In [1]:
```
import warnings
# Ignore all warnings
warnings.filterwarnings("ignore")
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
```

RAIN PREDICTION

TABLE OF CONTENTS

LIBRARIES

IMPORTING LIBRARIES

In [2]:
```
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
from sklearn.preprocessing import LabelEncoder
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import seaborn as sns
from keras.layers import Dense, BatchNormalization, Dropout, LSTM
from keras.models import Sequential
from keras.utils import to_categorical
from keras.optimizers import Adam
from tensorflow.keras import regularizers
from sklearn.metrics import precision_score, recall_score, confusion_matrix, classification_report, accuracy_score, f1
from keras import callbacks

np.random.seed(0)
```

LOADING DATA

LOADING DATA

In [3]:
```
data = pd.read_csv("weatherAUS.csv")
data.head()
```

Out[3]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	...	Humidity9am	H
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	W	...	71.0	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	NNW	...	44.0	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	W	...	38.0	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	SE	...	45.0	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	ENE	...	82.0	

5 rows × 23 columns

About the data:

The dataset contains about 10 years of daily weather observations from different locations across Australia. Observations were drawn from numerous weather stations.

In this project, I will use this data to predict whether or not it will rain the next day. There are 23 attributes including the target variable "RainTomorrow", indicating whether or not it will rain the next day or not.

In [4]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype  
---  ------         --------------   -----  
 0   Date           145460 non-null  object 
 1   Location       145460 non-null  object 
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object 
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object 
 10  WindDir3pm     141232 non-null  object 
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object 
 22  RainTomorrow   142193 non-null  object 
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

Points to notice:

- There are missing values in the dataset
- Dataset includes numeric and categorical values

DATA VISUALIZATION AND CLEANING

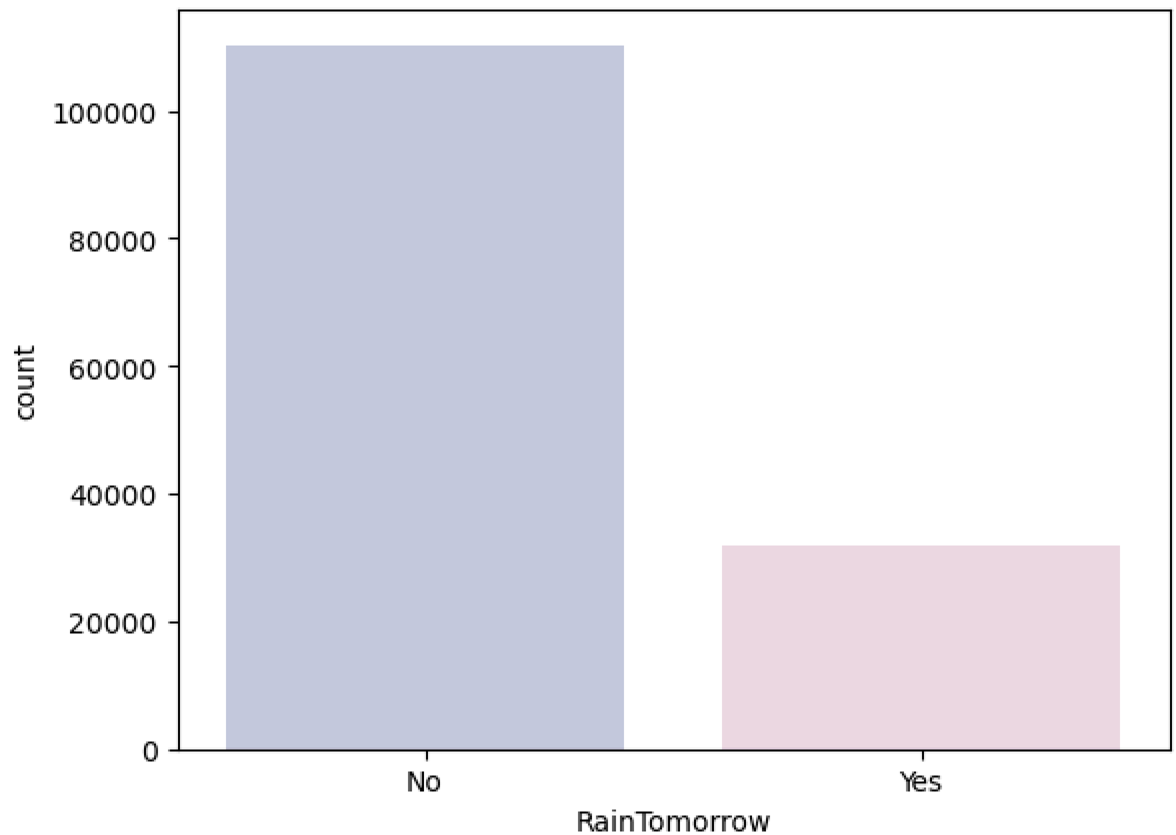
DATA VISUALIZATION AND CLEANING

Steps involves in this section:

- Count plot of target column
- Correlation amongst numeric attributes
- Parse Dates into datetime
- Encoding days and months as continuous cyclic features

In [5]:
```
#first of all let us evaluate the target and find out if our data is imbalanced or not
cols= ["#C2C4E2","#EED4E5"]
sns.countplot(x= data["RainTomorrow"], palette= cols)
```

Out[5]:
```
<AxesSubplot:xlabel='RainTomorrow', ylabel='count'>
```

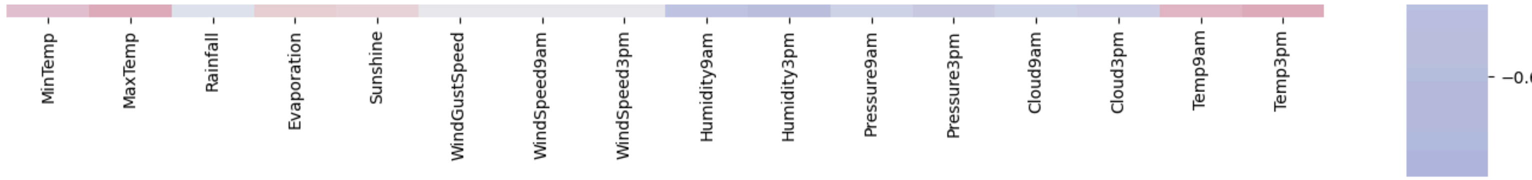


In [6]:
```
# Correlation amongst numeric attributes
corrmat = data.corr()
cmap = sns.diverging_palette(260,-10,s=50, l=75, n=6, as_cmap=True)
```

```
plt.subplots(figsize=(18,18))
sns.heatmap(corrmat,cmap= cmap,annot=True, square=True)
```

Out[6]: <AxesSubplot:>


MinTemp 1 0.74 0.1 0.47 0.073 0.18 0.18 0.18 -0.23 0.0061 -0.45 -0.46 0.079 0.022 0.9 0.71
MaxTemp 0.74 1 -0.075 0.59 0.47 0.068 0.014 0.05 -0.5 -0.51 -0.33 -0.43 -0.29 -0.28 0.89 0.98
Rainfall 0.1 -0.075 1 -0.064 -0.23 0.13 0.087 0.058 0.22 0.26 -0.17 -0.13 0.2 0.17 0.011 -0.08
Evaporation 0.47 0.59 -0.064 1 0.37 0.2 0.19 0.13 -0.5 -0.39 -0.27 -0.29 -0.18 -0.18 0.55 0.57
Sunshine 0.073 0.47 -0.23 0.37 1 -0.035 0.0055 0.054 -0.49 -0.63 0.042 -0.02 -0.68 -0.7 0.29 0.49
WindGustSpeed 0.18 0.068 0.13 0.2 -0.035 1 0.61 0.69 -0.22 -0.026 -0.46 -0.41 0.072 0.11 0.15 0.033
WindSpeed9am 0.18 0.014 0.087 0.19 0.0055 0.61 1 0.52 -0.27 -0.032 -0.23 -0.18 0.025 0.055 0.13 0.0046
WindSpeed3pm 0.18 0.05 0.058 0.13 0.054 0.69 0.52 1 -0.15 0.016 -0.3 -0.26 0.053 0.025 0.16 0.028
Humidity9am -0.23 -0.5 0.22 -0.5 -0.49 -0.22 -0.27 -0.15 1 0.67 0.14 0.19 0.45 0.36 -0.47 -0.5
Humidity3pm 0.0061 -0.51 0.26 -0.39 -0.63 -0.026 -0.032 0.016 0.67 1 -0.028 0.052 0.52 0.52 -0.22 -0.56
Pressure9am -0.45 -0.33 -0.17 -0.27 0.042 -0.46 -0.23 -0.3 0.14 -0.028 1 0.96 -0.13 -0.15 -0.42 -0.29
Pressure3pm -0.46 -0.43 -0.13 -0.29 -0.02 -0.41 -0.18 -0.26 0.19 0.052 0.96 1 -0.061 -0.085 -0.47 -0.39
Cloud9am 0.079 -0.29 0.2 -0.18 -0.68 0.072 0.025 0.053 0.45 0.52 -0.13 -0.061 1 0.6 -0.14 -0.3
Cloud3pm 0.022 -0.28 0.17 -0.18 -0.7 0.11 0.055 0.025 0.36 0.52 -0.15 -0.085 0.6 1 -0.13 -0.32
Temp9am 0.9 0.89 0.011 0.55 0.29 0.15 0.13 0.16 -0.47 -0.22 -0.42 -0.47 -0.14 -0.13 1 0.86
Temp3pm 0.71 0.98 -0.08 0.57 0.49 0.033 0.0046 0.028 -0.5 -0.56 -0.29 -0.39 -0.3 -0.32 0.86 1
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4



Now I will parse Dates into datetime.

My goal is to build an artificial neural network(ANN). I will encode dates appropriately, i.e. I prefer the months and days in a cyclic continuous feature. As, date and time are inherently cyclical. To let the ANN model know that a feature is cyclical I split it into periodic subsections. Namely, years, months and days. Now for each subsection, I create two new features, deriving a sine transform and cosine transform of the subsection feature.

In [7]:
```
#Parsing datetime
#exploring the length of date objects
lengths = data["Date"].str.len()
lengths.value_counts()
```

Out[7]:
```
10    145460
Name: Date, dtype: int64
```

In [8]:
```
#There don't seem to be any error in dates so parsing values into datetime
data['Date']= pd.to_datetime(data["Date"])
#Creating a collumn of year
data['year'] = data.Date.dt.year

# function to encode datetime into cyclic parameters.
#As I am planning to use this data in a neural network I prefer the months and days in a cyclic continuous feature.

def encode(data, col, max_val):
    data[col + '_sin'] = np.sin(2 * np.pi * data[col]/max_val)
    data[col + '_cos'] = np.cos(2 * np.pi * data[col]/max_val)
    return data

data['month'] = data.Date.dt.month
data = encode(data, 'month', 12)

data['day'] = data.Date.dt.day
data = encode(data, 'day', 31)

data.head()
```

Out[8]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am	...	Temp3pm	RainT
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	W	...	21.8	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	NNW	...	24.3	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	W	...	23.2	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	SE	...	26.5	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	ENE	...	29.7	

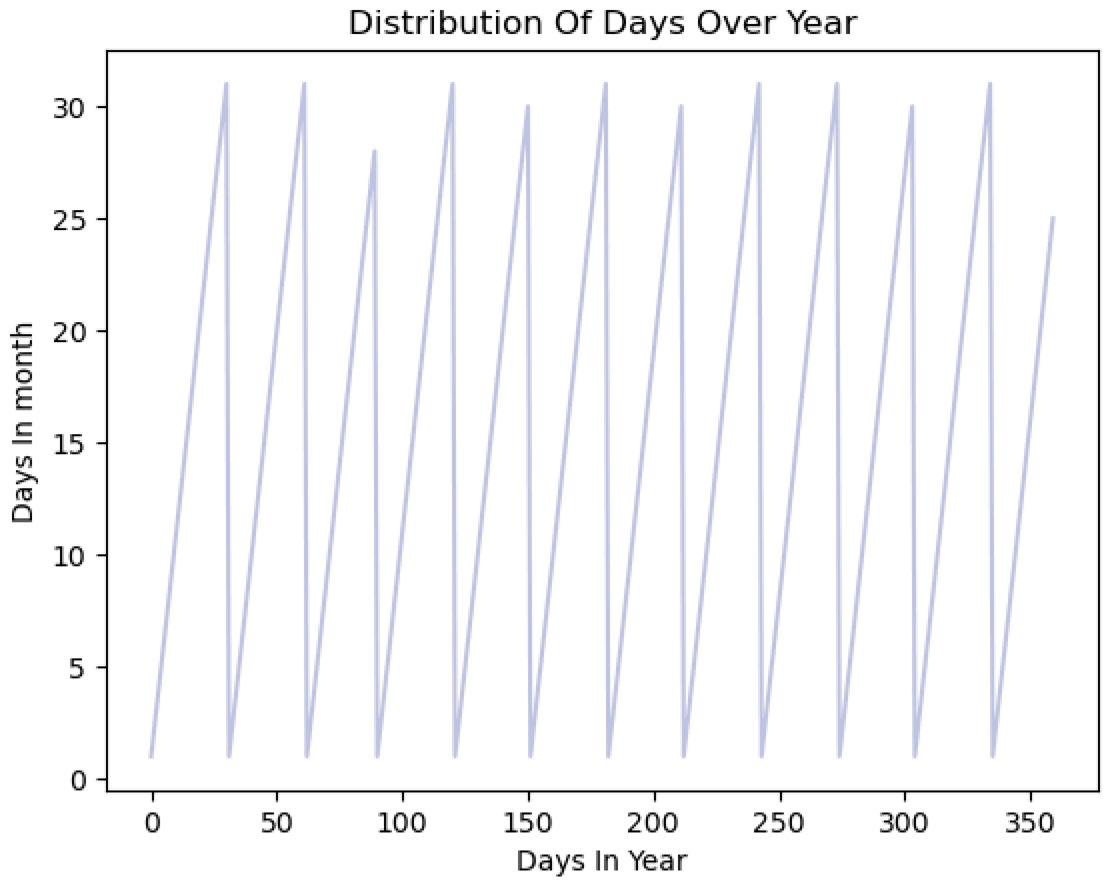
5 rows × 30 columns

In [9]:

```
# roughly a year's span section
section = data[:360]
tm = section["day"].plot(color="#C2C4E2")
tm.set_title("Distribution Of Days Over Year")
tm.set_ylabel("Days In month")
tm.set_xlabel("Days In Year")
```

Out[9]:

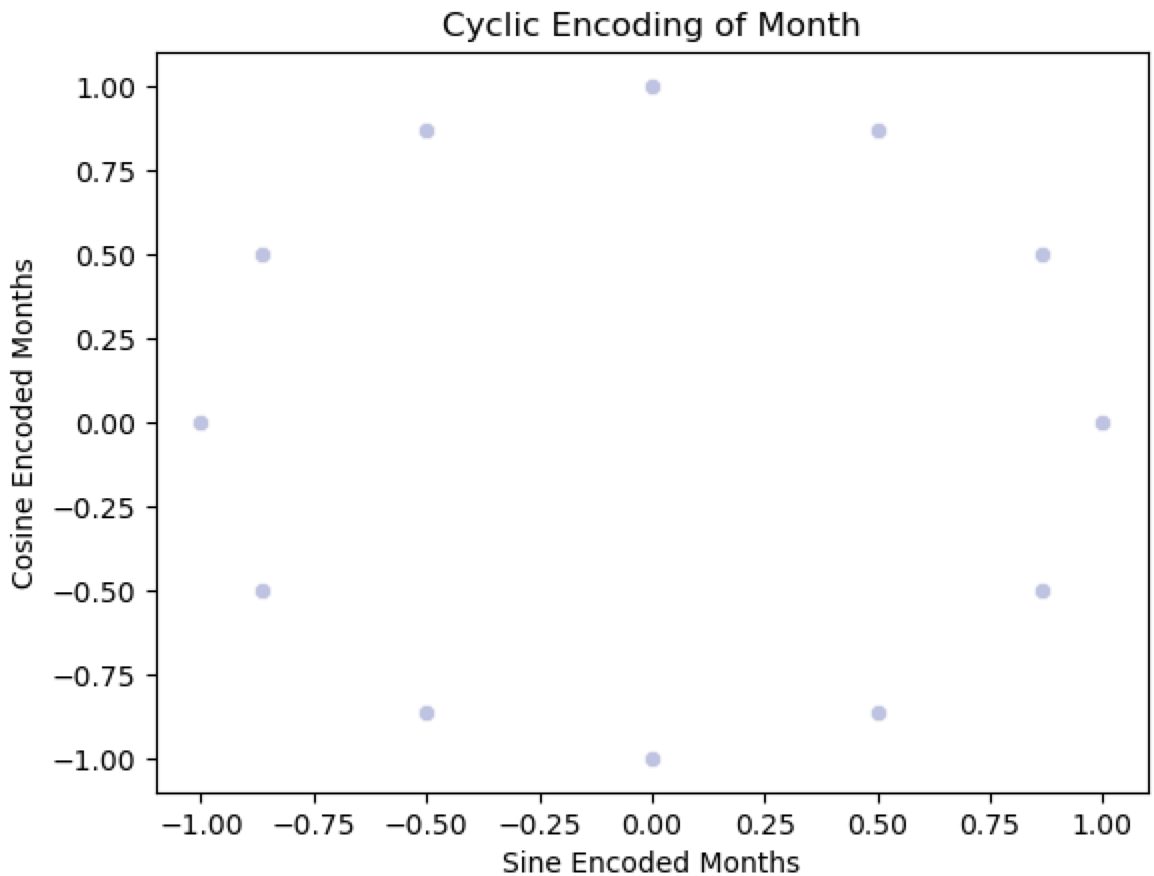
```
Text(0.5, 0, 'Days In Year')
```



As expected, the "year" attribute of data repeats. However in this for the true cyclic nature is not presented in a continuous manner. Splitting months and days into Sine and cosine combination provides the cyclical continuous feature. This can be used as input features to ANN.

In [10]:
```
cyclic_month = sns.scatterplot(x="month_sin",y="month_cos",data=data, color="#C2C4E2")
cyclic_month.set_title("Cyclic Encoding of Month")
cyclic_month.set_ylabel("Cosine Encoded Months")
cyclic_month.set_xlabel("Sine Encoded Months")
```

Out[10]:
```
Text(0.5, 0, 'Sine Encoded Months')
```

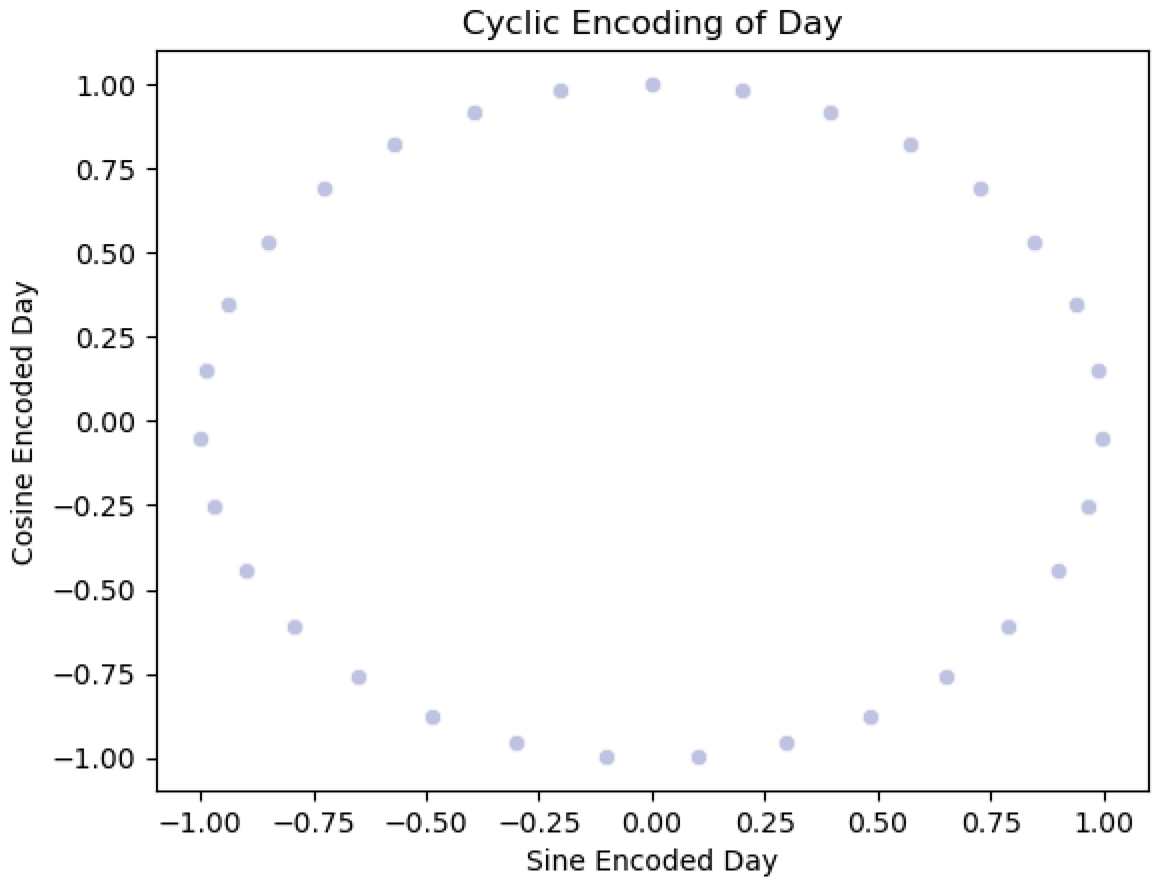


In [11]:

```
cyclic_day = sns.scatterplot(x='day_sin',y='day_cos',data=data, color="#C2C4E2")
cyclic_day.set_title("Cyclic Encoding of Day")
cyclic_day.set_ylabel("Cosine Encoded Day")
cyclic_day.set_xlabel("Sine Encoded Day")
```

Out[11]:

```
Text(0.5, 0, 'Sine Encoded Day')
```



Next, I will deal with missing values in categorical and numeric attributes separately

Categorical variables

- Filling missing values with mode of the column value

In [12]:
```
# Get list of categorical variables
s = (data.dtypes == "object")
object_cols = list(s[s].index)

print("Categorical variables:")
print(object_cols)
```

```
Categorical variables:
['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday', 'RainTomorrow']
```

In [13]:
```
# Missing values in categorical variables

for i in object_cols:
    print(i, data[i].isnull().sum())
```

```
Location 0
WindGustDir 10326
WindDir9am 10566
WindDir3pm 4228
RainToday 3261
RainTomorrow 3267
```

In [14]:
```
# Filling missing values with mode of the column in value

for i in object_cols:
    data[i].fillna(data[i].mode()[0], inplace=True)
```

Numerical variables

- Filling missing values with median of the column value

In [15]:
```
# Get list of neumeric variables
t = (data.dtypes == "float64")
num_cols = list(t[t].index)

print("Neumeric variables:")
print(num_cols)
```

```
Neumeric variables:
['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine', 'WindGustSpeed', 'WindSpeed9am', 'WindSpeed3pm', 'Humidi
ty9am', 'Humidity3pm', 'Pressure9am', 'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am', 'Temp3pm', 'month_sin', 'month
_cos', 'day_sin', 'day_cos']
```

In [16]:
```
# Missing values in numeric variables

for i in num_cols:
    print(i, data[i].isnull().sum())
```

```
MinTemp 1485
MaxTemp 1261
Rainfall 3261
Evaporation 62790
Sunshine 69835
WindGustSpeed 10263
WindSpeed9am 1767
WindSpeed3pm 3062
Humidity9am 2654
Humidity3pm 4507
Pressure9am 15065
Pressure3pm 15028
Cloud9am 55888
Cloud3pm 59358
Temp9am 1767
Temp3pm 3609
month_sin 0
month_cos 0
day_sin 0
day_cos 0
```

In [17]:

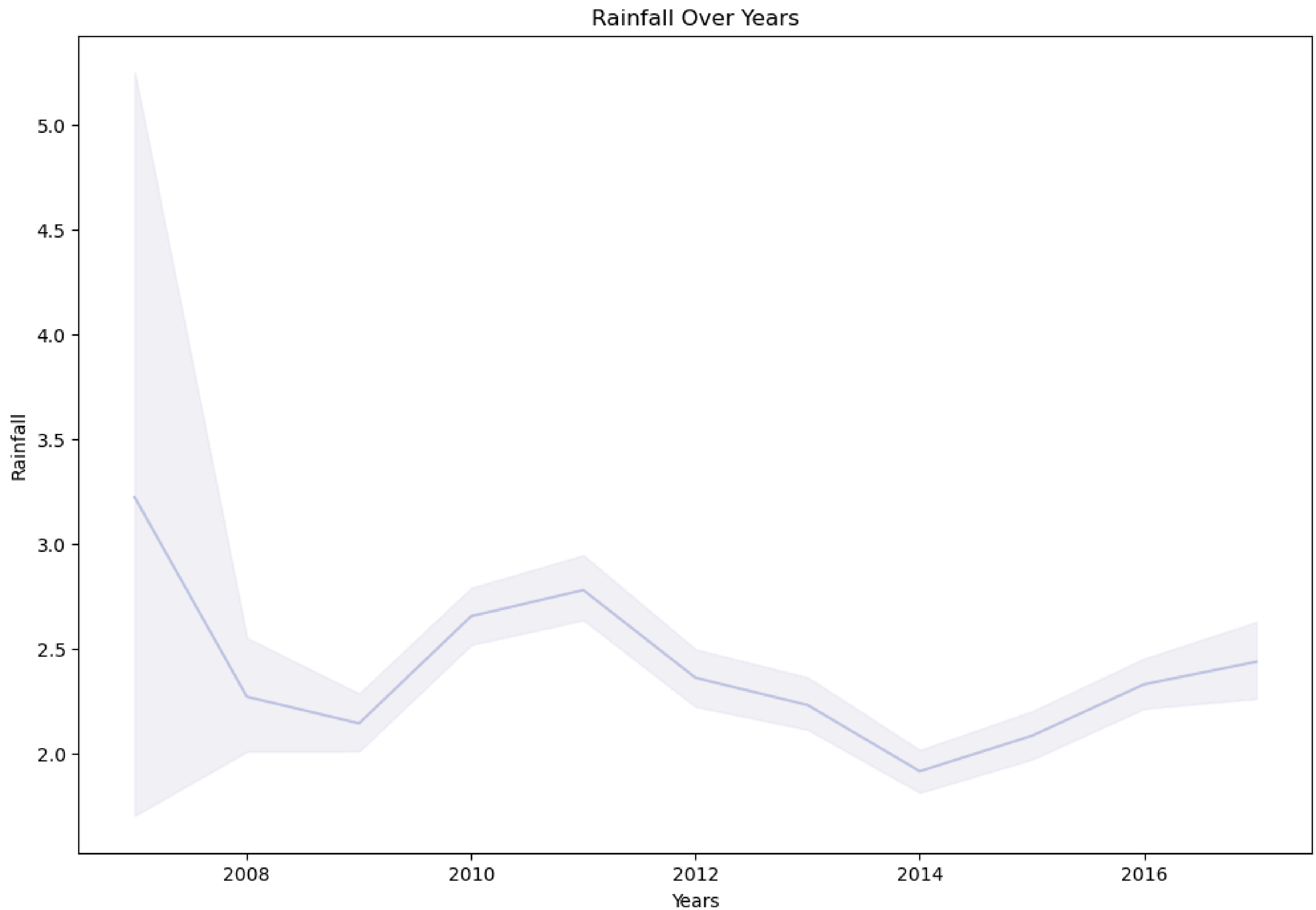
```python
# Filling missing values with median of the column in value

for i in num_cols:
    data[i].fillna(data[i].median(), inplace=True)

data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 30 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           145460 non-null  datetime64[ns]
 1   Location       145460 non-null  object
 2   MinTemp        145460 non-null  float64
 3   MaxTemp        145460 non-null  float64
 4   Rainfall       145460 non-null  float64
 5   Evaporation    145460 non-null  float64
 6   Sunshine       145460 non-null  float64
 7   WindGustDir    145460 non-null  object
 8   WindGustSpeed  145460 non-null  float64
 9   WindDir9am     145460 non-null  object
 10  WindDir3pm     145460 non-null  object
 11  WindSpeed9am   145460 non-null  float64
 12  WindSpeed3pm   145460 non-null  float64
 13  Humidity9am    145460 non-null  float64
 14  Humidity3pm    145460 non-null  float64
 15  Pressure9am    145460 non-null  float64
 16  Pressure3pm    145460 non-null  float64
 17  Cloud9am       145460 non-null  float64
 18  Cloud3pm       145460 non-null  float64
 19  Temp9am        145460 non-null  float64
 20  Temp3pm        145460 non-null  float64
 21  RainToday      145460 non-null  object
 22  RainTomorrow   145460 non-null  object
 23  year           145460 non-null  int64
 24  month          145460 non-null  int64
 25  month_sin      145460 non-null  float64
 26  month_cos      145460 non-null  float64
 27  day            145460 non-null  int64
 28  day_sin        145460 non-null  float64
 29  day_cos        145460 non-null  float64
dtypes: datetime64[ns](1), float64(20), int64(3), object(6)
memory usage: 33.3+ MB
```

In [18]:

```python
#plotting a lineplot rainfall over years
plt.figure(figsize=(12,8))
Time_series=sns.lineplot(x=data['Date'].dt.year,y="Rainfall",data=data,color="#C2C4E2")
Time_series.set_title("Rainfall Over Years")
Time_series.set_ylabel("Rainfall")
Time_series.set_xlabel("Years")
```

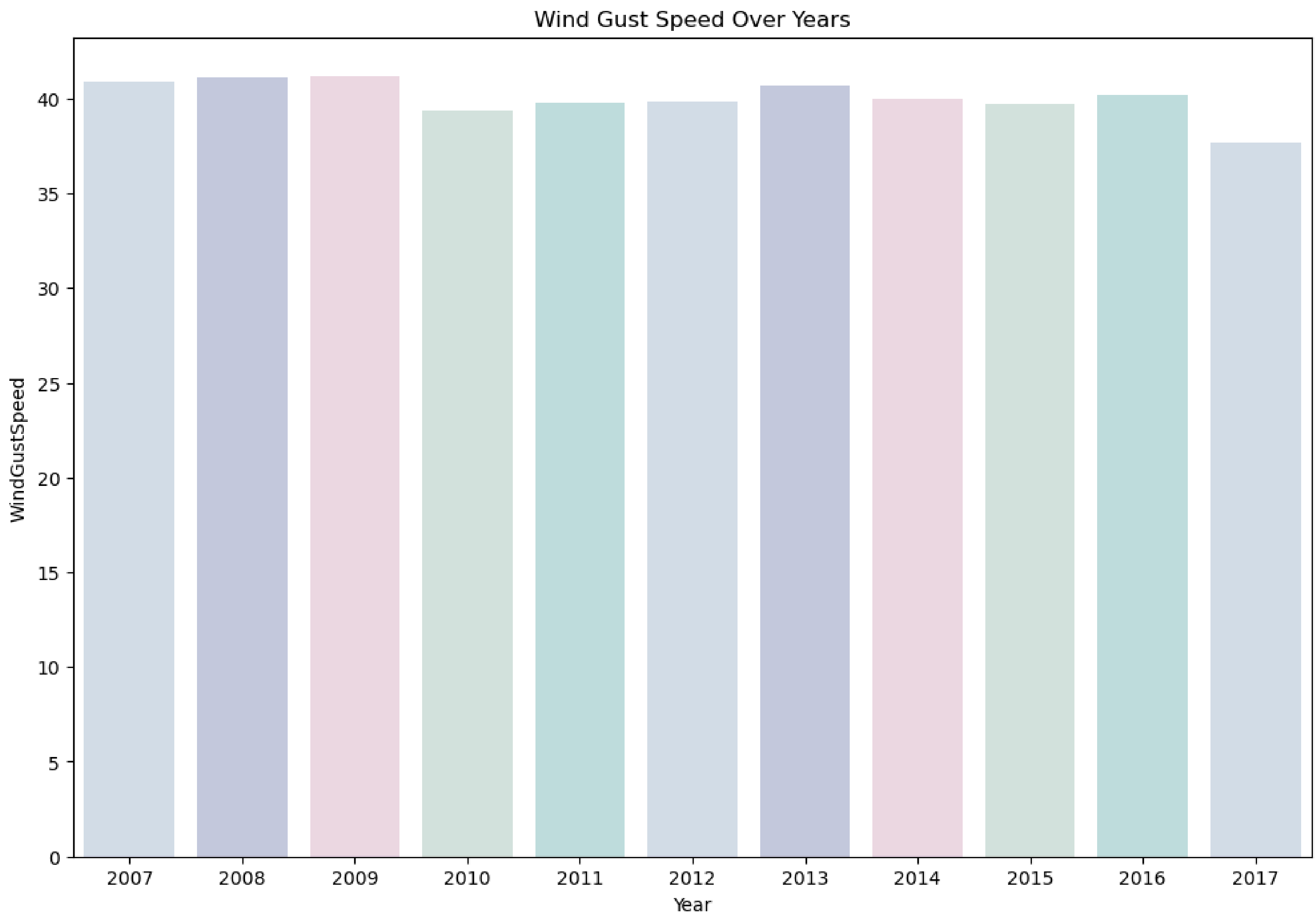
Out[18]: Text(0.5, 0, 'Years')



In [19]:
```
#Evauating Wind gust speed over years
colours = ["#D0DBEE", "#C2C4E2", "#EED4E5", "#D1E6DC", "#BDE2E2"]
plt.figure(figsize=(12,8))
Days_of_week=sns.barplot(x=data['Date'].dt.year,y="WindGustSpeed",data=data, ci =None,palette = colours)
Days_of_week.set_title("Wind Gust Speed Over Years")
```

```
Days_of_week.set_ylabel("WindGustSpeed")
Days_of_week.set_xlabel("Year")
```

Out[19]: Text(0.5, 0, 'Year')



DATA PREPROCESSING

DATA PREPROCESSING

Steps involved in Data Preprocessing:

- Label encoding columns with categorical data
- Perform the scaling of the features
- Detecting outliers
- Dropping the outliers based on data analysis

Label encoding the catagorical varable

In [20]:
```
# Apply label encoder to each column with categorical data
label_encoder = LabelEncoder()
for i in object_cols:
    data[i] = label_encoder.fit_transform(data[i])

data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 30 columns):
 #   Column         Non-Null Count   Dtype
---  ------         --------------   -----
 0   Date           145460 non-null  datetime64[ns]
 1   Location       145460 non-null  int32
 2   MinTemp        145460 non-null  float64
 3   MaxTemp        145460 non-null  float64
 4   Rainfall       145460 non-null  float64
 5   Evaporation    145460 non-null  float64
 6   Sunshine       145460 non-null  float64
 7   WindGustDir    145460 non-null  int32
 8   WindGustSpeed  145460 non-null  float64
 9   WindDir9am     145460 non-null  int32
 10  WindDir3pm     145460 non-null  int32
 11  WindSpeed9am   145460 non-null  float64
 12  WindSpeed3pm   145460 non-null  float64
 13  Humidity9am    145460 non-null  float64
 14  Humidity3pm    145460 non-null  float64
 15  Pressure9am    145460 non-null  float64
 16  Pressure3pm    145460 non-null  float64
 17  Cloud9am       145460 non-null  float64
 18  Cloud3pm       145460 non-null  float64
 19  Temp9am        145460 non-null  float64
 20  Temp3pm        145460 non-null  float64
 21  RainToday      145460 non-null  int32
 22  RainTomorrow   145460 non-null  int32
 23  year           145460 non-null  int64
 24  month          145460 non-null  int64
 25  month_sin      145460 non-null  float64
 26  month_cos      145460 non-null  float64
 27  day            145460 non-null  int64
 28  day_sin        145460 non-null  float64
 29  day_cos        145460 non-null  float64
dtypes: datetime64[ns](1), float64(20), int32(6), int64(3)
memory usage: 30.0 MB
```

In [21]:

```
# Prepairing attributes of scale data

features = data.drop(['RainTomorrow', 'Date','day', 'month'], axis=1) # dropping target and extra columns

target = data['RainTomorrow']

#Set up a standard scaler for the features
```

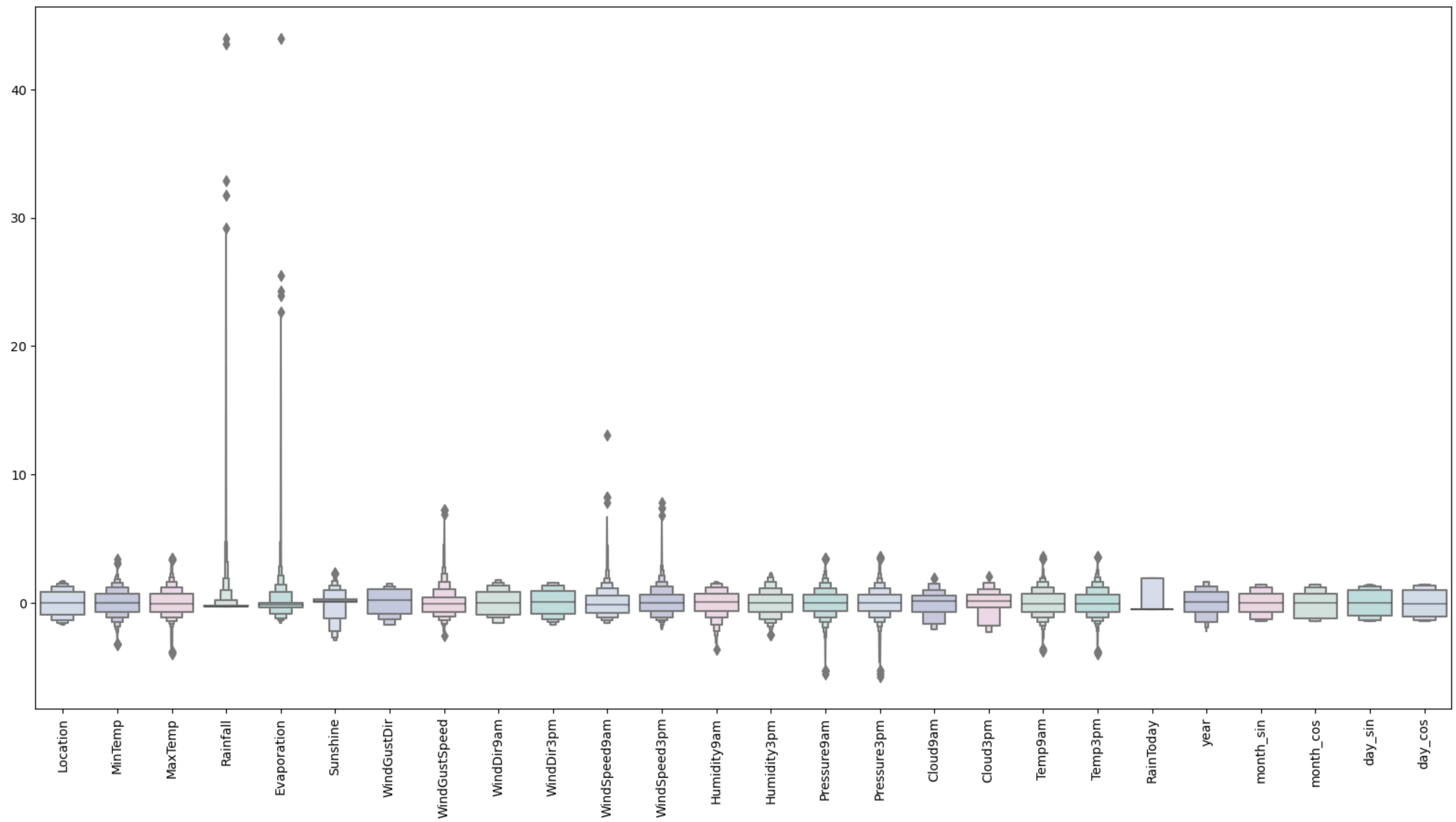
```
col_names = list(features.columns)
s_scaler = preprocessing.StandardScaler()
features = s_scaler.fit_transform(features)
features = pd.DataFrame(features, columns=col_names)

features.describe().T
```

Out[21]:

	count	mean	std	min	25%	50%	75%	max
Location	145460.0	-5.633017e-14	1.000003	-1.672228	-0.899139	0.014511	0.857881	1.701250
MinTemp	145460.0	-4.243854e-15	1.000003	-3.250525	-0.705659	-0.030170	0.723865	3.410112
MaxTemp	145460.0	6.513740e-16	1.000003	-3.952405	-0.735852	-0.086898	0.703133	3.510563
Rainfall	145460.0	9.152711e-15	1.000003	-0.275097	-0.275097	-0.275097	-0.203581	43.945571
Evaporation	145460.0	1.352327e-14	1.000003	-1.629472	-0.371139	-0.119472	0.006361	43.985108
Sunshine	145460.0	-4.338304e-15	1.000003	-2.897217	0.076188	0.148710	0.257494	2.360634
WindGustDir	145460.0	1.864381e-14	1.000003	-1.724209	-0.872075	0.193094	1.045228	1.471296
WindGustSpeed	145460.0	-1.167921e-14	1.000003	-2.588407	-0.683048	-0.073333	0.460168	7.243246
WindDir9am	145460.0	-7.433272e-15	1.000003	-1.550000	-0.885669	0.000105	0.885879	1.771653
WindDir3pm	145460.0	1.791486e-15	1.000003	-1.718521	-0.837098	0.044324	0.925747	1.586813
WindSpeed9am	145460.0	-3.422029e-14	1.000003	-1.583291	-0.793380	-0.116314	0.560752	13.086472
WindSpeed3pm	145460.0	1.618238e-14	1.000003	-2.141841	-0.650449	0.037886	0.611499	7.839016
Humidity9am	145460.0	-4.803490e-15	1.000003	-3.654212	-0.631189	0.058273	0.747734	1.649338
Humidity3pm	145460.0	-6.041889e-15	1.000003	-2.518329	-0.710918	0.021816	0.656852	2.366565
Pressure9am	145460.0	2.313398e-14	1.000003	-5.520544	-0.616005	-0.006653	0.617561	3.471111
Pressure3pm	145460.0	4.709575e-15	1.000003	-5.724832	-0.622769	-0.007520	0.622735	3.653960
Cloud9am	145460.0	-2.525820e-14	1.000003	-2.042425	-0.727490	0.149133	0.587445	1.902380
Cloud3pm	145460.0	4.796901e-15	1.000003	-2.235619	-0.336969	0.137693	0.612356	2.036343
Temp9am	145460.0	-3.332880e-15	1.000003	-3.750358	-0.726764	-0.044517	0.699753	3.599302
Temp3pm	145460.0	-2.901899e-15	1.000003	-3.951301	-0.725322	-0.083046	0.661411	3.653834
RainToday	145460.0	1.263303e-14	1.000003	-0.529795	-0.529795	-0.529795	-0.529795	1.887521
year	145460.0	1.663818e-14	1.000003	-2.273637	-0.697391	0.090732	0.878855	1.666978
month_sin	145460.0	1.478014e-15	1.000003	-1.434333	-0.725379	-0.016425	0.692529	1.401483
month_cos	145460.0	4.043483e-16	1.000003	-1.388032	-1.198979	0.023080	0.728636	1.434192
day_sin	145460.0	-6.534944e-18	1.000003	-1.403140	-1.019170	-0.003198	1.012774	1.396744

	count	mean	std	min	25%	50%	75%	max
day_cos	145460.0	-9.540621e-19	1.000003	-1.392587	-1.055520	-0.044639	1.011221	1.455246

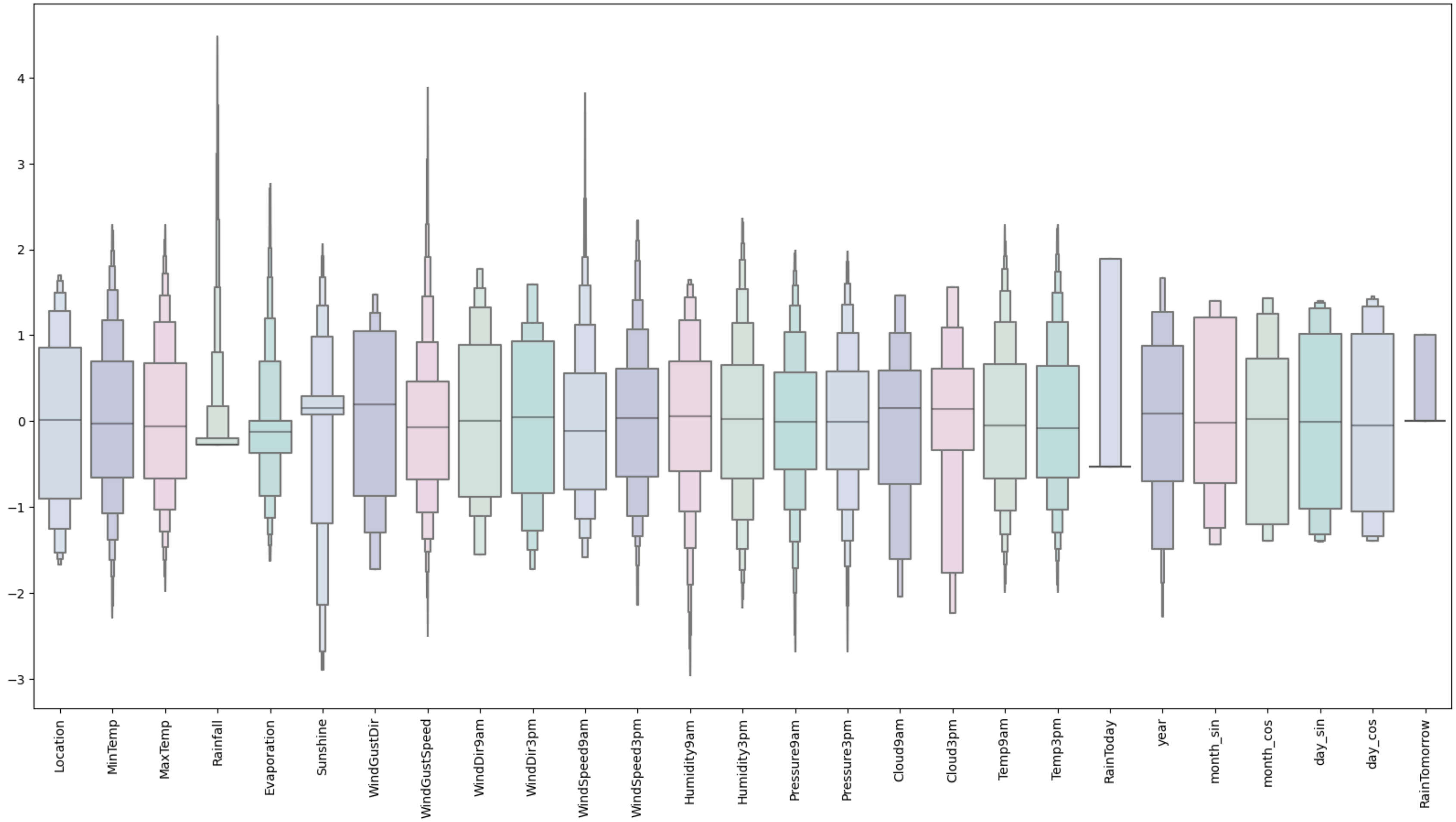
In [22]:
```
#Detecting outliers
#looking at the scaled features
colours = ["#D0DBEE", "#C2C4E2", "#EED4E5", "#D1E6DC", "#BDE2E2"]
plt.figure(figsize=(20,10))
sns.boxenplot(data = features,palette = colours)
plt.xticks(rotation=90)
plt.show()
```



In [23]:
```
#full data for
features["RainTomorrow"] = target

#Dropping with outlier

features = features[(features["MinTemp"]<2.3)&(features["MinTemp"]>-2.3)]
features = features[(features["MaxTemp"]<2.3)&(features["MaxTemp"]>-2)]
features = features[(features["Rainfall"]<4.5)]
features = features[(features["Evaporation"]<2.8)]
features = features[(features["Sunshine"]<2.1)]
features = features[(features["WindGustSpeed"]<4)&(features["WindGustSpeed"]>-4)]
features = features[(features["WindSpeed9am"]<4)]
features = features[(features["WindSpeed3pm"]<2.5)]
features = features[(features["Humidity9am"]>-3)]
features = features[(features["Humidity3pm"]>-2.2)]
features = features[(features["Pressure9am"]< 2)&(features["Pressure9am"]>-2.7)]
features = features[(features["Pressure3pm"]< 2)&(features["Pressure3pm"]>-2.7)]
features = features[(features["Cloud9am"]<1.8)]
features = features[(features["Cloud3pm"]<2)]
features = features[(features["Temp9am"]<2.3)&(features["Temp9am"]>-2)]
features = features[(features["Temp3pm"]<2.3)&(features["Temp3pm"]>-2)]


features.shape
```

Out[23]:
```
(127536, 27)
```

In [24]:
```
#looking at the scaled features without outliers

plt.figure(figsize=(20,10))
sns.boxenplot(data = features,palette = colours)
plt.xticks(rotation=90)
plt.show()
```



Looks Good. Up next is building artificial neural network.

MODEL BUILDING

MODEL BUILDING

In this project, we build an artificial neural network.

Following steps are involved in the model building

- Assining X and y the status of attributes and tags
- Splitting test and training sets
- Initialising the neural network
- Defining by adding layers
- Compiling the neural network
- Train the neural network

In [25]:
```
X = features.drop(["RainTomorrow"], axis=1)
y = features["RainTomorrow"]

# Splitting test and training sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 42)

X.shape
```

Out[25]:
```
(127536, 26)
```

In [26]:
```
#Early stopping
early_stopping = callbacks.EarlyStopping(
    min_delta=0.001, # minimium amount of change to count as an improvement
    patience=20, # how many epochs to wait before stopping
    restore_best_weights=True,
)

# Initialising the NN
model = Sequential()

# layers

model.add(Dense(units = 32, kernel_initializer = 'uniform', activation = 'relu', input_dim = 26))
model.add(Dense(units = 32, kernel_initializer = 'uniform', activation = 'relu'))
model.add(Dense(units = 16, kernel_initializer = 'uniform', activation = 'relu'))
model.add(Dropout(0.25))
model.add(Dense(units = 8, kernel_initializer = 'uniform', activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(units = 1, kernel_initializer = 'uniform', activation = 'sigmoid'))

# Compiling the ANN
opt = Adam(learning_rate=0.00009)
model.compile(optimizer = opt, loss = 'binary_crossentropy', metrics = ['accuracy'])
```

```
# Train the ANN
history = model.fit(X_train, y_train, batch_size = 32, epochs = 150, callbacks=[early_stopping], validation_split=0.2)
```

```
Epoch 1/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 8s 2ms/step - accuracy: 0.7821 - loss: 0.5654 - val_accuracy: 0.7860 - val_loss: 0.3927
Epoch 2/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7842 - loss: 0.4150 - val_accuracy: 0.7860 - val_loss: 0.3878
Epoch 3/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7852 - loss: 0.4112 - val_accuracy: 0.7860 - val_loss: 0.3841
Epoch 4/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7820 - loss: 0.4118 - val_accuracy: 0.7860 - val_loss: 0.3821
Epoch 5/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7847 - loss: 0.4058 - val_accuracy: 0.7860 - val_loss: 0.3805
Epoch 6/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.7861 - loss: 0.4017 - val_accuracy: 0.8433 - val_loss: 0.3792
Epoch 7/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8420 - loss: 0.3998 - val_accuracy: 0.8435 - val_loss: 0.3775
Epoch 8/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8405 - loss: 0.3978 - val_accuracy: 0.8440 - val_loss: 0.3765
Epoch 9/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8409 - loss: 0.4007 - val_accuracy: 0.8430 - val_loss: 0.3757
Epoch 10/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8395 - loss: 0.4003 - val_accuracy: 0.8444 - val_loss: 0.3748
Epoch 11/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8418 - loss: 0.3967 - val_accuracy: 0.8444 - val_loss: 0.3741
Epoch 12/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8414 - loss: 0.4011 - val_accuracy: 0.8449 - val_loss: 0.3733
Epoch 13/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8440 - loss: 0.3999 - val_accuracy: 0.8446 - val_loss: 0.3729
Epoch 14/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8427 - loss: 0.3977 - val_accuracy: 0.8448 - val_loss: 0.3723
Epoch 15/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8434 - loss: 0.3988 - val_accuracy: 0.8450 - val_loss: 0.3717
Epoch 16/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8419 - loss: 0.3972 - val_accuracy: 0.8449 - val_loss: 0.3710
Epoch 17/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8438 - loss: 0.3943 - val_accuracy: 0.8446 - val_loss: 0.3706
Epoch 18/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8442 - loss: 0.3950 - val_accuracy: 0.8447 - val_loss: 0.3702
Epoch 19/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8407 - loss: 0.3968 - val_accuracy: 0.8446 - val_loss: 0.3700
Epoch 20/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8447 - loss: 0.3945 - val_accuracy: 0.8450 - val_loss: 0.3695
Epoch 21/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8430 - loss: 0.3942 - val_accuracy: 0.8453 - val_loss: 0.3692
Epoch 22/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8434 - loss: 0.3931 - val_accuracy: 0.8455 - val_loss: 0.3690
Epoch 23/150
```

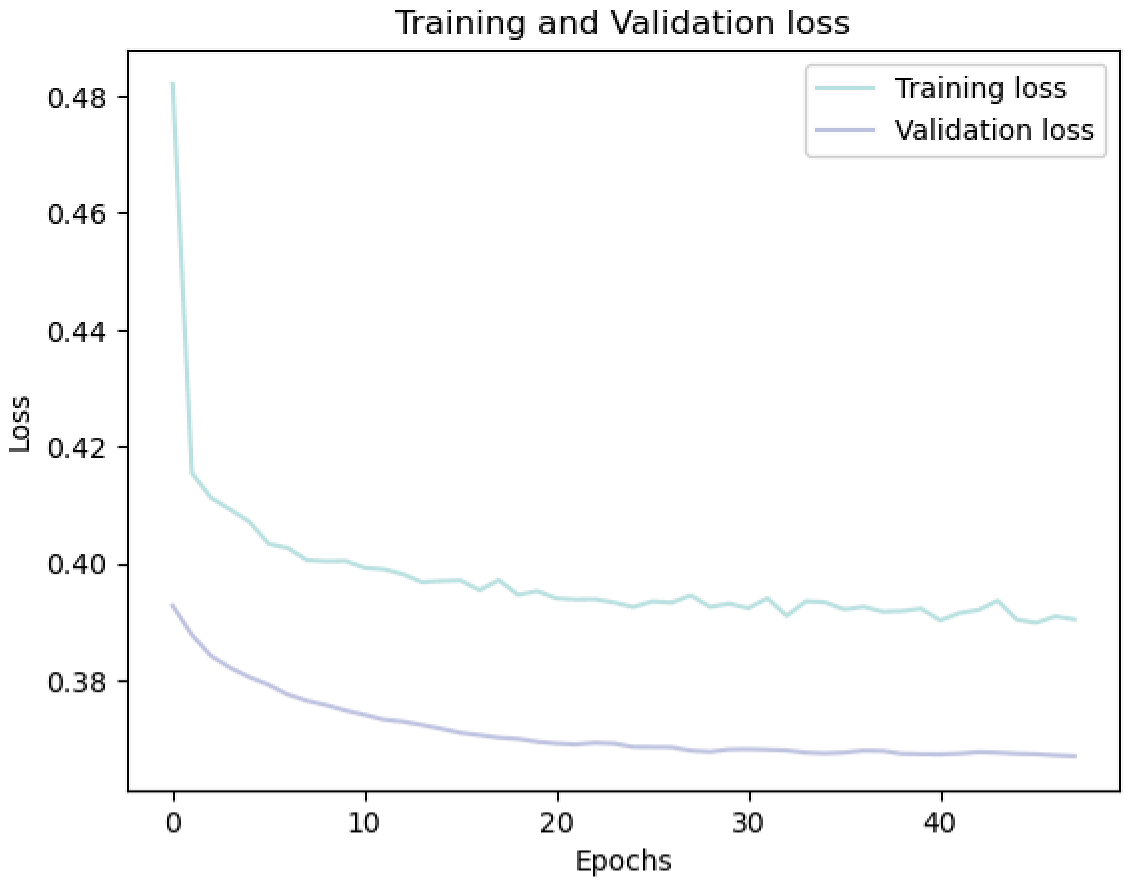
```
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8438 - loss: 0.3924 - val_accuracy: 0.8449 - val_loss: 0.3693
Epoch 24/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8414 - loss: 0.3965 - val_accuracy: 0.8446 - val_loss: 0.3692
Epoch 25/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8417 - loss: 0.3952 - val_accuracy: 0.8450 - val_loss: 0.3686
Epoch 26/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8457 - loss: 0.3892 - val_accuracy: 0.8442 - val_loss: 0.3686
Epoch 27/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8435 - loss: 0.3957 - val_accuracy: 0.8443 - val_loss: 0.3685
Epoch 28/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8447 - loss: 0.3944 - val_accuracy: 0.8445 - val_loss: 0.3680
Epoch 29/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8460 - loss: 0.3889 - val_accuracy: 0.8441 - val_loss: 0.3678
Epoch 30/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8452 - loss: 0.3914 - val_accuracy: 0.8447 - val_loss: 0.3681
Epoch 31/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8435 - loss: 0.3939 - val_accuracy: 0.8441 - val_loss: 0.3682
Epoch 32/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8445 - loss: 0.3951 - val_accuracy: 0.8446 - val_loss: 0.3681
Epoch 33/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8428 - loss: 0.3926 - val_accuracy: 0.8449 - val_loss: 0.3680
Epoch 34/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8397 - loss: 0.3984 - val_accuracy: 0.8447 - val_loss: 0.3676
Epoch 35/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8428 - loss: 0.3949 - val_accuracy: 0.8447 - val_loss: 0.3675
Epoch 36/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8435 - loss: 0.3955 - val_accuracy: 0.8450 - val_loss: 0.3676
Epoch 37/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8445 - loss: 0.3932 - val_accuracy: 0.8440 - val_loss: 0.3680
Epoch 38/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8432 - loss: 0.3927 - val_accuracy: 0.8445 - val_loss: 0.3679
Epoch 39/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8466 - loss: 0.3874 - val_accuracy: 0.8443 - val_loss: 0.3674
Epoch 40/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 6s 2ms/step - accuracy: 0.8438 - loss: 0.3916 - val_accuracy: 0.8440 - val_loss: 0.3674
Epoch 41/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8441 - loss: 0.3900 - val_accuracy: 0.8446 - val_loss: 0.3673
Epoch 42/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8433 - loss: 0.3933 - val_accuracy: 0.8442 - val_loss: 0.3674
Epoch 43/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8432 - loss: 0.3927 - val_accuracy: 0.8443 - val_loss: 0.3677
Epoch 44/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8451 - loss: 0.3937 - val_accuracy: 0.8444 - val_loss: 0.3676
Epoch 45/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8444 - loss: 0.3877 - val_accuracy: 0.8444 - val_loss: 0.3674
```

```
Epoch 46/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8449 - loss: 0.3880 - val_accuracy: 0.8445 - val_loss: 0.3674
Epoch 47/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8442 - loss: 0.3926 - val_accuracy: 0.8442 - val_loss: 0.3671
Epoch 48/150
2551/2551 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8449 - loss: 0.3903 - val_accuracy: 0.8449 - val_loss: 0.3670
```

Plotting training and validation loss over epochs

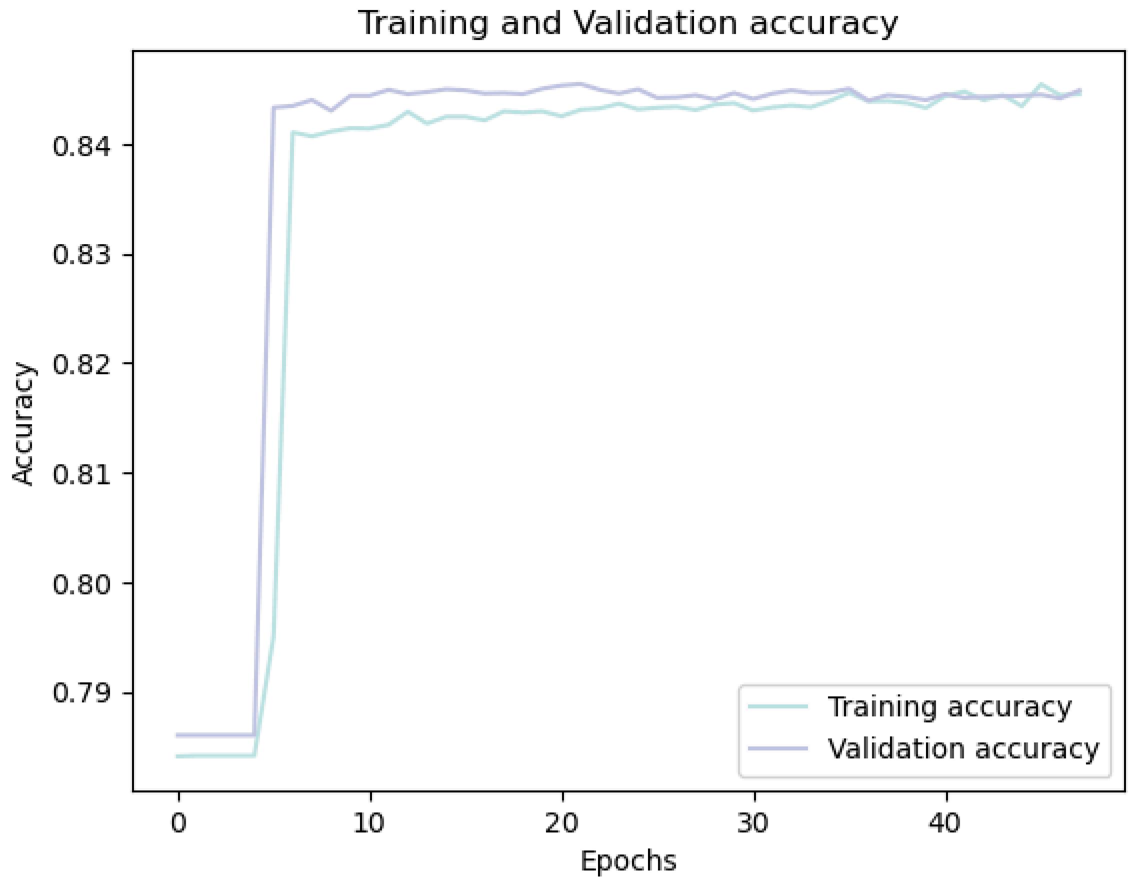
In [27]:

```python
history_df = pd.DataFrame(history.history)

plt.plot(history_df.loc[:, ['loss']], "#BDE2E2", label='Training loss')
plt.plot(history_df.loc[:, ['val_loss']],"#C2C4E2", label='Validation loss')
plt.title('Training and Validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend(loc="best")

plt.show()
```



Plotting training and validation accuracy over epochs

In [28]:
```
history_df = pd.DataFrame(history.history)

plt.plot(history_df.loc[:, ['accuracy']], "#BDE2E2", label='Training accuracy')
plt.plot(history_df.loc[:, ['val_accuracy']], "#C2C4E2", label='Validation accuracy')

plt.title('Training and Validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```



CONCLUSION

CONCLUSIONS

Concluding the model with:

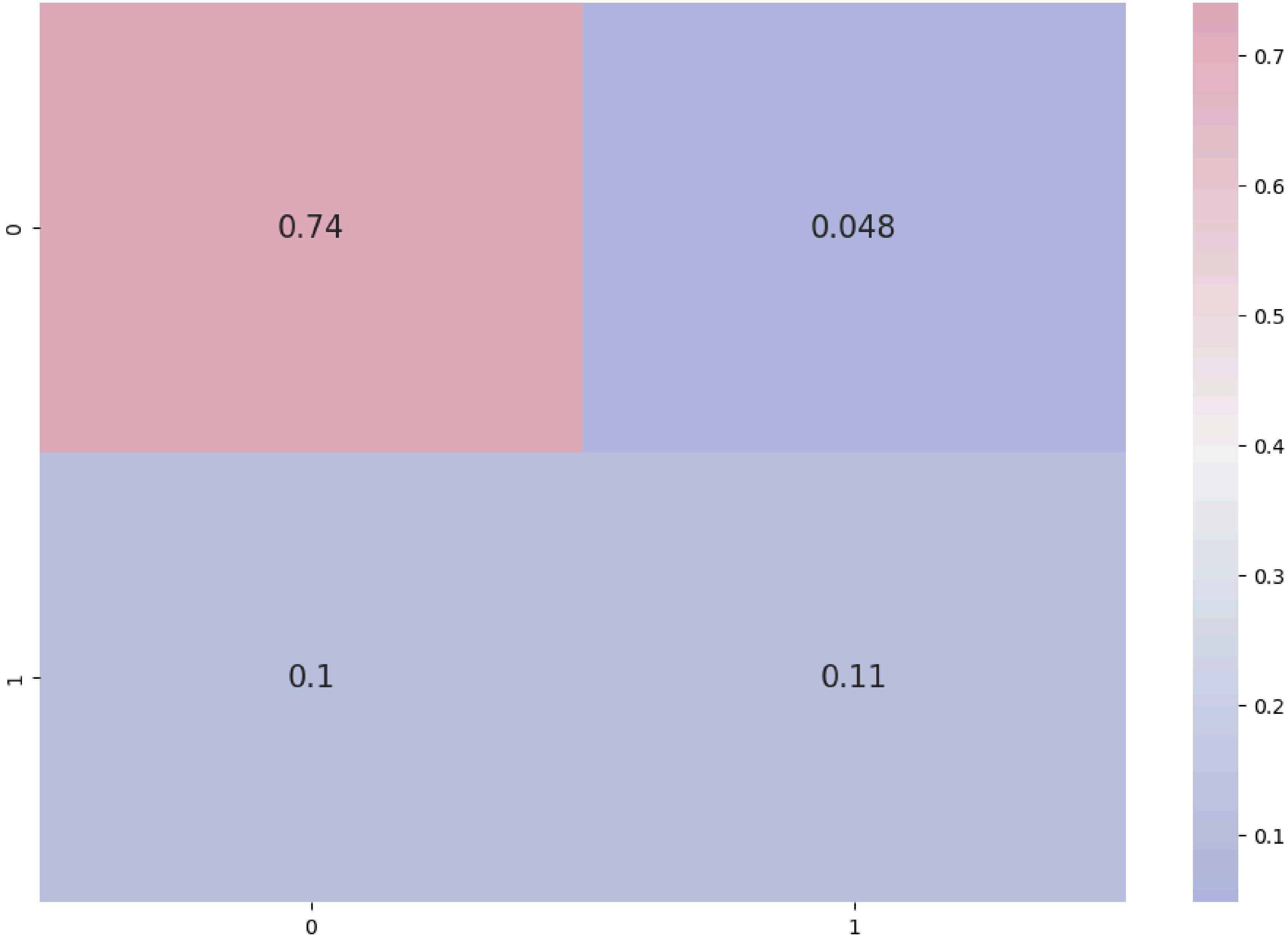
- Testing on the test set
- Evaluating the confusion matrix
- Evaluating the classification report

In [29]:
```
# Predicting the test set results
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)
```

```
798/798 ━━━━━━━━━━━━━━━━━━━━ 1s 1ms/step
```

In [30]:
```
# confusion matrix
cmap1 = sns.diverging_palette(260,-10,s=50, l=75, n=5, as_cmap=True)
plt.subplots(figsize=(12,8))
cf_matrix = confusion_matrix(y_test, y_pred)
sns.heatmap(cf_matrix/np.sum(cf_matrix), cmap = cmap1, annot = True, annot_kws = {'size':15})
```

Out[30]:
```
<AxesSubplot:>
```



In [31]:
```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.88      0.94      0.91     20110
           1       0.69      0.51      0.59      5398

    accuracy                           0.85     25508
   macro avg       0.78      0.72      0.75     25508
weighted avg       0.84      0.85      0.84     25508
```

In [ ]:

In [ ]: