bad predictions, which is used to evaluate model's effectiveness
```

In [47]:
```
from sklearn.linear_model import LogisticRegression
```

In [48]:
```
model = LogisticRegression(solver='liblinear')
```

In [49]:
```
# training the model
model.fit(train_inputs[numeric_cols+encoded_cols],train_targets)
# not giving train_inputs as the first parameter because
# our training set still contains location as a categorical feature
# FYI - "train_targets" is still categorical since it contains Yes and No, but our target can have categorical values
```

Out[49]:
```
LogisticRegression(solver='liblinear')
```

In [50]:
```
# basically - targets can be categorical but inputs cannot be categorical in logReg
```

In [51]:
```
# checking the coefficient values for each value of X:
n = len(model.coef_.tolist())
weight_df = pd.DataFrame({
    'feature':(numeric_cols + encoded_cols),
    'weight':model.coef_.tolist()[0]
})
# higher the weight - more important is the feature
# +ve sign - +ve correlation
# -ve sign - -ve correlation
```

In [52]:
```
print(model.intercept_)
```

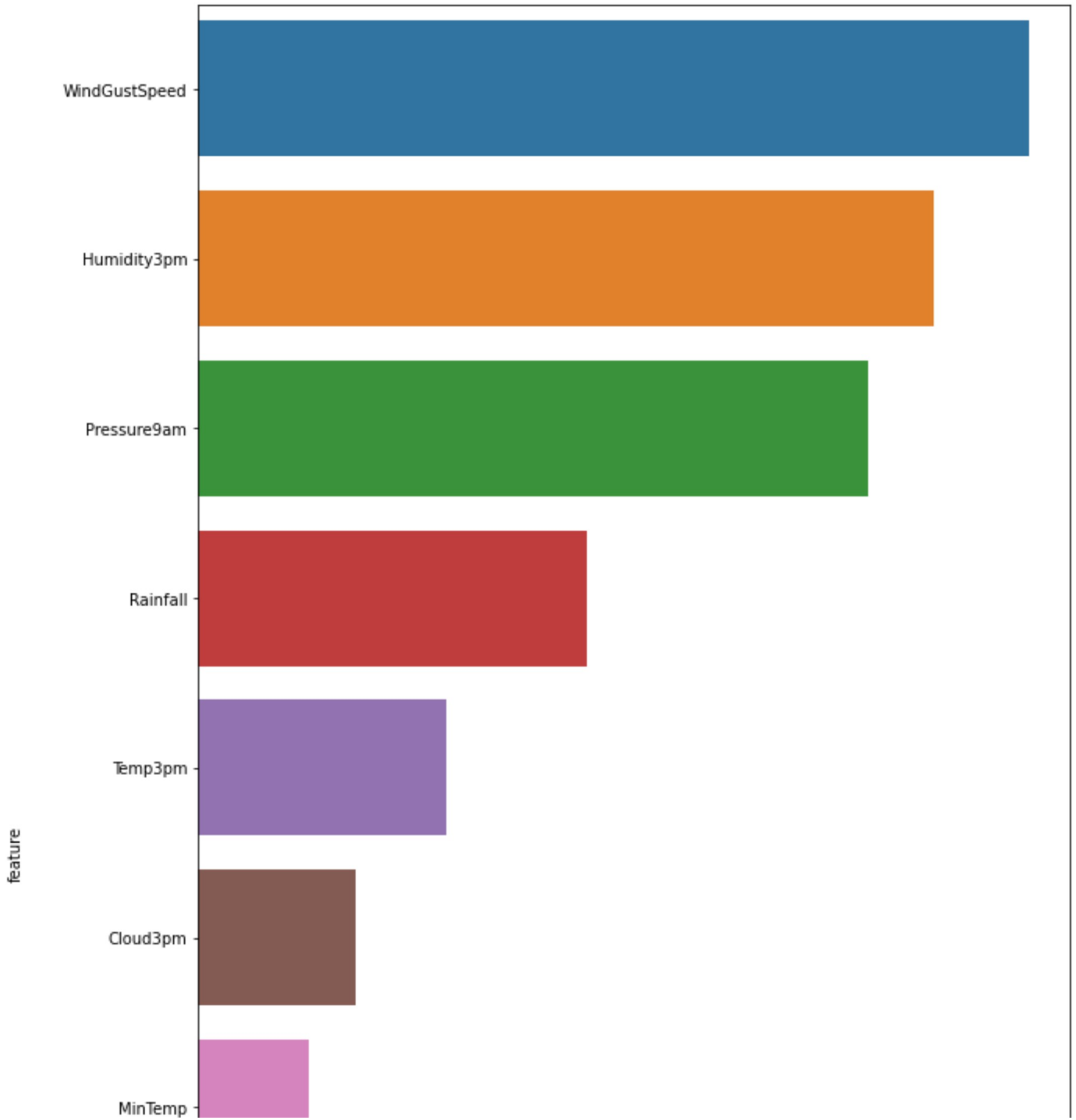
```
[-2.44955563]
```

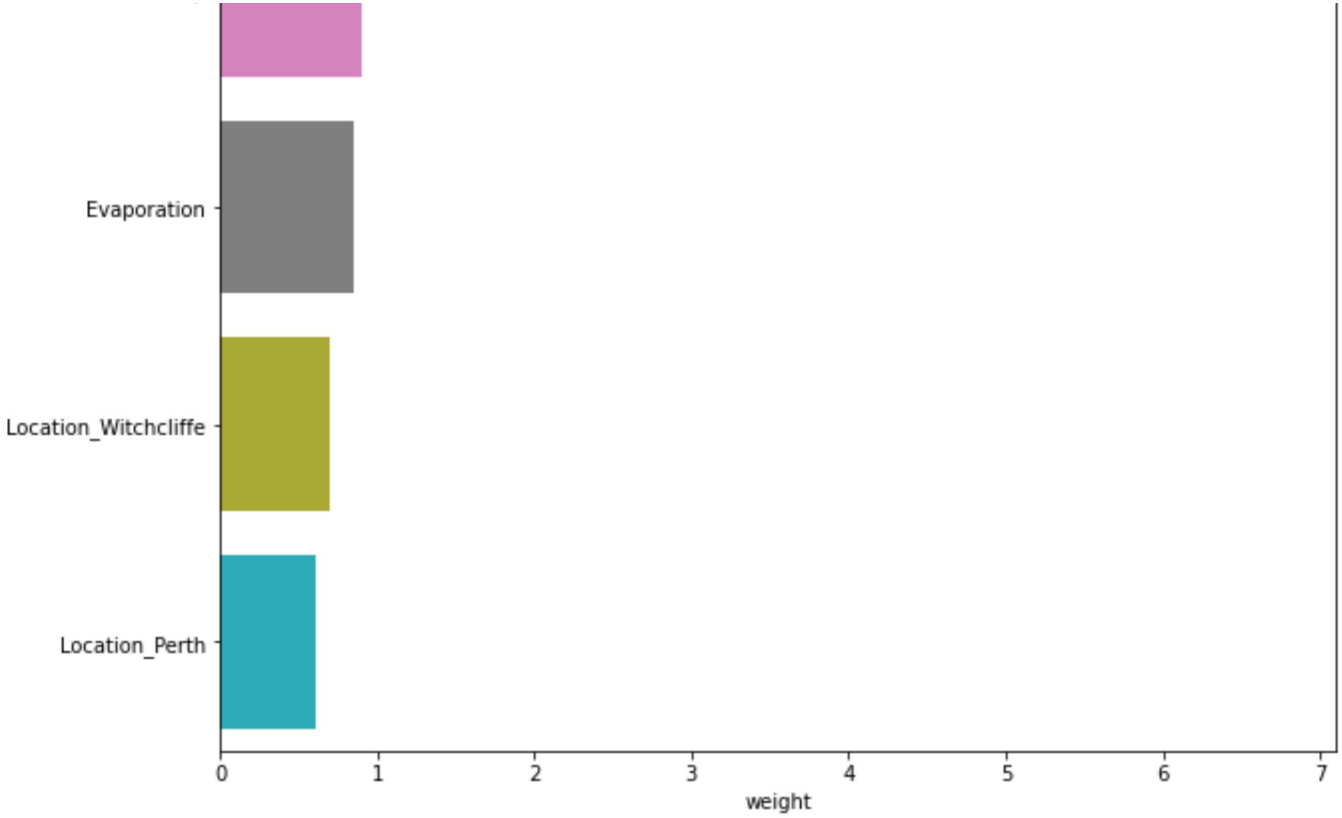
In [53]:
```
# we repeat the optimization method many many times, and that is what max_iter denotes - how many times will the mode
# be optimized before the output is finalized
# the model is not optimized a lot of time because that leads to overfitting
```

In [54]:
```
# visualizing data with bar_edge graph
```

In [55]:
```
plt.figure(figsize=(10,20))
sns.barplot(data=weight_df.sort_values('weight',ascending=False).head(10),x='weight',y='feature')
```

Out[55]:
```
<AxesSubplot:xlabel='weight', ylabel='feature'>
```


WindGustSpeed
Humidity3pm
Pressure9am
Rainfall
Temp3pm
Cloud3pm
MinTemp
feature



Evaluate the model

In [56]:
```
# evaluate on training test and validation set
```

In [57]:
```
# using the trained model to make predictions on the training, test and validation set
```

In [58]:
```
x_train = train_inputs[numeric_cols+encoded_cols]
x_val = val_input[numeric_cols+encoded_cols]
x_test = test_input[numeric_cols+encoded_cols]
```

In [59]:
```
# predicting on training set
train_preds = model.predict(x_train)
# here, we only give a lift of inputs to the model to predict the targets - as output, our model gives us a list of p
```

In [60]:
```
# these are the predicted outputs that our model has created
train_preds
```

Out[60]:
```
array(['No', 'No', 'No', ..., 'No', 'No', 'No'], dtype=object)
```

In [61]:
```
# checking accuracy
from sklearn.metrics import accuracy_score
```

In [62]:
```
# train_targets => the actual Y values in our training set which we have actually observed
# train_preds => the predicted Y values that have been predicted by our model
accuracy_score(train_targets,train_preds)
# we can see our projected accuracy is 85%, which sounds great but remember that our dataset is imbalanced towards No
# so we should check other parameters also
```

Out[62]:
```
0.8519206433440829
```

In [63]:
```
# for logReg, we can also output probabilities of each row's predicted Y using predict_proba
train_probs = model.predict_proba(x_train)
train_probs
# this gives us the probability of achieving a Y or N on that particular day with [a,b] where a% is the % of No and b
```

Out[63]:
```
array([[0.94401144, 0.05598856],
       [0.94074129, 0.05925871],
       [0.96093612, 0.03906388],
       ...,
       [0.98749101, 0.01250899],
       [0.98334667, 0.01665333],
       [0.87453315, 0.12546685]])
```

In [64]:
```
# we should break down the accuracy into a confusion matrix to do further analysis
from sklearn.metrics import confusion_matrix
```

In [65]:
```
confusion_matrix(train_targets,train_preds,normalize='true')
```

Out[65]:
```
array([[0.94621341, 0.05378659],
       [0.4776585 , 0.5223415 ]])
```

In [66]:
```
confusion_matrix(train_targets,train_preds)
```

Out[66]:
```
array([[72092,  4098],
       [10412, 11386]], dtype=int64)
```

In [67]:
```
# to make the model better, we can work on reducing our False Negative but otherwise our accuracy seems good
```

Comparing model accuracy with NULL accuracy

In [68]:
```
#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 [69]:
```
rain_df['RainTomorrow'].value_counts()
```

Out[69]:
```
No     109586
Yes     31201
Name: RainTomorrow, dtype: int64
```

In [70]:
```
null_accuracy = (109586/(109586+31201))
print('Null accuracy score: {0:0.4f}'. format(null_accuracy))
```

```
Null accuracy score: 0.7784
```

Interpretation of NULL model

In [71]:
```
#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.
```

In [72]:
```
# depending on kind of problem - we attempt at reducing the False Positive or False Negative
# FN is reduced when we wish to make accurate predictions about our Y
# FP is reduced when we wish to be absolutely confident that our predicted Y is actually Y
```

Saving model to disk

In [73]:
```
import joblib
```

In [74]:
```
# creating dictionary to save everything:

aussie_rain = {
    'model' : model,
    'imputer' : imputer,
    'scaler' : scaler,
    'encoder' : encoder,
    'input_cols' : input_cols,
    'target_col' : target_col,
    'numeric_cols' : numeric_cols,
    'categorical_cols' : categorical_cols,
    'encoded_cols' : encoded_cols
}
```

In [75]:
```
# we can now save this to a file using joblib.dump

joblib.dump(aussie_rain,'aussie_rain.joblib')
```

Out[75]:
```
['aussie_rain.joblib']
```

In [76]:
```
# the object can be loaded back using joblib.load

aussie_rain2 = joblib.load('aussie_rain.joblib')
```