SPOD_Group7_Project

June 16, 2022

1 Weather in Australia - Team 7

This cell just loads all used moduls for running the notebook. Please install any package if you dont have it installed in your environment so far.

[1]:
```
#disable some annoying warnings
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)
#---------------------------#
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib import pyplot
#plots the figures in place instead of a new window
%matplotlib inline

import statistics

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn import decomposition
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import KMeans
from matplotlib import pyplot
from sklearn.cluster import AffinityPropagation
from sklearn.cluster import AgglomerativeClustering
from IPython.display import display, clear_output

from sklearn.ensemble import GradientBoostingClassifier
from sklearn.tree import DecisionTreeClassifier, DecisionTreeRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
from sklearn import tree
from sklearn.model_selection import GridSearchCV
```

```
from sklearn.metrics import accuracy_score, confusion_matrix, recall_score,␣
 ↪precision_score, roc_auc_score, roc_curve,\
    explained_variance_score, mean_squared_error, r2_score, mean_absolute_error
from sklearn import preprocessing
from sklearn.preprocessing import LabelEncoder
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from xgboost import XGBClassifier, XGBRegressor

from abess import LinearRegression
import statsmodels.api as sm
```

2 Dataset Overview

We chose the rain in Australia dataset from Kaggle because we thought that it could be interesting to analyze a dataset with around 145000 rows. It is also interesting that data from about 10 years of daily observations from different locations throughout Australia has been collected.

Besides several numerical attributes, also several categorical attributes are provided. The attributes of the used dataset are explained below.

Date: The observation's date

Location: The location of the observation

MinTemp: The minimum temperature on that day (°C)

MaxTemp: The maximum temperature on that day (°C)

Rainfall: The rainfall amount measured in mm

Evaporation: The evaporation also measured in mm

Sunshine: The number of sunshine hours

WindGustDir: The strongest wind gust's direction

WindGustSpeed: The strongest wind gust's speed in km/h

WindDir9am: The wind's direction at 9 AM

WindDir3pm: The wind's direction at 3 PM

WindSpeed9am: The wind's speed (km/h) at 9 AM

WindSpeed3pm: The wind's speed (km/h) at 3 PM

Humidity9am: The humidity percentage at 9 AM

Humidity3pm: The humidity percentage at 3 PM

Pressure9am: The atmospheric pressure (hpa) at 9 AM

Pressure3pm: The atmospheric pressure (hpa) at 3 PM

Cloud9am: Fraction of obscured sky by clouds (in "oktas") at 9 AM

Cloud3pm: Same as above but at 3 PM

Temp9am: Temperature in °C at 9 AM

Temp3pm: Temperature in °C at 3 PM

RainToday: True, if it has been raining on that day, otherwise False

RainTomorrow: True, if it has been raining on the next day, otherwise False; target variable

[2]:
```
# use the weather dataset of heterogenous data and plot first 5 lines
weather = pd.read_csv('data/weatherAUS.csv')
weather.head()
```

[2]:
```
         Date Location  MinTemp  MaxTemp  Rainfall  Evaporation  Sunshine  \
0  2008-12-01   Albury     13.4     22.9       0.6          NaN       NaN
1  2008-12-02   Albury      7.4     25.1       0.0          NaN       NaN
2  2008-12-03   Albury     12.9     25.7       0.0          NaN       NaN
3  2008-12-04   Albury      9.2     28.0       0.0          NaN       NaN
4  2008-12-05   Albury     17.5     32.3       1.0          NaN       NaN

  WindGustDir  WindGustSpeed WindDir9am  … Humidity9am  Humidity3pm  \
0           W           44.0          W  …        71.0         22.0
1         WNW           44.0        NNW  …        44.0         25.0
2         WSW           46.0          W  …        38.0         30.0
3          NE           24.0         SE  …        45.0         16.0
4           W           41.0        ENE  …        82.0         33.0

   Pressure9am  Pressure3pm  Cloud9am  Cloud3pm  Temp9am  Temp3pm  RainToday  \
0       1007.7       1007.1       8.0       NaN     16.9     21.8         No
1       1010.6       1007.8       NaN       NaN     17.2     24.3         No
2       1007.6       1008.7       NaN       2.0     21.0     23.2         No
3       1017.6       1012.8       NaN       NaN     18.1     26.5         No
4       1010.8       1006.0       7.0       8.0     17.8     29.7         No

   RainTomorrow
0            No
1            No
2            No
3            No
4            No

[5 rows x 23 columns]
```

[3]:
```
# overview of the created datatypes
weather.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
```

3 Data Preparation - Adjust Date Values

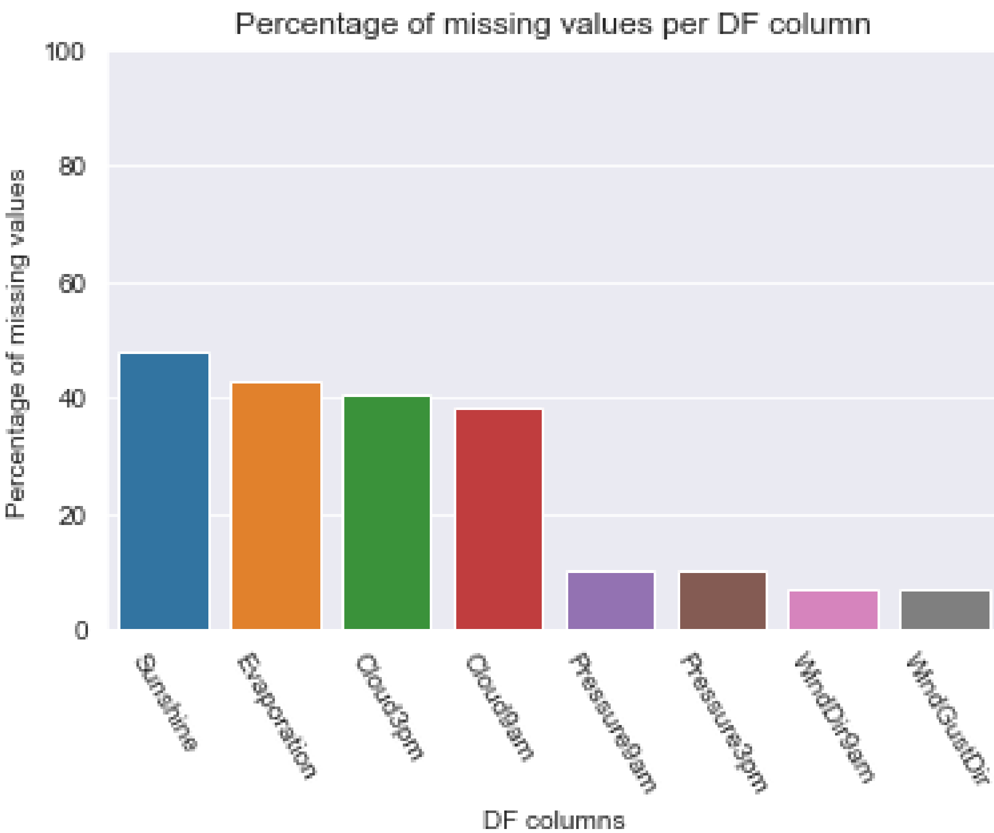
In this step, the data gets adjusted, in order to fit for our analysis. This adjustments go especially for the Date in the first place. Here the whole Date value gets split up into a new year month and day column, in order to better aggregate over the set.

[4]:
```
# Convert Date to a date type and create new columns
weather['Date_converted'] = pd.to_datetime(weather['Date'], format='%Y-%m-%d')
weather['Year'] = weather['Date_converted'].dt.year
weather['Month'] = weather['Date_converted'].dt.month
weather['Day'] = weather['Date_converted'].dt.day
```

4 Overview of missing values

In order to to a proper data cleaning and having a feeling, how many values are even missing, we analysed the amount of missing data per column. It can be seen that for some columns nearly half of the values (40 - 48%) are missing (shown in the table as well as the plot above).

[5]:
```
# Calculate percentage of null values per attribute
missing_in_percentage = weather.isnull().sum() * 100 / len(weather)
missing = pd.DataFrame({'col': weather.columns, 'missing_percent':␣
↪missing_in_percentage})
missing.sort_values('missing_percent', inplace=True, ascending=False)

ax = sns.barplot(x="col", y="missing_percent", data=missing.head(8))
ax.set_ylim((0, 100))
ax.set_xticklabels(ax.get_xticklabels(), rotation=300)
ax.set_title('Percentage of missing values per DF column')
ax.set_xlabel('DF columns')
_ = ax.set_ylabel('Percentage of missing values')
```



5 Base for missing values

5.1 Missing values in different seasons

Now we further investigate this issue by looking at the columns sunshine, evaporation, cloud3pm and cloud9am by grouping the percentage of missing values first by season, to look whether we can see a seasonal affect. We also group the percentage of missing values by location to see if we can spot a locational affect. But as you can also see in the table below, there is no real trend, if the values tend to be not recorded in a specific season.

[6]:
```
# Mapping the dates to seasons and calculate for each season and attribute the␣
↪percentage of missing values.
seasons = {
    1: 'Winter',
    2: 'Spring',
    3: 'Summer',
    4: 'Autumn'
}
df_values_season = weather[['Year', 'Month', 'Sunshine', 'Evaporation',␣
↪'Cloud3pm', 'Cloud9am']].copy()

df_values_season['Season'] = (df_values_season['Month'] % 12 + 3) // 3
df_values_season['Season_name'] = df_values_season['Season'].map(seasons)

df_season_count_null = df_values_season[['Sunshine', 'Evaporation', 'Cloud3pm',␣
↪'Cloud9am']].isnull().groupby(df_values_season['Season_name']).sum()
df_season_count_all = df_values_season[['Sunshine', 'Evaporation', 'Cloud3pm',␣
↪'Cloud9am']].isnull().groupby(df_values_season['Season_name']).count()

df_missing_values_percent = (df_season_count_null / df_season_count_all) * 100
df_missing_values_percent['Season'] = df_missing_values_percent.index.tolist()
df_missing_values_percent.style.hide_index()
```

[6]: <pandas.io.formats.style.Styler at 0x27c403ca470>

5.2 Missing values in different locations

As it can be seen, for 22 of the 49 locations no values are tracked which explains the large amount of missing data for the attributes ‘Sunshine’, ‘Evaporation’, ‘Cloud3pm’ and ‘Cloud9am’. The reason for this is, however, unknown.

[7]:
```
df_values_location = weather[['Location', 'Sunshine', 'Evaporation',␣
↪'Cloud3pm', 'Cloud9am']]
df_values_location_count_null = weather[['Sunshine', 'Evaporation', 'Cloud3pm',␣
↪'Cloud9am']].isnull().groupby(weather['Location']).sum()
# fillna is needed in order to get the
df_values_location_count_all = weather[['Sunshine', 'Evaporation', 'Cloud3pm',␣
↪'Cloud9am']].isnull().groupby(weather['Location']).count()

df_missing_values_percent = (df_values_location_count_null /␣
↪df_values_location_count_all) * 100
df_missing_values_percent['Location'] = df_missing_values_percent.index.tolist()
mask = (df_missing_values_percent == 100.).any(axis=1)
```

```
print(f'Untracked values based on location: {df_missing_values_percent[mask].
↪shape[0]} of {df_missing_values_percent.shape[0]}')
```

```
Untracked values based on location: 22 of 49
```

6 Remove missing values

Since we can not clearly 'clean' missing values in any case, because we dont have information about the geo coordinates and also no mapping of close location, we simply drop these values. Still - 112925 samples are present

[8]:
```
weather.drop(['Date','Sunshine', 'Evaporation', 'Cloud3pm',␣
↪'Cloud9am'],axis=1,inplace=True)
```

6.1 Create artifical data for missing values in numeric attribute vectors when possible

For numeric data we set missing values for numeric attributes (given in the numerical_columns value) to the median based on the year, month and (location) when possible

For the categorical values we used the mode, imputation is based on location and current month, if we do not have data for a location than only the month was used.

[9]:
```
numerical_columns = ["Pressure9am", "Pressure3pm", "Humidity3pm",␣
↪"Humidity9am", "WindGustSpeed", "Temp3pm",
                     "WindSpeed3pm", "WindSpeed9am", "Temp9am", "MinTemp",␣
↪"MaxTemp", "Rainfall"]

for col in numerical_columns:
    weather[col] = weather[col].fillna(weather.groupby(['Year', 'Month',␣
↪'Location'])[col].transform("mean"))
    weather[col] = weather[col].fillna(weather.groupby(['Year', 'Month'])[col].
↪transform("mean"))

categorical_columns = ["WindDir9am", "WindGustDir", "WindDir3pm"]

for col in categorical_columns:
    weather[col] = weather[col].fillna(weather.groupby(['Year', 'Month',␣
↪'Location'])[col].transform(statistics.mode))
    weather[col] = weather[col].fillna(weather.groupby(['Year', 'Month'])[col].
↪transform(statistics.mode))
```

[10]:
```
weather.dropna(inplace=True)
print(f'Amount of samples without missing values in any column: {weather.
↪shape[0]}')
weather.head()
```

```
Amount of samples without missing values in any column: 140787
```

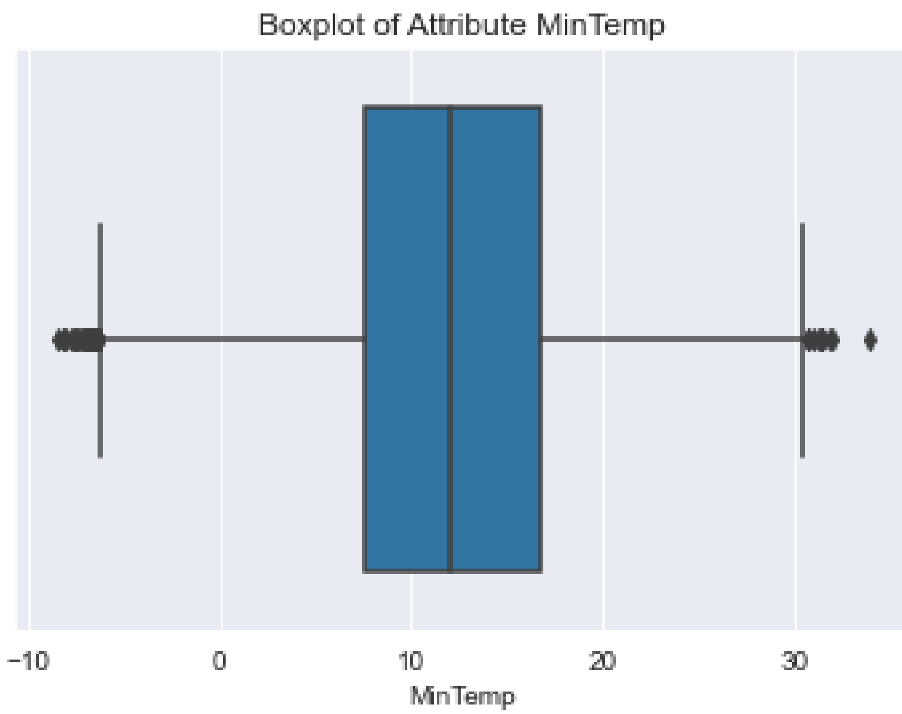
[10]:
```
  Location  MinTemp  MaxTemp  Rainfall WindGustDir  WindGustSpeed WindDir9am  \
0   Albury     13.4     22.9       0.6           W           44.0          W
1   Albury      7.4     25.1       0.0         WNW           44.0        NNW
2   Albury     12.9     25.7       0.0         WSW           46.0          W
3   Albury      9.2     28.0       0.0          NE           24.0         SE
4   Albury     17.5     32.3       1.0           W           41.0        ENE

  WindDir3pm  WindSpeed9am  WindSpeed3pm  …  Pressure9am  Pressure3pm  \
0        WNW          20.0          24.0  …       1007.7       1007.1
1        WSW           4.0          22.0  …       1010.6       1007.8
2        WSW          19.0          26.0  …       1007.6       1008.7
3          E          11.0           9.0  …       1017.6       1012.8
4         NW           7.0          20.0  …       1010.8       1006.0

   Temp9am  Temp3pm  RainToday  RainTomorrow Date_converted  Year Month  Day
0     16.9     21.8         No            No     2008-12-01  2008    12    1
1     17.2     24.3         No            No     2008-12-02  2008    12    2
2     21.0     23.2         No            No     2008-12-03  2008    12    3
3     18.1     26.5         No            No     2008-12-04  2008    12    4
4     17.8     29.7         No            No     2008-12-05  2008    12    5

[5 rows x 22 columns]
```

7 Check for valid values in all remaining (numeric) columns

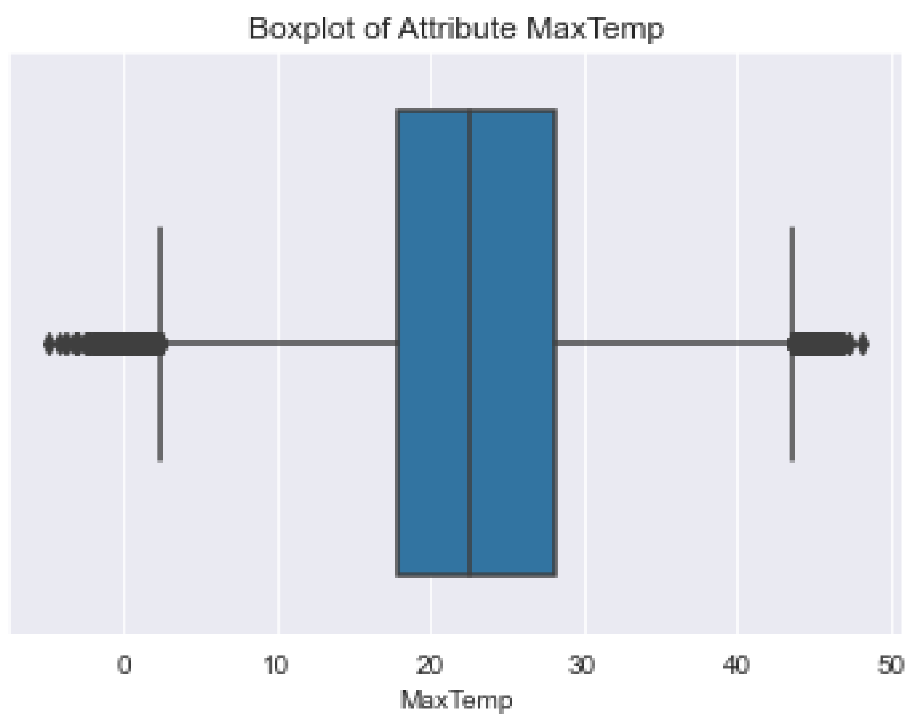
In the next step, # check for minimum and maximum values in numeric attributes (in our case all attributes in the frame which have the datatype of float64. Here no out of range values could be detected.

[11]:
```python
# check for minimum and maximum values in numeric attributes:
for col in weather.loc[:, weather.dtypes == 'float64']:
    print(f'Attribute {col}:')
    print("Min: {:.2f}, Q1: {:.2f}, Median {:.2f}, Q3: {:.2f}, Max: {:.2f}".
 ↪format(weather[col].min(),weather[col].quantile(.25),weather[col].
 ↪median(),weather[col].quantile(.75), weather[col].max()))
    sns.boxplot(x=weather[col])
    plt.title(f'Boxplot of Attribute {col}')
    plt.show()
```

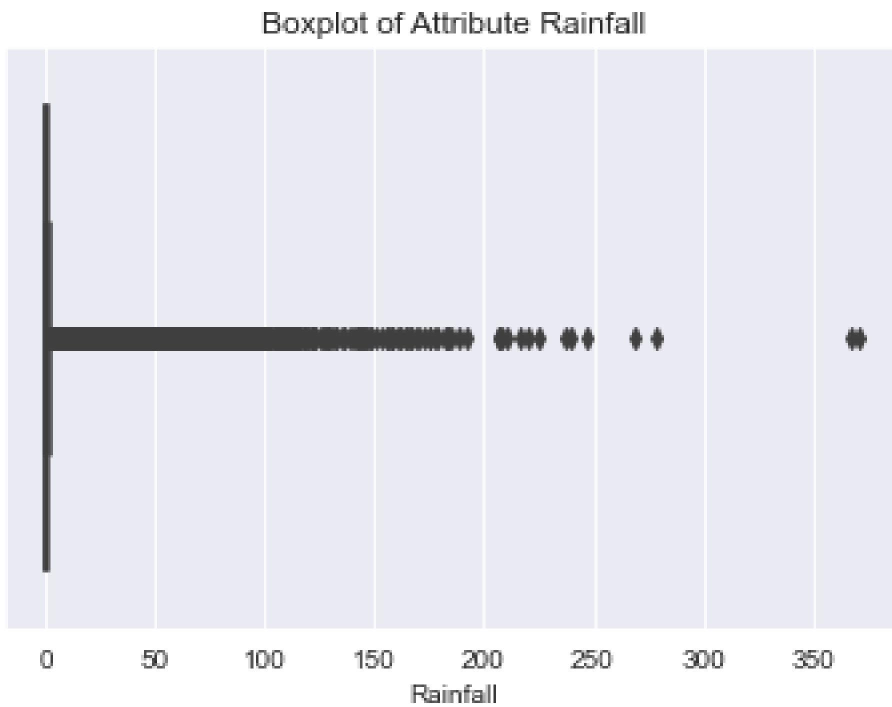
```
Attribute MinTemp:
Min: -8.50, Q1: 7.60, Median 12.00, Q3: 16.80, Max: 33.90
```



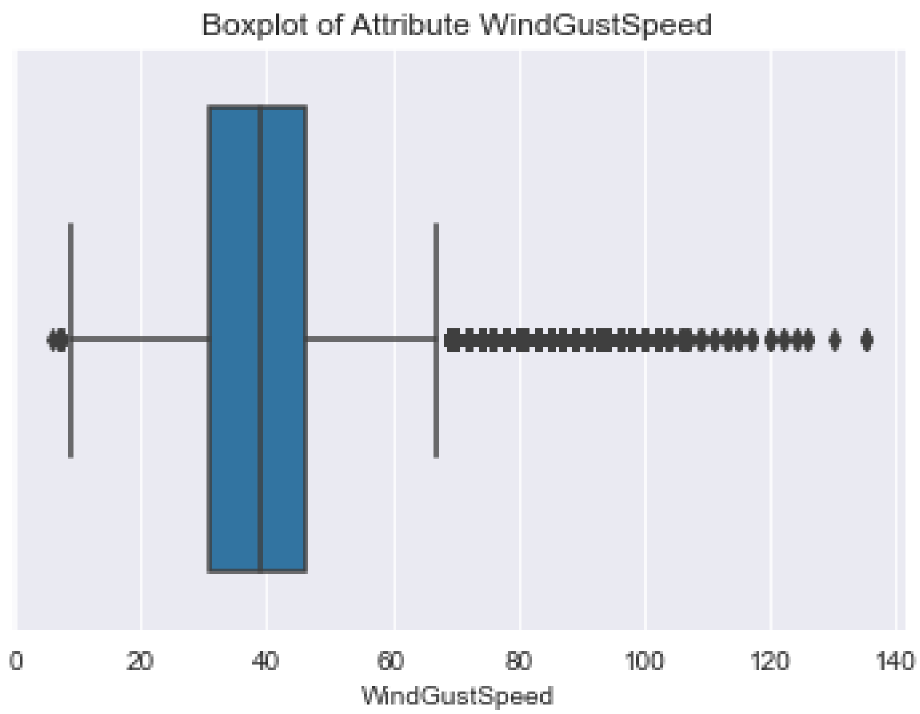
Attribute MaxTemp:
Min: -4.80, Q1: 17.90, Median 22.60, Q3: 28.20, Max: 48.10



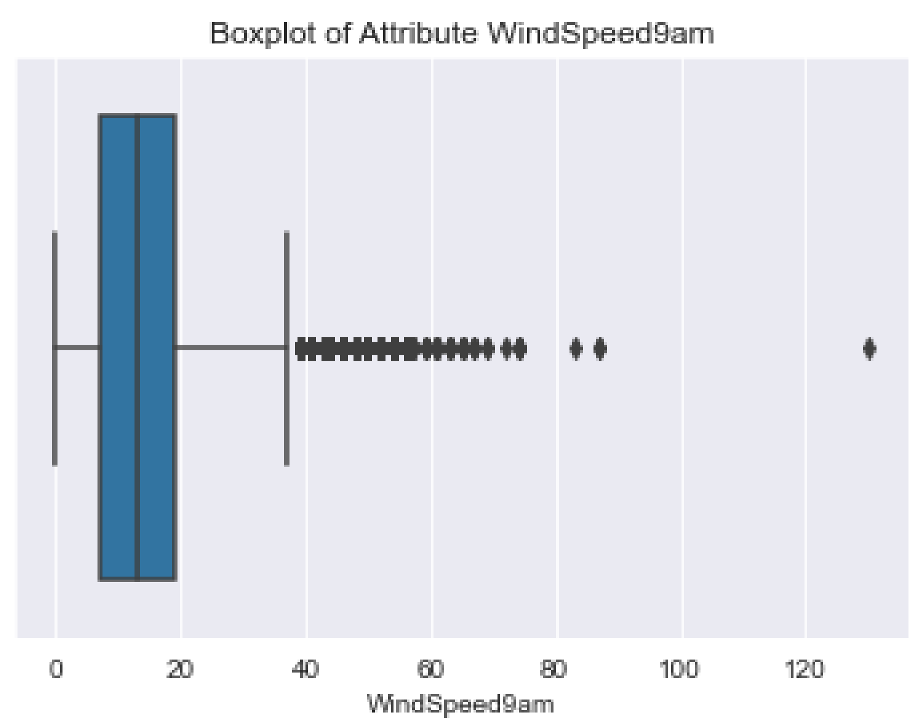
```
Attribute Rainfall:
Min: 0.00, Q1: 0.00, Median 0.00, Q3: 0.80, Max: 371.00
```



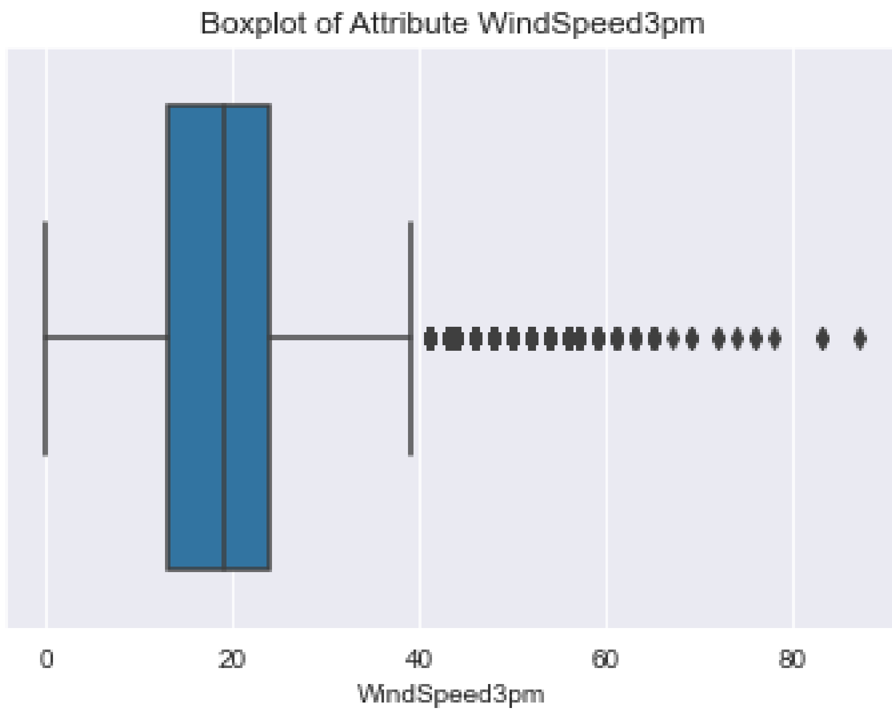

```
Attribute WindGustSpeed:
Min: 6.00, Q1: 31.00, Median 39.00, Q3: 46.00, Max: 135.00
```



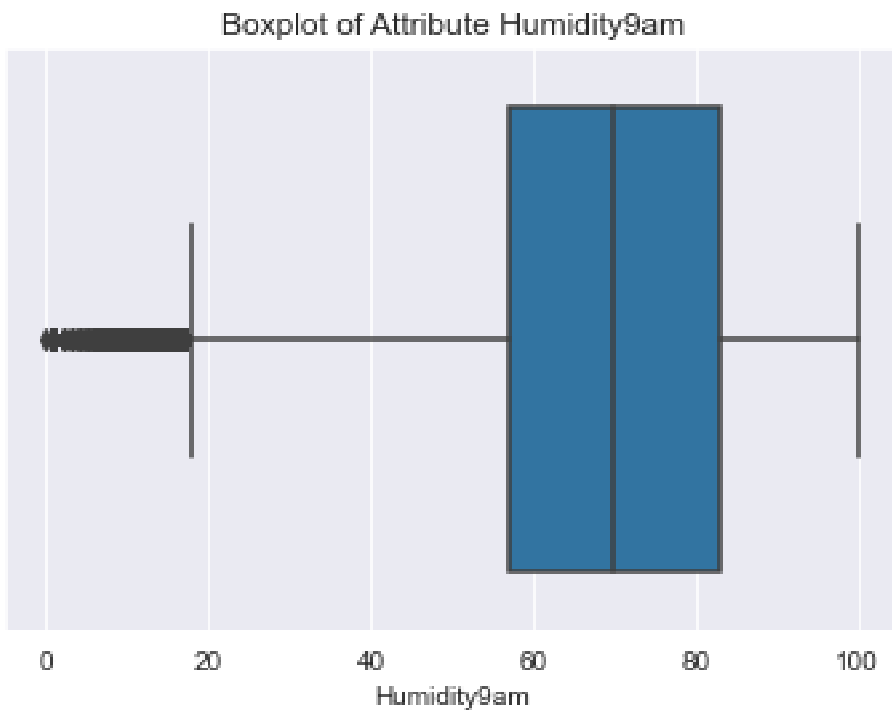

```
Attribute WindSpeed9am:
Min: 0.00, Q1: 7.00, Median 13.00, Q3: 19.00, Max: 130.00
```



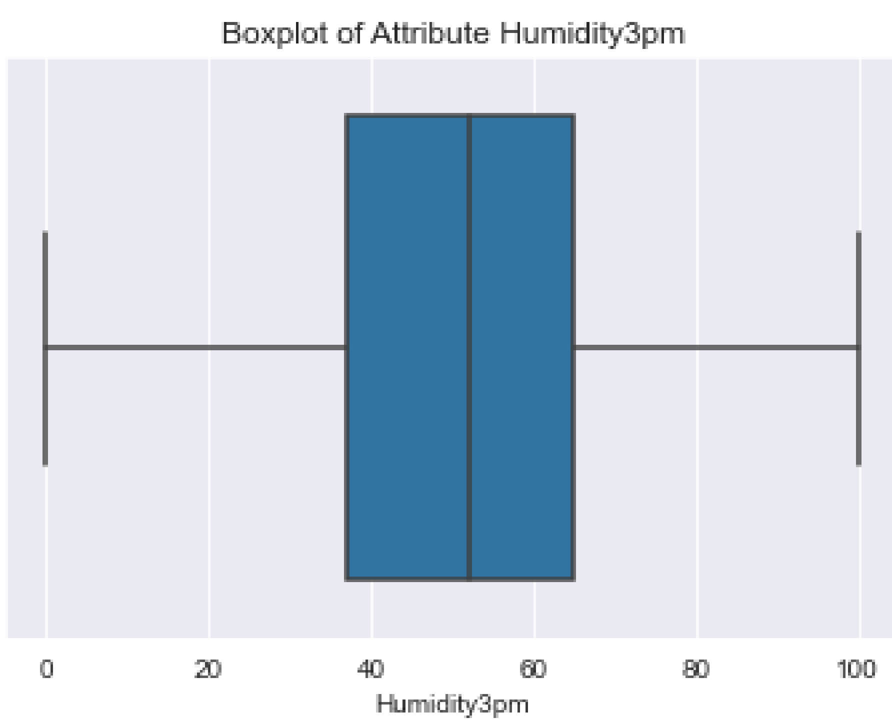
```
Attribute WindSpeed3pm:
Min: 0.00, Q1: 13.00, Median 19.00, Q3: 24.00, Max: 87.00
```



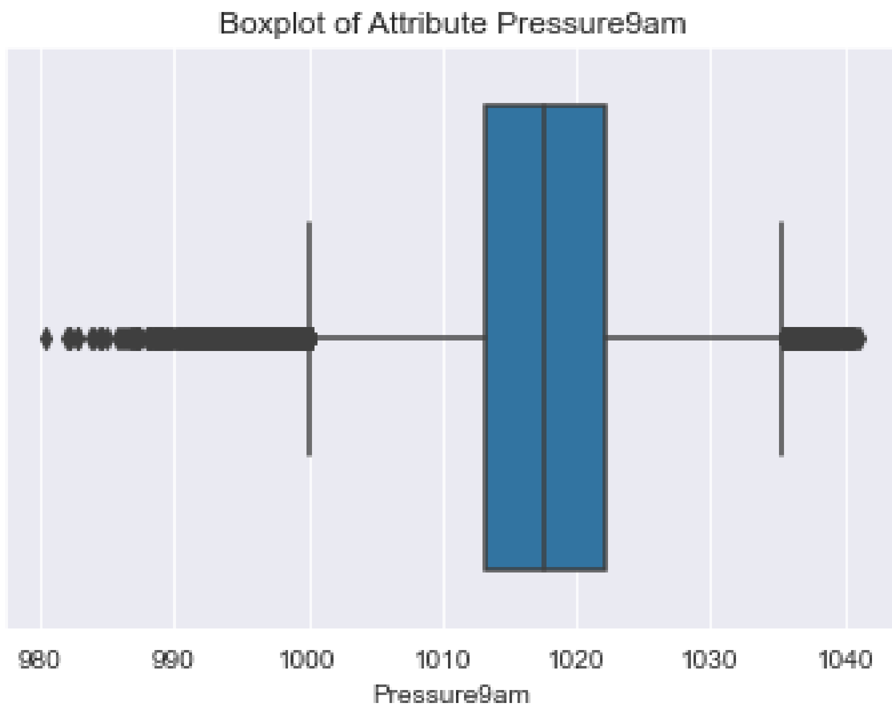

```
Attribute Humidity9am:
Min: 0.00, Q1: 57.00, Median 70.00, Q3: 83.00, Max: 100.00
```



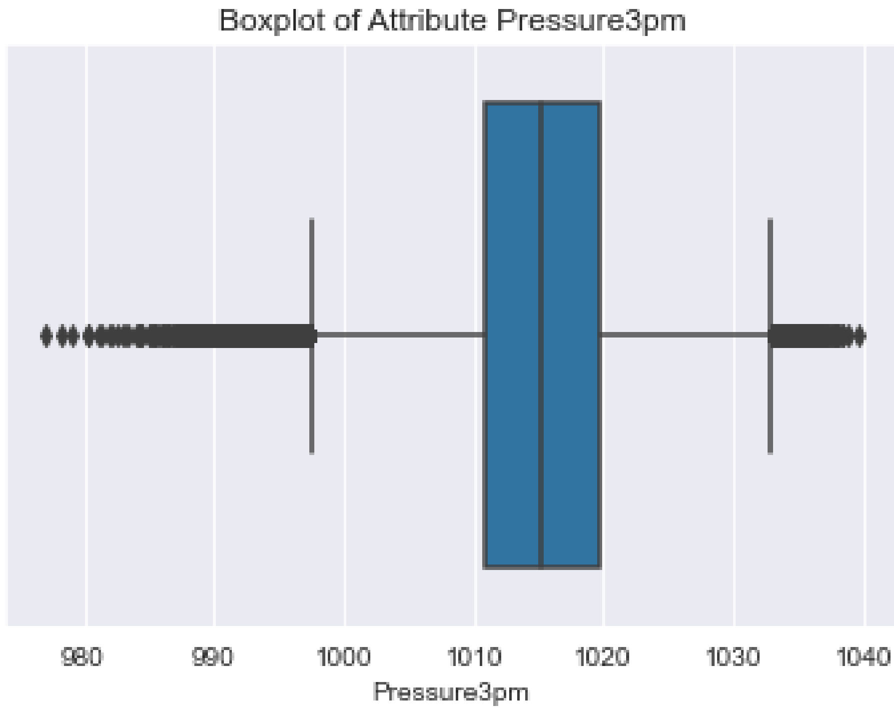

```
Attribute Humidity3pm:
Min: 0.00, Q1: 37.00, Median 52.00, Q3: 65.00, Max: 100.00
```



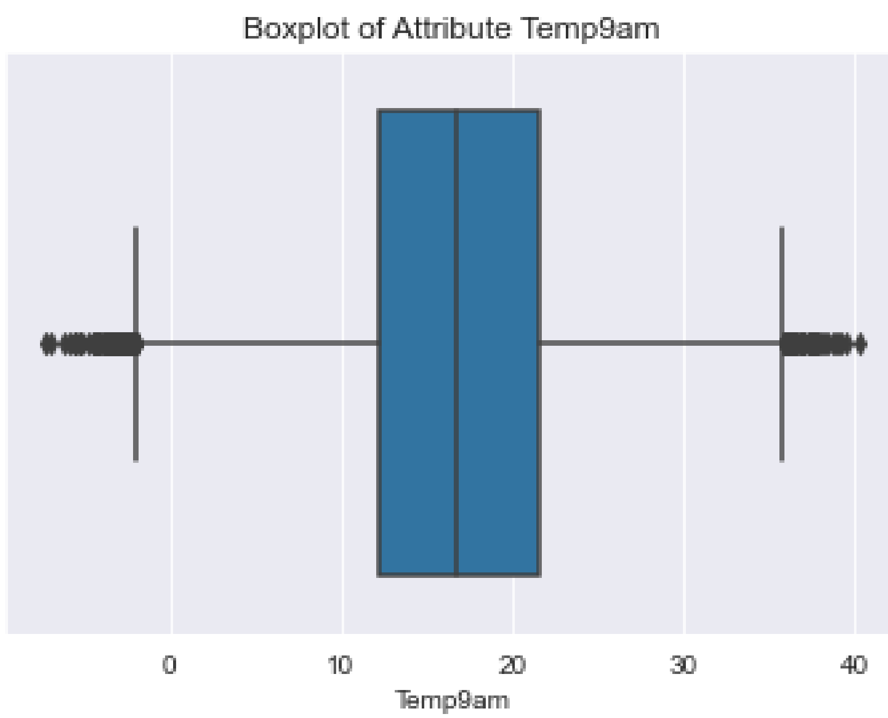
```
Attribute Pressure9am:
Min: 980.50, Q1: 1013.30, Median 1017.60, Q3: 1022.10, Max: 1041.00
```



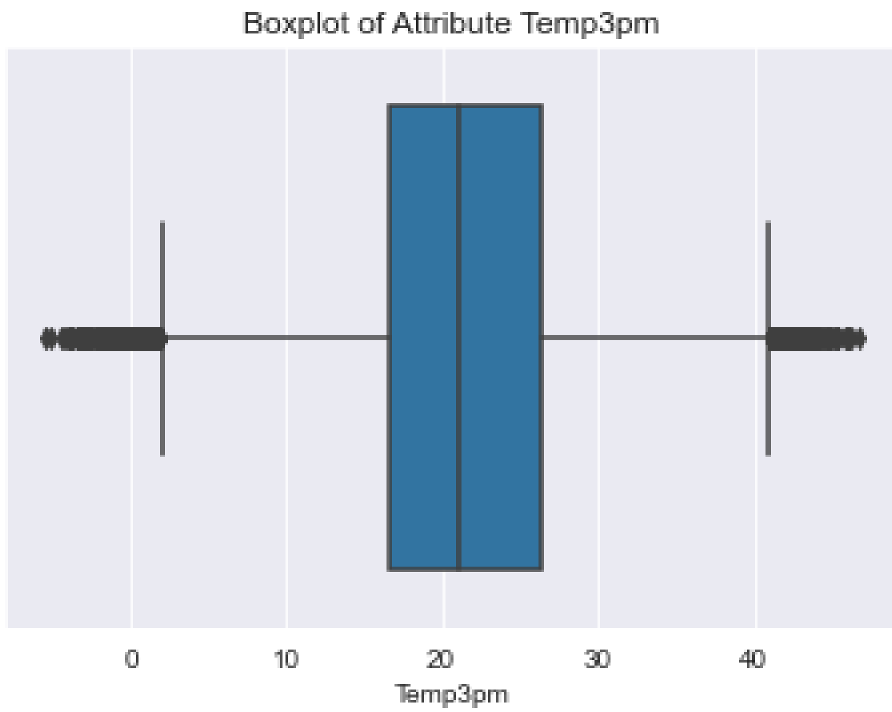

```
Attribute Pressure3pm:
Min: 977.10, Q1: 1010.80, Median 1015.20, Q3: 1019.68, Max: 1039.60
```




```
Attribute Temp9am:
Min: -7.20, Q1: 12.20, Median 16.70, Q3: 21.60, Max: 40.20
```



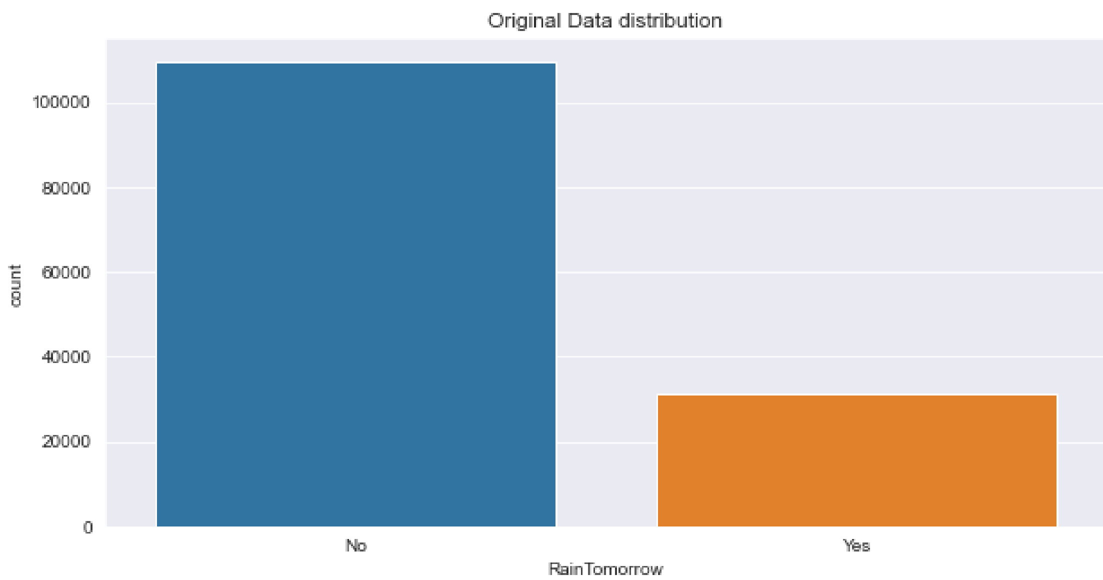
```
Attribute Temp3pm:
Min: -5.40, Q1: 16.60, Median 21.10, Q3: 26.40, Max: 46.70
```



8 Check the distribution of RainTomorrow samples

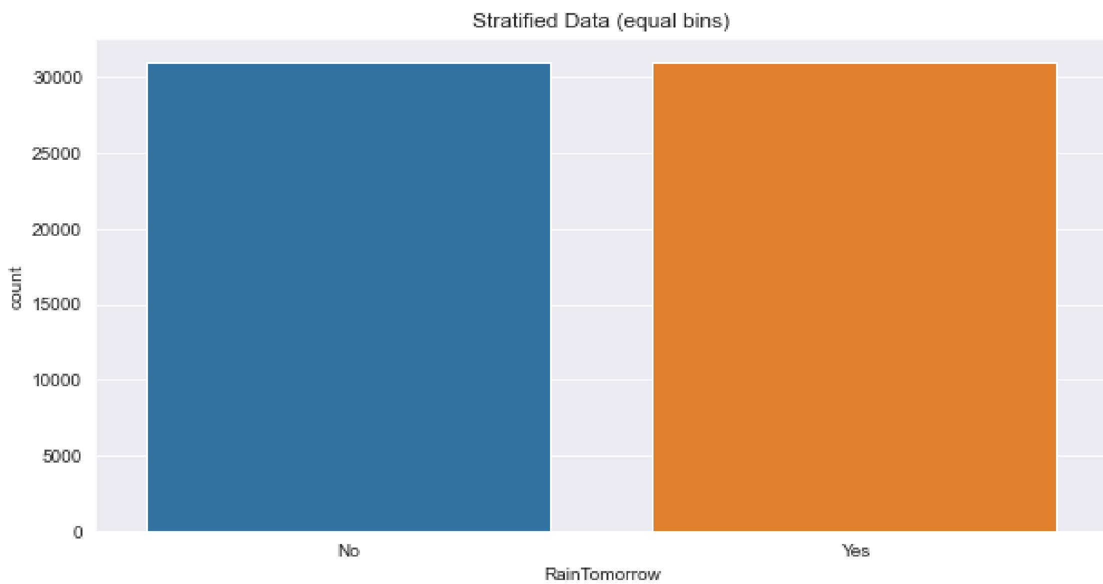
As we can clearly see in the next cell, there are a lot more samples of NOT-raining tomorrow, as samples WITH raining tomorrow

[12]:
```
plt.figure(figsize=(10,5))
sns.countplot(x="RainTomorrow", data=weather);
plt.title('Original Data distribution')
plt.show()
```



[13]:
```
# Disproportionate sampling:
# randomly select 4 samples from each stratum
stratified = weather.groupby('RainTomorrow', group_keys=False).apply(lambda x:␣
↪x.sample(31000))
```

[14]:
```
plt.figure(figsize=(10,5))
sns.countplot(x="RainTomorrow", data=stratified)
plt.title("Stratified Data (equal bins)")
plt.show()
```



9 PCA to explore the underlying structure of the data

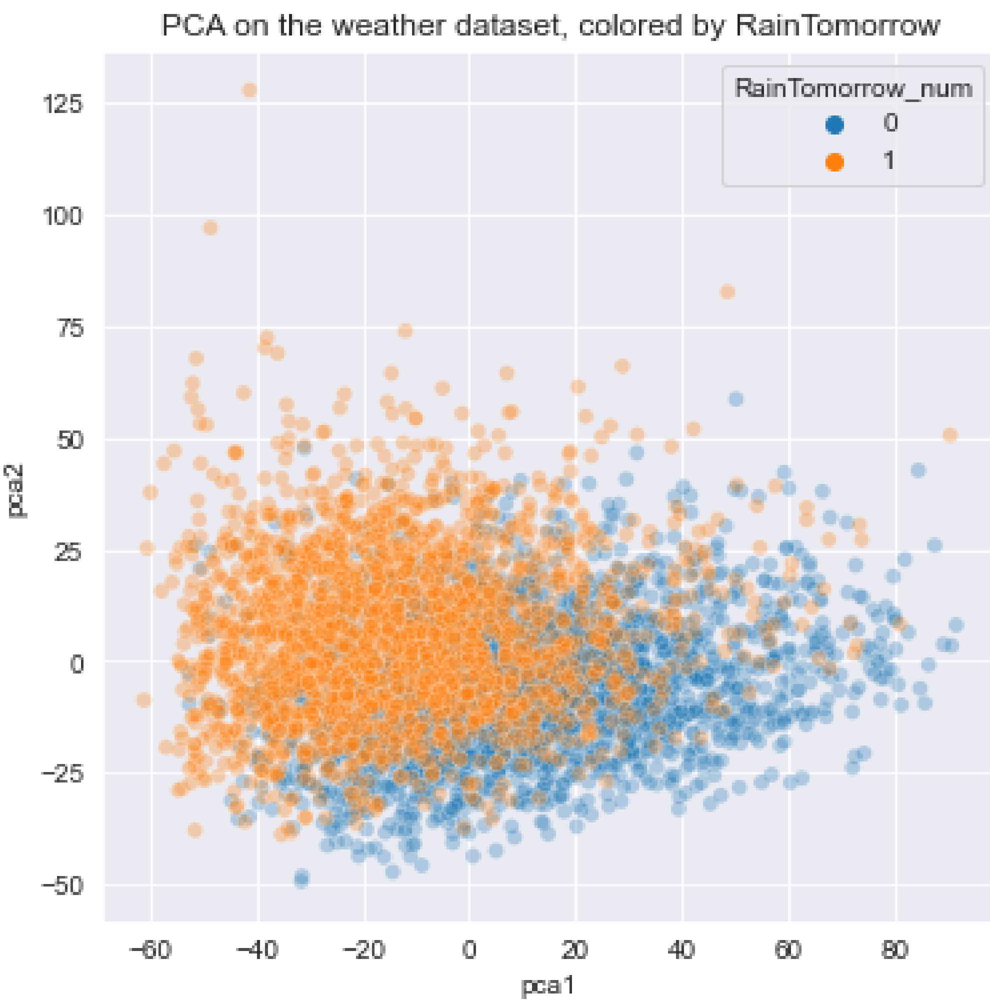
[15]:
```
stratified.drop('Date_converted',axis=1,inplace=True)
for col in stratified.loc[:, stratified.dtypes == object]:
    # creating instance of labelencoder
    labelencoder = LabelEncoder()
    # Assigning numerical values and storing in another column
    stratified[f'{col}_num'] = labelencoder.fit_transform(stratified[col])
    # drop non-numeric column
    stratified.drop(col,axis=1,inplace=True)
```

[16]:
```
stratified.head()
```

[16]:
```
        MinTemp  MaxTemp  Rainfall  WindGustSpeed  WindSpeed9am  WindSpeed3pm  \
109137      8.2     20.7       0.0      34.525486           7.0           4.0
103414      8.4     10.1       0.0      48.000000          15.0          13.0
5304       10.3     17.6       0.0      19.000000           6.0           0.0
81365      12.7     19.7       0.0      39.000000           9.0          13.0
118358     19.3     36.3       0.0      52.000000          19.0          28.0

        Humidity9am  Humidity3pm  Pressure9am  Pressure3pm  …  Temp3pm  \
109137         87.0         54.0       1025.2       1022.1  …     18.8
103414         77.0         61.0       1031.7       1030.2  …      9.7
5304           89.0         64.0       1034.9       1031.6  …     16.9
81365          61.0         52.0       1021.2       1020.6  …     18.4
118358         36.0         31.0       1011.1       1007.5  …     34.2

        Year  Month  Day  Location_num  WindGustDir_num  WindDir9am_num  \
109137  2010      6   20             1               13               7
103414  2011      6   11            28                9               9
5304    2015      6   12             4               14              12
81365   2010      1    3            12               12              15
118358  2010     12   31            32                2              10

        WindDir3pm_num  RainToday_num  RainTomorrow_num
109137              12              0                 0
103414               9              0                 0
5304                 4              0                 0
81365                8              0                 0
118358              15              0                 0

[5 rows x 21 columns]
```

[17]:
```
n_components = 7

pca = decomposition.PCA(n_components=n_components)
pca_pos = pca.fit_transform(stratified)

stratified['pca1']= pca_pos[:, 0]
stratified['pca2']= pca_pos[:, 1]
```

[18]:
```
plt.figure(figsize=(6,6))
reducedPoints = stratified.groupby('RainTomorrow_num', group_keys=False).
 ↪apply(lambda x: x.sample(2500))
sns.scatterplot(data=reducedPoints, x="pca1", y="pca2",␣
 ↪hue="RainTomorrow_num",alpha=0.3)
plt.title('PCA on the weather dataset, colored by RainTomorrow')
plt.show()
```



10 Decision Tree

In this section, we try to fit a Decision Tree classifier to our data. Therefore we do a GridSearch, where we try different criterions, maximum depths of the tree and splitting methods. The trained classifier also gets evaluated on 15% of the total data afterwards.

To keep the dataset clean, we removed all additional added attributes, we used in the previous section due to have more comfort. This does not change the actual data at all.

Note, that the data is also stratified like in the PCA above, so all classes are evenly distributed (standard would be to have a much higher amount of samples in the RainTomorrow=No comapred to RainTomorrow=Yes)

After creating the training and test sets, training and evaluating using a confusion matrix and accuracy as a score, we also provided an overview of the feature importance learned by the decision tree.

[19]:
```
"""
Evaluates the model and returns accuracy as well as a confusion matrix. Also␣
 ↪the time for prediction can is calculated.
@param model, sklearn model,trained model
@param x_test, np ndarray, data matrix
@param y_test, np ndarray, data vector
"""
def get_evaluation(model, x_test, y_test):
    y_pred = model.predict(x_test)
    accuracy = accuracy_score(y_test, y_pred)
    conf_mat = confusion_matrix(y_test, y_pred)
    rec_result = recall_score(y_test, y_pred, average=None, labels=[0,1])
    prec_result = precision_score(y_test, y_pred, average=None, labels=[0,1])


    print('\nAccuracy of Classifier on Test Image Data: ', accuracy)
    print()
    print('Recall (No Rain Tomorrow) of Classifier on Test Image Data: ',␣
 ↪rec_result[0])
    print('Recall (Rain Tomorrow) of Classifier on Test Image Data: ',␣
 ↪rec_result[1])
    print()
    print('Precision (No Rain Tomorrow) of Classifier on Test Image Data: ',␣
 ↪prec_result[0])
    print('Precision (Rain Tomorrow) of Classifier on Test Image Data: ',␣
 ↪prec_result[1])
    print()
    print('\nConfusion Matrix: \n', conf_mat)

    plt.matshow(conf_mat)
    plt.title('Confusion Matrix')
    plt.colorbar()
```

```
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    return None
```

[20]:
```
def get_ROC(model, x_test, y_test):
    """
    Calculates AUC score and plots ROC curve
    @param model, sklearn model,trained model
    @param x_test, np ndarray, data matrix
    @param y_test, np ndarray, data vector
    """
    predictions = model.predict_proba(x_test)

    print('AUC score:')
    print(roc_auc_score(y_test, predictions[:,1]))

    fpr, tpr, _ = roc_curve(y_test, predictions[:,1])

    plt.clf()
    plt.plot(fpr, tpr)
    plt.xlabel('FPR')
    plt.ylabel('TPR')
    plt.title('ROC curve with AUC: {:.3f}'.format(roc_auc_score(y_test,␣
 ↪predictions[:,1])))
    plt.show()
```

[21]:
```
param_grid = {
    'criterion': ['gini','entropy'],
    'max_depth': range(1,20),
    'splitter': ['random', 'best']
}

"""
Trains a decision tree using cross-validation and returns certain attributes of␣
 ↪the received model including the best
parameter combination.
@param x_train, np ndarray, data matrix
@param y_train, np ndarray, data vector
@param param_grid, dict, grid holding the paramaters for search
"""
def train_dec_tree(x_train,y_train,param_grid):
    tree = DecisionTreeClassifier(random_state=55)
    model = GridSearchCV(tree,param_grid=param_grid,n_jobs = -1)
    model.fit(x_train,y_train)
    return model.best_params_,model.best_estimator_
```

[22]:
```python
# remove target value and addtional added columns
X = stratified.drop(['RainTomorrow_num','pca1','pca2'], axis=1)
y = stratified['RainTomorrow_num']
print(f'shape of data matrix: {X.shape}')
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=2)
print(f'shape of train matrix: {x_train.shape}')
print(f'shape of test matrix: {x_test.shape}')
X.head()
```

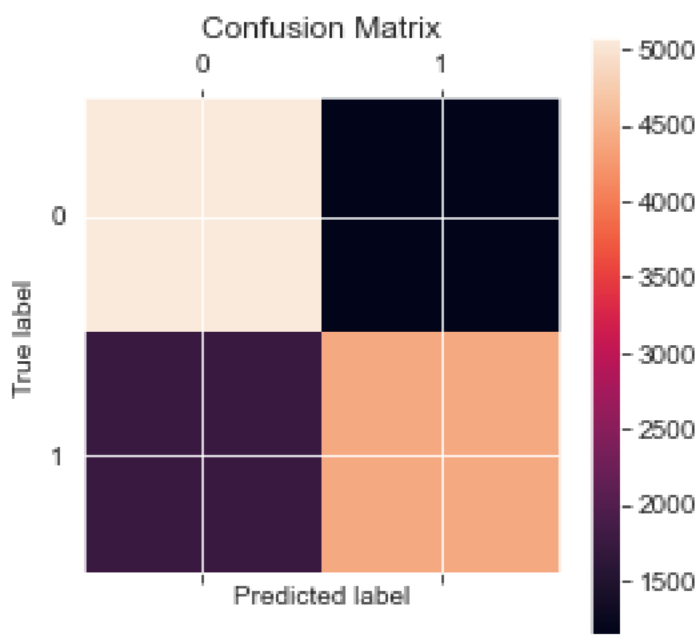
```
shape of data matrix: (62000, 20)
shape of train matrix: (49600, 20)
shape of test matrix: (12400, 20)
```

[22]:
```
        MinTemp  MaxTemp  Rainfall  WindGustSpeed  WindSpeed9am  WindSpeed3pm  \
109137      8.2     20.7       0.0      34.525486           7.0           4.0
103414      8.4     10.1       0.0      48.000000          15.0          13.0
5304       10.3     17.6       0.0      19.000000           6.0           0.0
81365      12.7     19.7       0.0      39.000000           9.0          13.0
118358     19.3     36.3       0.0      52.000000          19.0          28.0

        Humidity9am  Humidity3pm  Pressure9am  Pressure3pm  Temp9am  Temp3pm  \
109137         87.0         54.0       1025.2       1022.1     11.0     18.8
103414         77.0         61.0       1031.7       1030.2      8.8      9.7
5304           89.0         64.0       1034.9       1031.6     12.9     16.9
81365          61.0         52.0       1021.2       1020.6     15.5     18.4
118358         36.0         31.0       1011.1       1007.5     26.6     34.2

        Year  Month  Day  Location_num  WindGustDir_num  WindDir9am_num  \
109137  2010      6   20             1               13               7
103414  2011      6   11            28                9               9
5304    2015      6   12             4               14              12
81365   2010      1    3            12               12              15
118358  2010     12   31            32                2              10

        WindDir3pm_num  RainToday_num
109137              12              0
103414               9              0
5304                 4              0
81365                8              0
118358              15              0
```

[23]:
```python
# train decision tree with created training set and evaluate on created target␣
↪set
params_dec_tree, model_dec_tree = train_dec_tree(x_train, y_train, param_grid)
_ = get_evaluation(model_dec_tree, x_test, y_test)
print("The best parameters are: {}".format(params_dec_tree))
```

```
Accuracy of Classifier on Test Image Data:  0.7659677419354839

Recall (No Rain Tomorrow) of Classifier on Test Image Data:  0.815367340391402
Recall (Rain Tomorrow) of Classifier on Test Image Data:  0.7160233538760947

Precision (No Rain Tomorrow) of Classifier on Test Image Data:
0.7437810945273632
Precision (Rain Tomorrow) of Classifier on Test Image Data:  0.7932087675170679


Confusion Matrix:
 [[5083 1151]
 [1751 4415]]
The best parameters are: {'criterion': 'entropy', 'max_depth': 8, 'splitter':
'best'}
```

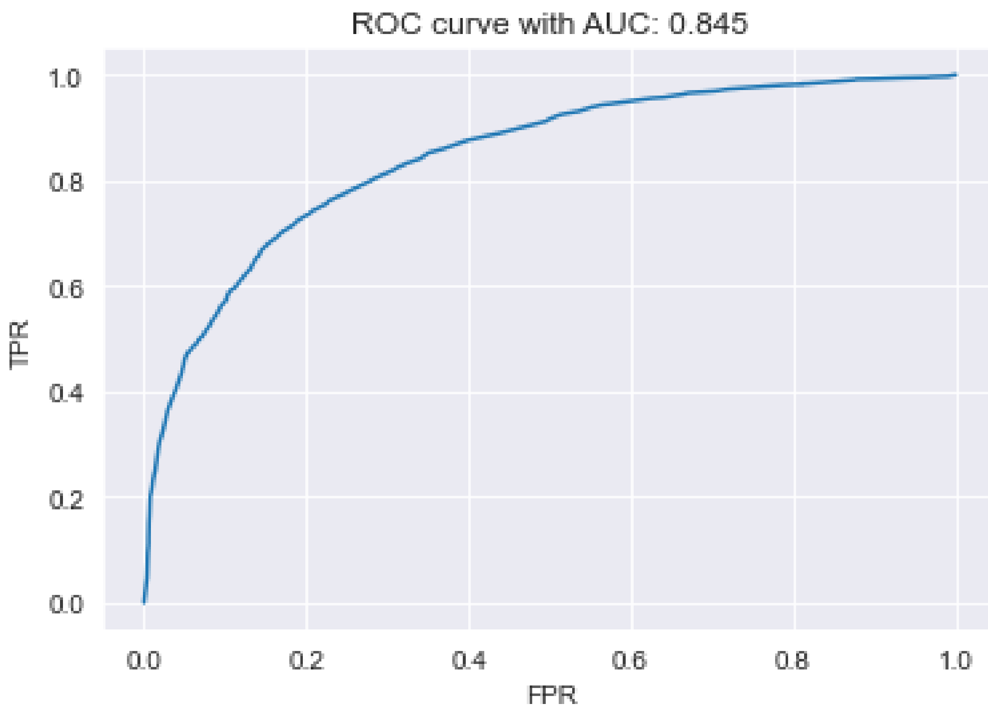
```
C:\Users\fnern\AppData\Local\Temp\ipykernel_19852\1287647186.py:27:
MatplotlibDeprecationWarning: Auto-removal of grids by pcolor() and pcolormesh()
is deprecated since 3.5 and will be removed two minor releases later; please
call grid(False) first.
  plt.colorbar()
```



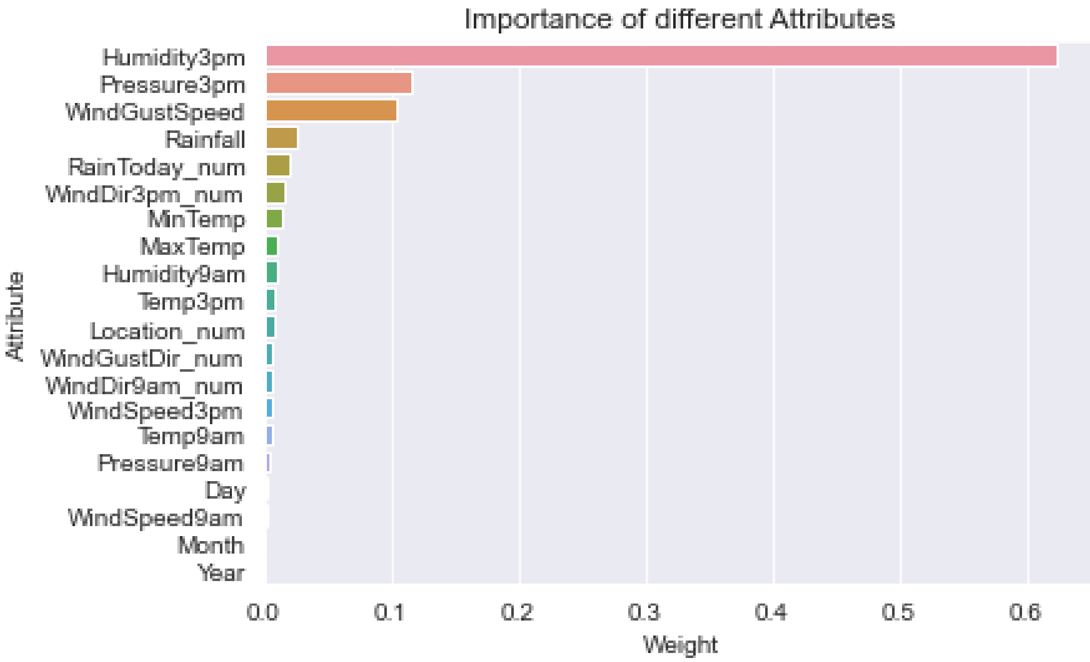
[24]:
```
# print AUC score and ROC curve
get_ROC(model_dec_tree, x_test, y_test)
```

```
AUC score:
```

```
0.8448320141989702
```



[25]:
```
# create overview of feature importance, of learned decision tree
attribute_weights = pd.DataFrame({
    'Attribute' : x_train.columns,
    'Weight' : model_dec_tree.feature_importances_
}).sort_values(by='Weight', ascending=False)
plt.title('Importance of different Attributes')
sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```



10.1 Random Forest

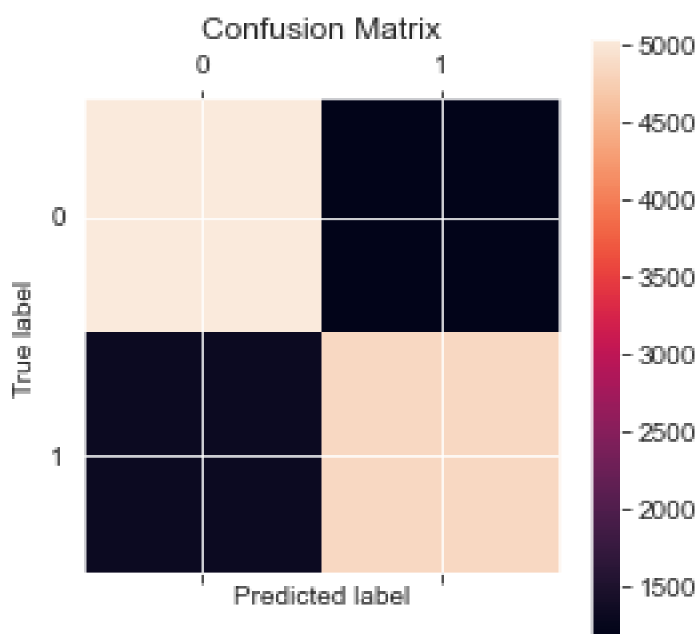
[26]:
```
param_grid_forest = {
    'criterion': ['gini','entropy'],
    'max_depth': range(5,25)
}

"""
Trains a random forest using cross-validation and returns certain attributes of␣
 ↪the received model including the best
parameter combination.
@param x_train, np ndarray, data matrix
@param y_train, np ndarray, data vector
@param param_grid, dict, grid holding the paramaters for search
"""
def train_random_forest(x_train,y_train,param_grid):
    ensemble = RandomForestClassifier(random_state=55)
    model = GridSearchCV(ensemble,param_grid=param_grid, n_jobs = -1)
    model.fit(x_train,y_train)
    return model.best_params_,model.best_estimator_
```

[27]:
```
# train decision tree with created training set and evaluate on created target␣
 ↪set
params_random_forest, model_random_forest = train_random_forest(x_train,␣
 ↪y_train, param_grid_forest)
```

```python
_ = get_evaluation(model_random_forest, x_test, y_test)
print("The best parameters are: {}".format(params_random_forest))
```

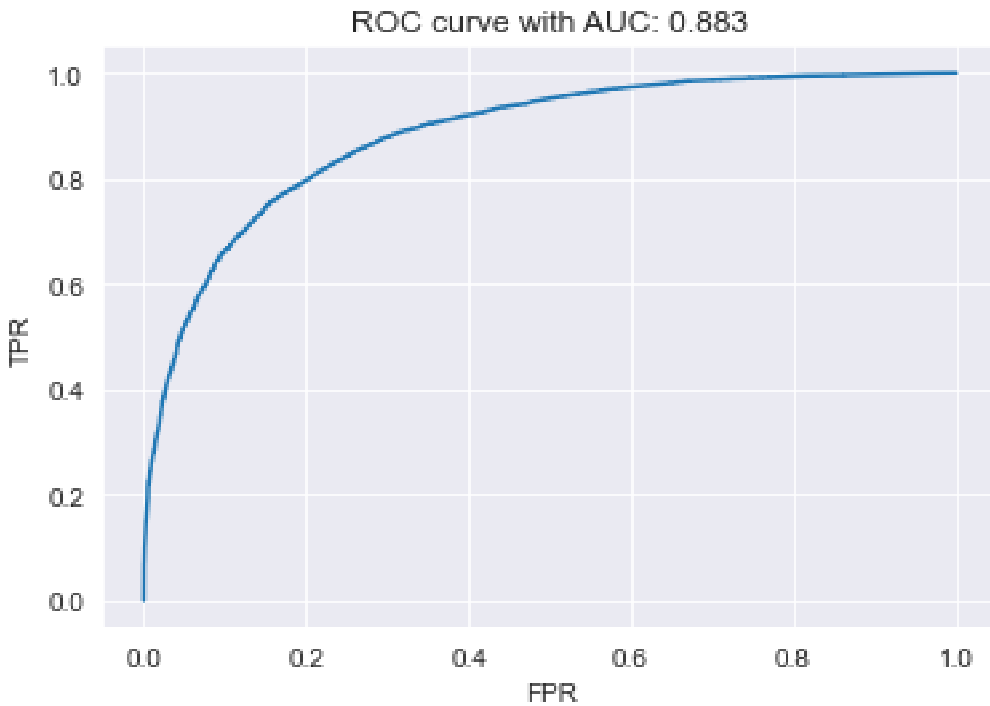
```
Accuracy of Classifier on Test Image Data:  0.7984677419354839

Recall (No Rain Tomorrow) of Classifier on Test Image Data:  0.809432146294514
Recall (Rain Tomorrow) of Classifier on Test Image Data:  0.7873824197210509

Precision (No Rain Tomorrow) of Classifier on Test Image Data:
0.7937706465313827
Precision (Rain Tomorrow) of Classifier on Test Image Data:  0.8034089028628165


Confusion Matrix:
 [[5046 1188]
 [1311 4855]]
The best parameters are: {'criterion': 'entropy', 'max_depth': 23}
```

```
C:\Users\fnern\AppData\Local\Temp\ipykernel_19852\1287647186.py:27:
MatplotlibDeprecationWarning: Auto-removal of grids by pcolor() and pcolormesh()
is deprecated since 3.5 and will be removed two minor releases later; please
call grid(False) first.
  plt.colorbar()
```

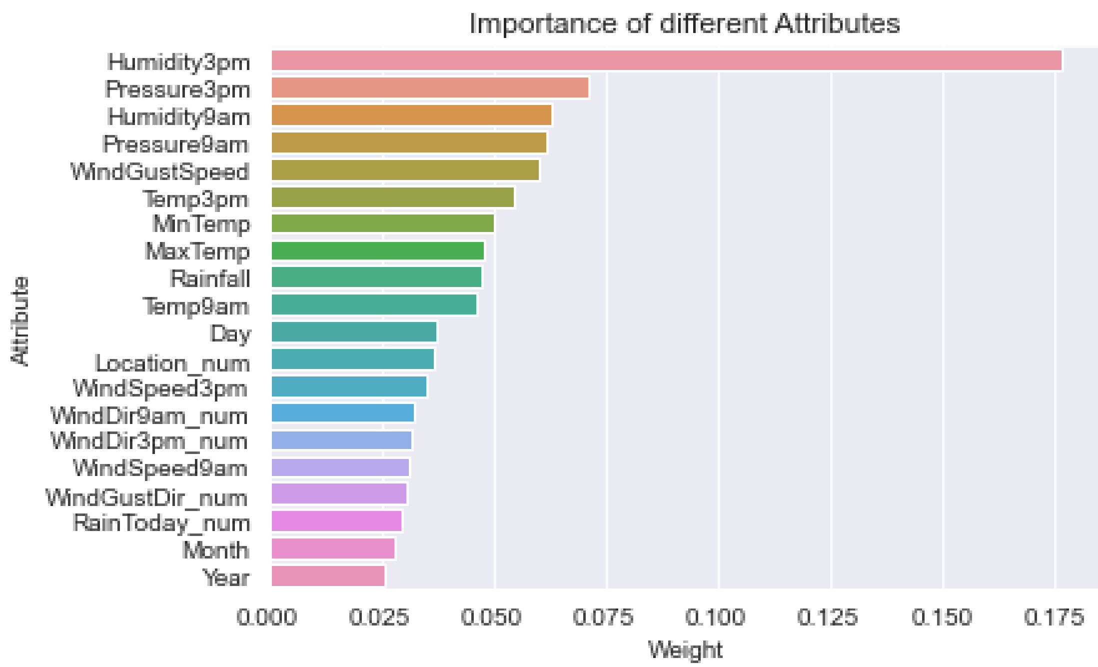


[28]:
```
# print AUC score and ROC curve
get_ROC(model_random_forest, x_test, y_test)
```

```
AUC score:
0.8826908270186274
```



[29]:
```
# create overview of feature importance, of learned decision tree
attribute_weights = pd.DataFrame({
    'Attribute' : x_train.columns,
    'Weight' : model_random_forest.feature_importances_
}).sort_values(by='Weight', ascending=False)
plt.title('Importance of different Attributes')
sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```



11 Extreme Gradient Boosting

[30]:
```
xgb = XGBClassifier()
xgb.fit(x_train, y_train)
```

```
C:\Users\fnern\miniforge3\envs\stat\lib\site-packages\xgboost\sklearn.py:1224:
UserWarning: The use of label encoder in XGBClassifier is deprecated and will be
removed in a future release. To remove this warning, do the following: 1) Pass
option use_label_encoder=False when constructing XGBClassifier object; and 2)
Encode your labels (y) as integers starting with 0, i.e. 0, 1, 2, …,
[num_class - 1].
  warnings.warn(label_encoder_deprecation_msg, UserWarning)
```

```
[18:51:36] WARNING: D:\bld\xgboost-split_1645118015404\work\src\learner.cc:1115:
Starting in XGBoost 1.3.0, the default evaluation metric used with the objective
'binary:logistic' was changed from 'error' to 'logloss'. Explicitly set
eval_metric if you'd like to restore the old behavior.
```

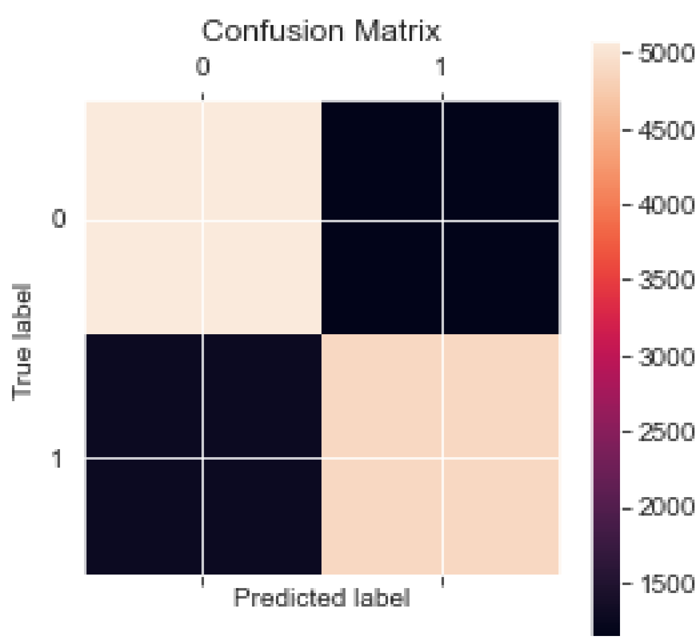
[30]:
```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, enable_categorical=False,
              gamma=0, gpu_id=-1, importance_type=None,
              interaction_constraints='', learning_rate=0.300000012,
              max_delta_step=0, max_depth=6, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=100, n_jobs=12,
              num_parallel_tree=1, predictor='auto', random_state=0,
```

```
                 reg_alpha=0, reg_lambda=1, scale_pos_weight=1, subsample=1,
                 tree_method='exact', validate_parameters=1, verbosity=None)
```

[31]:
```
_ = get_evaluation(xgb, x_test, y_test)
```

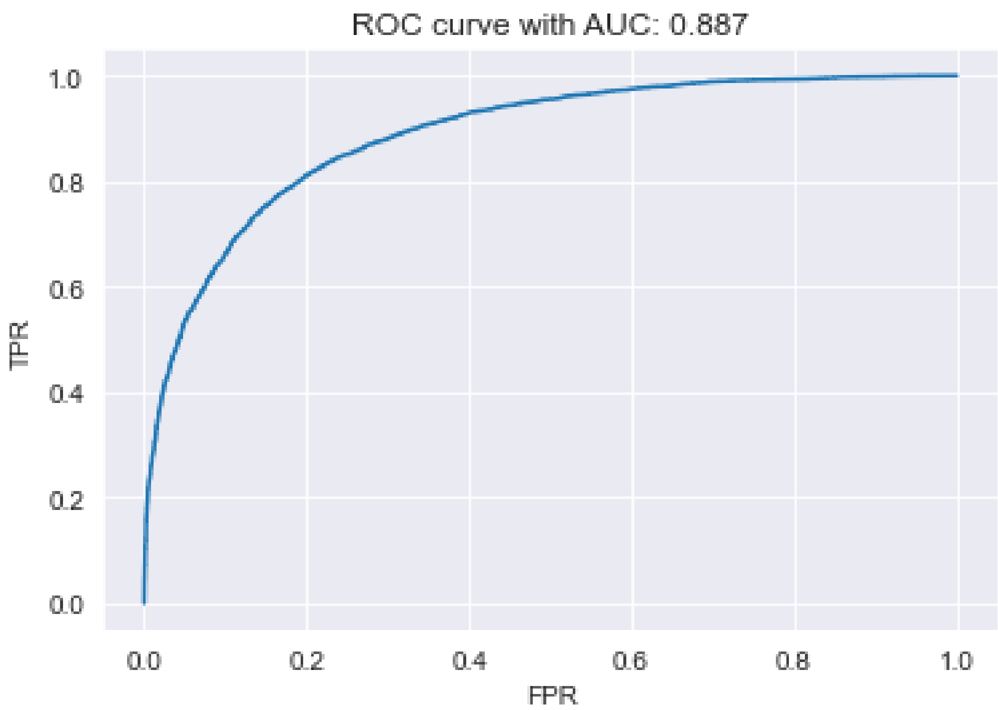
```
Accuracy of Classifier on Test Image Data:  0.8042741935483871

Recall (No Rain Tomorrow) of Classifier on Test Image Data:  0.8156881616939364
Recall (Rain Tomorrow) of Classifier on Test Image Data:  0.7927343496594227

Precision (No Rain Tomorrow) of Classifier on Test Image Data:
0.7991513437057991
Precision (Rain Tomorrow) of Classifier on Test Image Data:  0.8096736789796256


Confusion Matrix:
 [[5085 1149]
 [1278 4888]]
```

```
C:\Users\fnern\AppData\Local\Temp\ipykernel_19852\1287647186.py:27:
MatplotlibDeprecationWarning: Auto-removal of grids by pcolor() and pcolormesh()
is deprecated since 3.5 and will be removed two minor releases later; please
call grid(False) first.
  plt.colorbar()
```

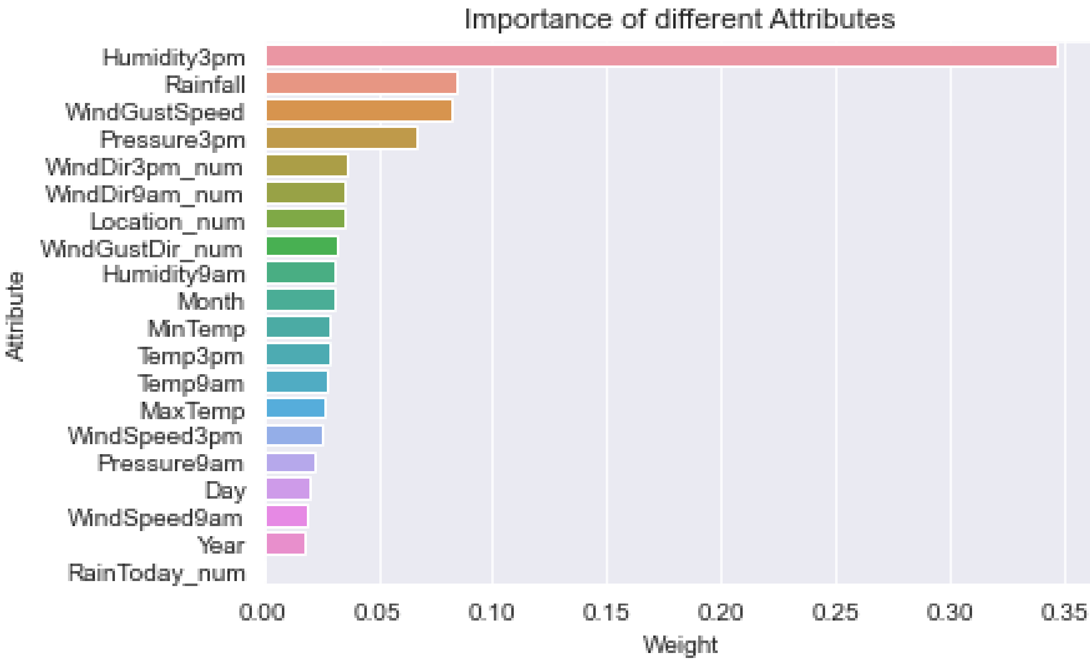


[32]:
```
# print AUC score and ROC curve
get_ROC(xgb, x_test, y_test)
```

```
AUC score:
0.8872340047479055
```



[33]:
```
# create overview of feature importance, of learned decision tree
attribute_weights = pd.DataFrame({
    'Attribute' : x_train.columns,
    'Weight' : xgb.feature_importances_
}).sort_values(by='Weight', ascending=False)
plt.title('Importance of different Attributes')
sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```



12 Regression

In this part, are going to develop an estimator for the rainfall. Since, rainfall is a continuous variable, this is obviously a regression task.

Since, this is going to be a multiple regression task, and therefore, not all variables might have a significant impact, we chose the best subset selection method for identifying the required variables.

12.1 Data preparation for the regression part

[34]:
```
x_train_reg = x_train.loc[x_train['Rainfall'] > 0, x_train.columns !=␣
↪'Rainfall'].copy()
x_test_reg = x_test.loc[x_test['Rainfall'] > 0, x_test.columns != 'Rainfall'].
↪copy()

y_train_reg = x_train[x_train['Rainfall'] > 0]['Rainfall'].copy()
y_test_reg = x_test[x_test['Rainfall'] > 0]['Rainfall'].copy()
```

Now, after the data is prepared for the regression part, we can now start to fit some regression models. We decided to use the regression version of our classifiers.

Our first model is the regression tree.

12.2 Regression tree

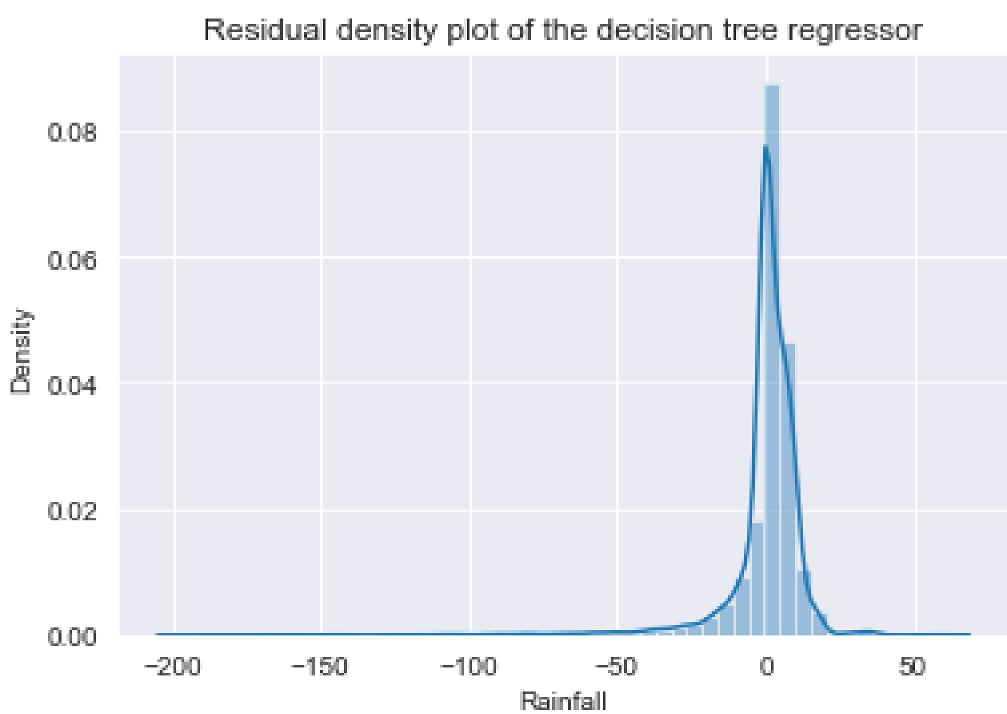
[35]:
```
"""
Evaluates the regression model.
@param model, sklearn model,trained model
@param x_test, np ndarray, data matrix
@param y_test, np ndarray, data vector
@param plot_title, str, the plot title
"""
def get_regression_evaluation(model, x_test, y_test, plot_title: str):
    y_pred = model.predict(x_test)

    explained_variance = explained_variance_score(y_test, y_pred)
    m_squared = mean_squared_error(y_test, y_pred)
    absolute = mean_absolute_error(y_test, y_pred)
    r2 = r2_score(y_test, y_pred)

    print(f"Explained variance: {explained_variance:.4f}")
    print(f"Mean squared error: {m_squared:.4f}")
    print(f"RMSE: {np.sqrt(m_squared):.4f}")
    print(f"Mean absolute error: {absolute:.4f}")
    print(f"R2 score: {r2:.4f}")

    sns.distplot(y_pred - y_test)
    plt.title(plot_title)

    return None
```

[36]:
```
dec_tree_grid = {
    'criterion': ['squared_error','absolute_error'],
    'max_depth': range(1,10),
    'splitter': ['random', 'best'],
    "max_features":["auto", "sqrt", None],
}

"""
Trains a decision tree regressor using cross-validation and returns attributes␣
 ↪of the received model including the best
parameter combination.
@param x_train, np ndarray, data matrix
@param y_train, np ndarray, data vector
@param param_grid, dict, grid holding the paramaters for search
@param use_pref_defined_model, bool, indicates whether the predefined model␣
 ↪version should be used
"""
def train_dec_tree_regressor(x_train, y_train, param_grid,␣
 ↪use_pref_defined_model: bool):
```

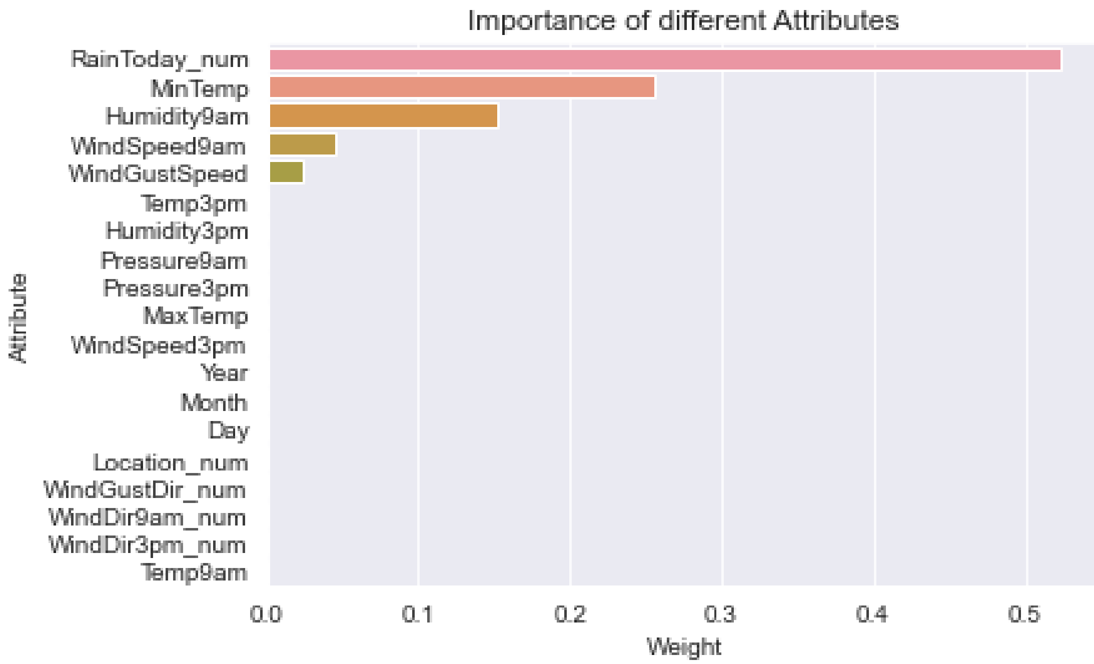
```
    if use_pref_defined_model:
        best_params =  {'criterion': 'squared_error', 'max_depth': 4,␣
↪'max_features': 'auto', 'splitter': 'best'}
        tree = DecisionTreeRegressor(random_state=55, **best_params)
        tree.fit(x_train, y_train)
        return best_params, tree

    tree = DecisionTreeRegressor(random_state=55)
    model = GridSearchCV(tree, param_grid=param_grid,␣
↪scoring="neg_mean_squared_error", verbose=10)
    model.fit(x_train, y_train)
    return model.best_params_, model.best_estimator_
```

[37]:
```
params_dec_tree_regressor, model_dec_tree_regressor =␣
↪train_dec_tree_regressor(x_train_reg, y_train_reg, dec_tree_grid, True)
_ = get_regression_evaluation(model_dec_tree_regressor, x_test_reg, y_test_reg,␣
↪"Residual density plot of the decision tree regressor")
#print("The best parameters are: {}".format(params_dec_tree_regressor))
```

```
Explained variance: 0.2025
Mean squared error: 164.8163
RMSE: 12.8381
Mean absolute error: 6.3843
R2 score: 0.2022
```



```
[38]: attribute_weights = pd.DataFrame({
          'Attribute' : x_train_reg.columns,
          'Weight' : model_dec_tree_regressor.feature_importances_
      }).sort_values(by='Weight', ascending=False)
      plt.title('Importance of different Attributes')
      sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```



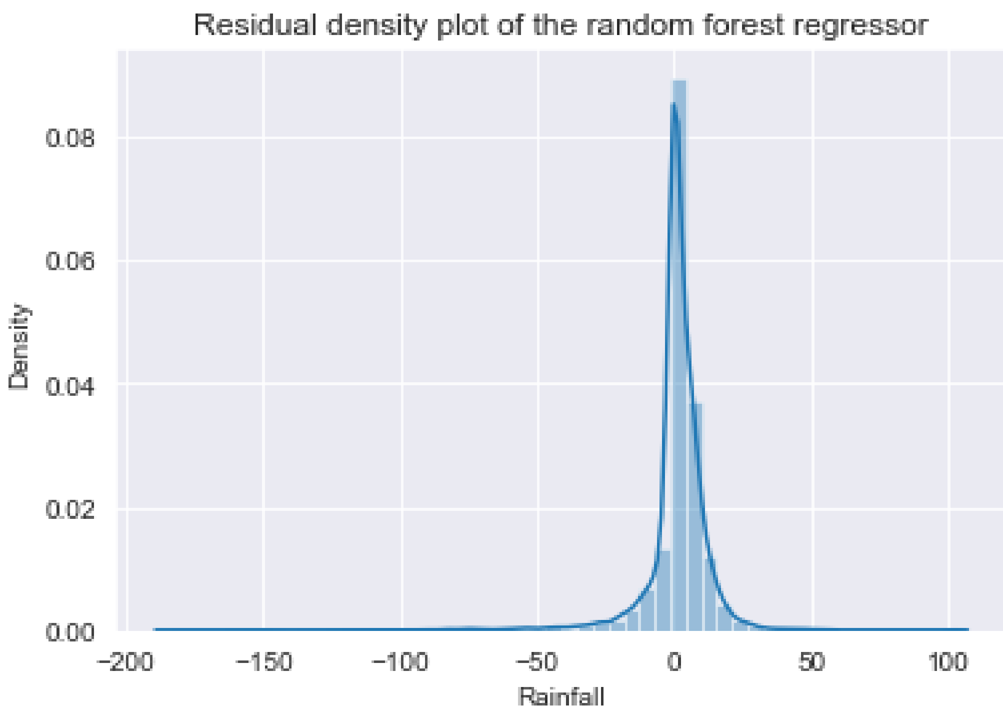
12.2.1 Random Forest Regressor

```
[39]: rand_forest_reg_grid = {
          'n_estimators': [50, 100, 150, 200],
          'criterion': ['squared_error','absolute_error'],
          "max_features":["auto", "sqrt"],
      }

      """
      Trains a random forest regressor using cross-validation and returns attributes␣
       ↪of the received model including the best
      parameter combination.
      @param x_train, np ndarray, data matrix
      @param y_train, np ndarray, data vector
      @param param_grid, dict, grid holding the paramaters for search
```

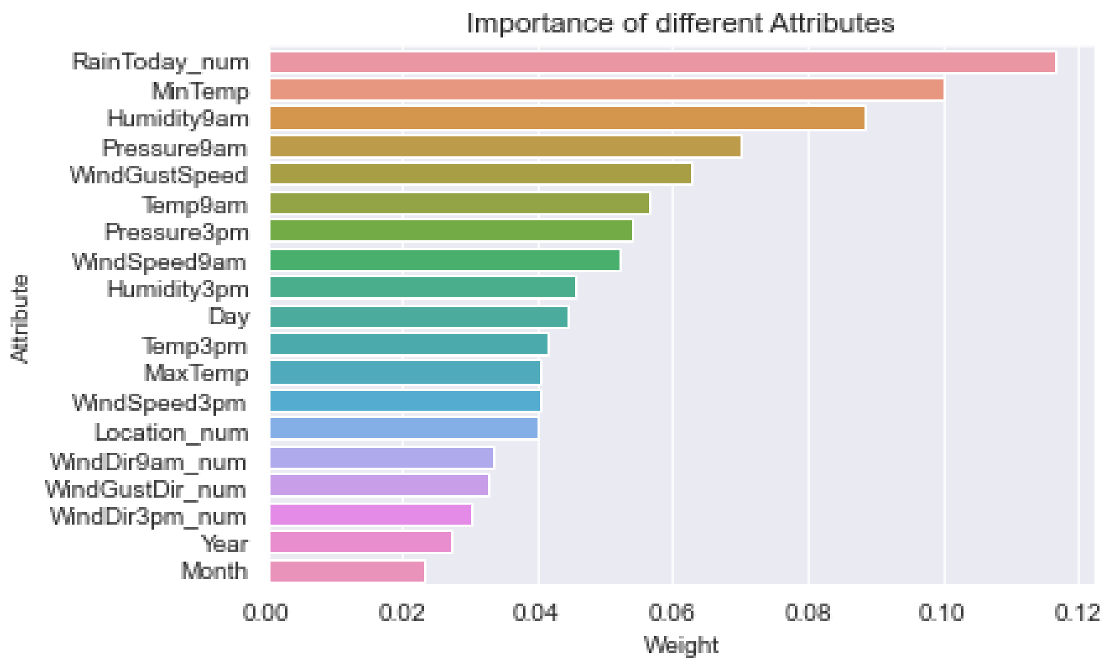
```python
@param use_pref_defined_model, bool, indicates whether the predefined model␣
 ↪version should be used
"""
def train_random_forest_regressor(x_train, y_train, param_grid,␣
 ↪use_pref_defined_model: bool):
    if use_pref_defined_model:
        best_params = {'criterion': 'squared_error', 'max_features': 'sqrt',␣
 ↪'n_estimators': 200}
        forest = RandomForestRegressor(random_state=55)
        forest.fit(x_train, y_train)
        return best_params, forest

    forest = RandomForestRegressor(random_state=55)
    model = GridSearchCV(forest, param_grid=param_grid,␣
 ↪scoring="neg_mean_squared_error", verbose=10)
    model.fit(x_train, y_train)
    return model.best_params_, model.best_estimator_
```

[40]:
```python
params_random_forest_regressor, model_random_forest_regressor =␣
 ↪train_random_forest_regressor(x_train_reg, y_train_reg,␣
 ↪rand_forest_reg_grid, True)
_ = get_regression_evaluation(model_random_forest_regressor, x_test_reg,␣
 ↪y_test_reg, "Residual density plot of the random forest regressor")
#print("The best parameters are: {}".format(params_random_forest_regressor))
```

```
Explained variance: 0.3143
Mean squared error: 142.9021
RMSE: 11.9542
Mean absolute error: 6.1734
R2 score: 0.3083
```



[41]:
```
attribute_weights = pd.DataFrame({
    'Attribute' : x_train_reg.columns,
    'Weight' : model_random_forest_regressor.feature_importances_
}).sort_values(by='Weight', ascending=False)
plt.title('Importance of different Attributes')
sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```



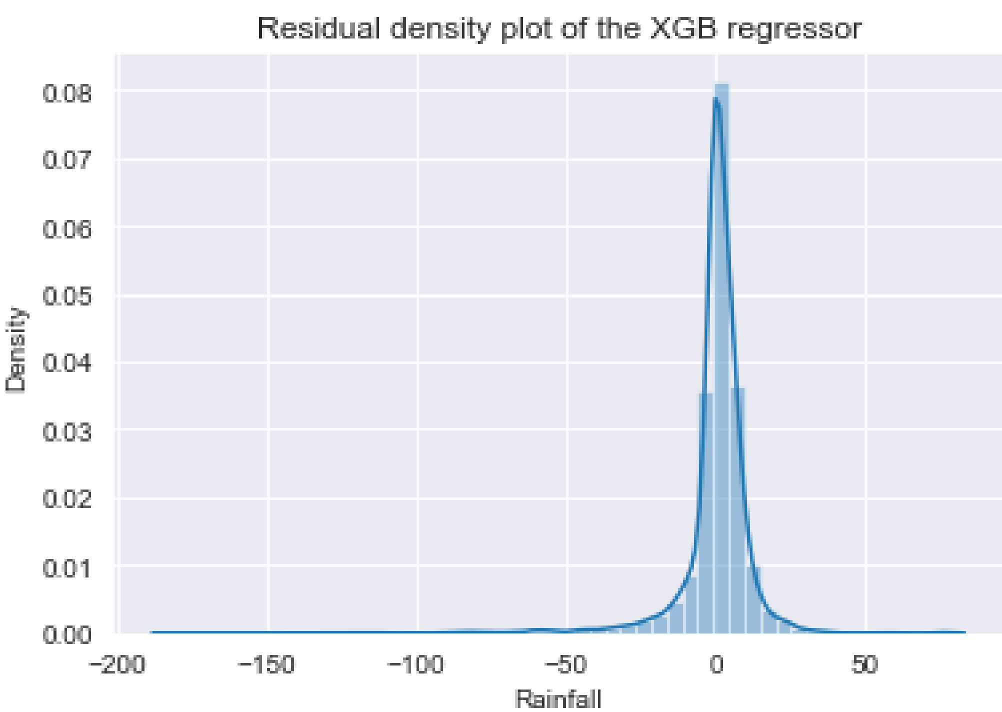
12.3 Extreme Gradient Boosting Regression

[42]:
```
xgb_grid = {
    'max_depth': [3,6,10],
    'learning_rate': [0.01, 0.05, 0.1],
    'n_estimators': [100, 500, 1000],
    'colsample_bytree': [0.3, 0.7]
}

"""
Trains an XGB regressor using cross-validation and returns attributes of the␣
↪received model including the best
parameter combination.
@param x_train, np ndarray, data matrix
@param y_train, np ndarray, data vector
@param param_grid, dict, grid holding the paramaters for search
@param use_pref_defined_model, bool, indicates whether the predefined model␣
↪version should be used
"""
def train_xgb_regressor(x_train, y_train, param_grid, use_pref_defined_model:␣
↪bool):
    if use_pref_defined_model:
        best_params = {'colsample_bytree': 0.3, 'learning_rate': 0.05,␣
↪'max_depth': 6, 'n_estimators': 500}
        xgb = XGBRegressor(seed = 55, **best_params)
```

```
        xgb.fit(x_train, y_train)
        return best_params, xgb
    xgb = XGBRegressor(seed = 55)
    model = GridSearchCV(xgb, param_grid=param_grid,␣
 ↪scoring="neg_mean_squared_error", verbose=10)
    model.fit(x_train, y_train)
    return model.best_params_, model.best_estimator_
```

[43]:
```
xgb_params, xgb_regressor = train_xgb_regressor(x_train_reg, y_train_reg,␣
 ↪xgb_grid, True)
_ = get_regression_evaluation(xgb_regressor, x_test_reg, y_test_reg, "Residual␣
 ↪density plot of the XGB regressor")
#print(f"The best parameters are: {xgb_params}")
```

```
Explained variance: 0.3290
Mean squared error: 138.7996
RMSE: 11.7813
Mean absolute error: 6.1798
R2 score: 0.3282
```



[44]:
```
attribute_weights = pd.DataFrame({
    'Attribute' : x_train_reg.columns,
    'Weight' : xgb_regressor.feature_importances_
}).sort_values(by='Weight', ascending=False)
```

```
plt.title('Importance of different Attributes')
sns.barplot(data = attribute_weights, x='Weight', y='Attribute');
```