Logistic Regression (classification)

**unlike linear, logistic regression is predominently used to in classification problems. where as linear is used for the continuous data For instace....

1. to predict whether tommorow will rain or not
2. Based on prior transaction, shall a bank lend the loan to the perticular customer or not
3. spam prediction is also a classification problem. that can be solved by logistic reg
4. More than two option to make descision, such as classification of numbers

Majorly in case of linear regression we take all faetures and apply weights to them (weighted sum) along with the biase finally get a result. But in case of of logistic regression we go further and put the weighted sum result to activation function called *"sigmoid"***
**which pushes whole and gives result in between 0 and 1.

$$\sigma(z) = \left(\frac{1}{1+e^{-z}}\right)$$

In [147]:
```
url='https://www.kaggle.com/jsphyg/weather-dataset-rattle-package'

from urllib.request import urlopen
```

In [148]:
```
import json

with open('path of json file','r') as f:
    data=json.load(f)
    print(data['username'])


key='*********************'
username='xxxxxxxx'
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-148-3bb489f2a285> in <module>
      1 import json
      2
----> 3 with open('path of json file','r') as f:
      4     data=json.load(f)
      5     print(data['username'])

FileNotFoundError: [Errno 2] No such file or directory: 'path of json file'
```

In [ ]:
```
### library to download datasets from kaggle (easy approach)
#!pip install opendatasets --upgrade --quiet
```

In [ ]:
```
import opendatasets as od
```

In [ ]:
```
#od.download(url)
```

In [ ]:
```
od.download(url)
```

In [ ]:
```
data_path='.\weather-dataset-rattle-package'
```

In [ ]:
```
import os
os.listdir(data_path)
```

In [ ]:
```
path=data_path+'/weatherAUS.csv'
path
```

In [ ]:
```
import pandas as pd

data=pd.read_csv('.//weather-dataset-rattle-package/weatherAUS.csv')
```

In [ ]:
```
pd.set_option('display.max_row', 50)
data
```

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

In [ ]:
```
data.describe()
```

In [ ]:
```
data.Cloud9am.isnull().sum()
```

In [ ]:
```
data.dropna(subset=['RainToday','RainTomorrow'], inplace=True)
```

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

Exploratory Data Analysis

it is neccessary to perform EDA before fitting data into model for training

In [ ]:
```
import numpy as np
import plotly.express as px
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

In [ ]:
```
sns.set_style('darkgrid')
matplotlib.rcParams['font.size']=16
matplotlib.rcParams['figure.figsize']=(14,6)
```

In [ ]:
```
data.nunique()
```

In [ ]:
```

fig=px.histogram(data,x='Location', color='RainToday', marginal='box', title=

fig.update_layout(bargap=0.2)

fig.show()
```

In [ ]:
```
data.nunique()
```

In [ ]:
```
fig=px.histogram(data, x='Temp3pm', color='RainTomorrow', title='temperature
fig.update_layout(bargap=0.1)
fig.show()
```

In [ ]:
```
sns.histplot(data=data, x='Temp3pm', hue='RainTomorrow',kde=True);
```

In [ ]:
```
px.histogram(data, x='RainTomorrow', color='RainToday', title='Rain today vs
```

In [ ]:
```
px.scatter(data.sample(2000), x='MinTemp',y='MaxTemp',title='max and mini tem
```

observtion

form plotting max and mini temperature with respect to RainToday, we can infer that, there is not much difference in the max and min temperaure of that day when it was rained.

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

In [ ]:
```
"""""
we specified 20% of data form whole as test , but we did not specified which
set as argument which normally provoke random number generator to pick the da
"""""
train_val, test=train_test_split(data, test_size=0.20, random_state=42)
train, validate=train_test_split(train_val, test_size=0.25, random_state=42)
```

In [ ]:
```
print("\033[1m training data :", train.shape)
print("validation data :", validate.shape)
print("test data :",test.shape)
```

In [ ]:
```
sns.countplot(x=pd.to_datetime(data.Date).dt.year)
plt.title('No of rows per year')
```

observation

However as we split the whole dataset into train, validate and test datasets, where in, each have mixed records with respect to Date
At the end, we train model with train dataset which will be having records collected from every year(because data is not filtered with time) and same goes with validation and test data. But here our main objective of building ML model is to predict feature events occurance based on past events data.Hence the train data is filtered with records merely collected before the data records present in test dataset.......

train->contain data records below 2015
validation->data collected at 2015
test->data collected after 2015

NOTE: "Below filterring of data will give more understandability"

In [ ]:
```
#### ignoring data split done prior
year=pd.to_datetime(data.Date).dt.year

train=data[year<2015]
validate=data[year==2015]
test=data[year>2015]
```

In [ ]:
```
### we choosing columns from datasets, where redundant  columns were ignored

input_cols=list(data.columns[1:-1])
## target column selection
target_col='RainTomorrow'
```

In [ ]:
```
input_cols
```

In [ ]:
```
train_input=train[input_cols].copy()
train_target=train[target_col].copy()
```

In [ ]:
```
test_input=test[input_cols].copy()
test_target=test[target_col].copy()
```

In [ ]:
```
val_input=validate[input_cols].copy()
val_target=validate[target_col].copy()
```

In [ ]:
```
# 'RainTomorrow'column is ignored along with the redundant
val_input
```

In [ ]:
```
##  'RainTomorrow' column records in pandas series (type)
val_target
```

points to remember

1. Ignored Date column because it does not contribute to the final result

2. For instance if we have rainfall_Tomorrow column in our datasets, it should be ignored too. Since we are predicting whether it will rain or not
3. We forging our model in accordance of location in the dataset (i,e in 49 locations data has been recorded), hence location place important role as each location have thier own unique climate attribute.

MOST IMPORTANTLY, THIS MODEL WILL WORK WELL ONLY WITH THE INPUTS MERELY HAVING THESE LOCATIONS. HOWEVER TO GENERALIZE (works irrespective of location) THE MODEL WE NEED TO IGNORE LOCATION COLUMN.(moreover these data is inadequate to generalize the model)

saparating numerical data and categorical data

In [ ]:
```
import numpy as np
```

In [ ]:
```
numeric_cols= train_input.select_dtypes(include=np.number).columns.tolist()
categorical=train_input.select_dtypes('object').columns.tolist()
```

In [ ]:
```
train_input[numeric_cols].describe()
```

In [ ]:
```
train_input[categorical].nunique()
```

Dealing with missing values (null/nNaN)

The process of filling missing values with valid values is called *IMPUTING*

1. there are many different approches of imputing one such is filling with **mean value** of that column
2. using ***median*** is also a best approach because sometimes there will be outliers which may affect the average as whole.

In [ ]:
```
### Sklearn imputer class is called for imputing

from sklearn.impute import SimpleImputer
```

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

In [ ]:
```
data[numeric_cols].isna().sum()
```

In [ ]:
```
train_input[numeric_cols].isna().sum()
```

In [ ]:
```
###imputer will return specified statistic (in this case mean) value
### remember it will not replace the missing value with stats value
imputer.fit(data[numeric_cols])
```

In [ ]:
```
imputer.statistics_
```

In [ ]:
```
imputer.transform(train_input[numeric_cols])
## above imputer just created numpy array but we need it in our dataframe hen
```

In [ ]:
```
### overwriting

train_input[numeric_cols]=imputer.transform(train_input[numeric_cols])
```

In [ ]:
```
train_input[numeric_cols].head(10)
```

In [ ]:
```
val_input[numeric_cols]=imputer.transform(val_input[numeric_cols])
val_input[numeric_cols].head(10)
```

In [ ]:
```
test_input[numeric_cols]=imputer.transform(test_input[numeric_cols])
```

scaling numeric columns

**Numeric columns are scaled to small range values. where it is neccessary to scale the data in order to avoid the disproportionate effect of perticular feature on model's loss, and to avoid the adverse effect of optimizer

In [ ]:
```
data.describe()
```

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
```

In [ ]:
```
?MinMaxScaler
```

In [ ]:
```
scaler=MinMaxScaler()

scaler.fit(data[numeric_cols])
```

In [ ]:
```
scaler.data_min_ , scaler.data_max_
```

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

In [ ]:
```
val_input[numeric_cols]
```

In [ ]:
```
train_input[numeric_cols].describe()
```

Lets deal with the categorical columns

Our obvious approach.... either one hot encoding or encoding

In [ ]:
```
### having comprehensive view on categorical

data[categorical].nunique()
```

In [ ]:
```
data.Location.unique()
```

In [ ]:
```
from sklearn.preprocessing import OneHotEncoder
```

In [ ]:
```
encode=OneHotEncoder(sparse=False, handle_unknown='ignore')
```

In [ ]:
```
### dealing with nan values
data_2=data[categorical].fillna('unkown')
```

In [ ]:
```
encode.fit(data_2)
```

In [ ]:
```
encode.categories_
```

In [ ]:
```
##generating column names for each columns by using get_feature_names method

encode_cols=list(encode.get_feature_names(categorical))

for i in encode_cols:
    print(i)
```

In [ ]:
```
train_input_2=train_input[categorical].fillna('unknown')
train_input[encode_cols]=encode.transform(train_input_2)

val_input_2=val_input[categorical].fillna('unknown')
val_input[encode_cols]=encode.transform(val_input_2)


test_input[encode_cols]=encode.transform(test_input[categorical].fillna('unkr
```

In [ ]:
```
## to see all columns
pd.set_option('max_columns',None)
train_input.head(100)
```

In [ ]:
```
##lets look into the shape of each data chunk

print('\033[1m Train input :', train_input.shape)
print('\033[1m Train target :', train_target.shape)
print('\033[1m validation input :', val_input.shape)
print('\033[1m validation traget :', val_target.shape)
print('\033[1m test input :', test_input.shape)
print('\033[1m Test target :', test_target.shape)
```

In [ ]:
```
""""
it is optional!!!
after all data preprocessing, if we need clean data to be used for another pr
it as csv or another efficient format is parquet, for that we need to install
"""""
#!pip install pyarrow --upgrade --quiet
```

In [ ]:
```
train_input.to_parquet('train_input.parquet')
val_input.to_parquet('val_input.parquet')
test_input.to_parquet('test_input.parquet')
```

In [ ]:
```

train_in=pd.read_parquet('train_input.parquet')
train_in.tail(5)
```

Finally our datasets are ready to be used to train our reggression model

\logistic \regression

1. first we take weighted sum of each input features, as we do in liner regression
2. Result set from the above set is fed into the the sigmoid function, which gives the result either 0 or 1
3. and at lost, to reduce the cost function we use cross entropy loss function instead of RMSE

Sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

cross entorpy loss function

$$L(y^*, y) = -ylogy^* + (1 - y)log(1 - y^*)$$

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

In [ ]:
```
## 'liblinear is the optimization '
model=LogisticRegression(solver='liblinear')
```

In [ ]:
```
%%time
### only nummeric columns need to be fed into the model in case of inputs
## but in case of targets sklearn will covert the categories into numeric acc
model.fit(train_input[numeric_cols + encode_cols], train_target)
```

In [ ]:
```
print(list(numeric_cols+encode_cols))
```

In [ ]:
```
##weights of each
print(list(model.coef_)[0])
```

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

In [ ]:
```
pd.set_option('max_rows',None)
weights=pd.DataFrame({
    'features': numeric_cols + encode_cols [:len(numeric_cols + encode_cols)]
    'weights' : model.coef_[0]
})
```

In [ ]:
```
plt.figure(figsize=(5,5))
sns.barplot(data=weights.sort_values('weights',ascending=False).head(10), x='
```

In [ ]:
```
sns.barplot(data=weights.sort_values('weights',ascending=True).head(10), x='w
```

In [ ]:
```
train_input.head(10)
```

In [ ]:
```
x_train=train_input[numeric_cols+encode_cols]

x_val=val_input[numeric_cols+encode_cols]

x_test=test_input[numeric_cols+encode_cols]
```

model prediction

In [ ]:
```
prediction=model.predict(x_train)
train_target.shape
```

In [ ]:
```

comp_df=pd.DataFrame({
    'Actual' : train_target,
    'prediction': prediction
})
px.histogram(comp_df,x='Actual', color='prediction')
```

In [ ]:
```
from sklearn.metrics import accuracy_score
```

In [ ]:
```
accuracy_score(train_target,prediction)
```

using a probability for prediction is a good meaure of confidence

In [ ]:
```
###probabilistic prediction,and this is only for logistic regressiion

train_prob=model.predict_proba(x_train)
```

In [ ]:
```
train_prob,model.classes_
```

confusion Matrix

True postive-->> prediction is true and actual is also true
True negative-->> prediction is false and the actual is also false
False positive--->> prediction is true but actual is false (Type1 error) **
****False negative --->> prediction is false but actual is true Type2 error)**

In [ ]:
```
### confusion matrix

from sklearn.metrics import confusion_matrix
```

In [ ]:
```
confusion_matrix(train_target,prediction, normalize='true')
```

In [ ]:
```
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
```

In [ ]:
```
def accuracy_cofmatrix(data_input,data_target, name=''):
    prediction=model.predict(data_input)
    accuracy=accuracy_score(data_target, prediction)
    print('\033[1m accuracy:',accuracy*100,'%')
    cf=confusion_matrix(data_target,prediction, normalize='true')
    sns.heatmap(cf,annot=True)
    plt.xlabel('prediction')
    plt.ylabel('target')
    plt.title('confusion matrix of {}'.format(name))
```

In [ ]:
```
accuracy_cofmatrix(test_input[numeric_cols+encode_cols],test_target,'validati
```

lets use some base line models to compare with our model

In [ ]:
```
def random_predict(inputs):
    rand_pred=np.random.choice(['no','yes'],len(inputs))
    return rand_pred

def all_yes(inputs):
    return np.full(len(inputs),'yes')

### given both our targets column with any of these functions ouput, find acc
```

lets give random input to our model (single input)

In [204]:
```
new_data={'Date':'2021-10-20',
          'Location':'SydneyAirport',
          'MinTemp':23.2,
          'MaxTemp':33.4,
          'Rainfall':10.2,
          'Evaporation':4.8,
          'Sunshine':np.nan,
          'WindGustDir':'NNW',
          'WindGustSpeed':10.0,
          'WindDir9am':'NW',
          'WindDir3pm':'NNE',
          'WindSpeed9am':13.0,
          'WindSpeed3pm':20.4,
          'Humidity9am':89.2,
          'Humidity3pm':58.0,
          'Pressure9am':1000,
          'Pressure3pm':1001.5,
          'Cloud9am':8.0,
          'Cloud3pm':5.0,
          'Temp9am':25.7,
          'Temp3pm':33.0,
          'RainToday':'Yes',


}
```

In [205]:
```
new_df=pd.DataFrame([new_data])
```

In [206]:
```
new_df
```

Out[206]:

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	W
0	2021-10-20	SydneyAirport	23.2	33.4	10.2	4.8	NaN	NNW	

In [207]:
```
#imputing to remove Nan values
new_df[numeric_cols]=imputer.transform(new_df[numeric_cols])

#scaling/standardizing column values to make it unifrom and to go easy while
new_df[numeric_cols]=scaler.transform(new_df[numeric_cols])

#one hot encoding categorical columns
new_df[encode_cols]=encode.transform(new_df[categorical])
```

In [208]:
```
prediction=model.predict(new_df[numeric_cols+encode_cols])

prediction
```

Out[208]:
```
array(['No'], dtype=object)
```

In [209]:
```
prob=model.predict_proba(new_df[numeric_cols+encode_cols])

prob, model.classes_
```

Out[209]:
```
(array([[0.86995778, 0.13004222]]), array(['No', 'Yes'], dtype=object))
```

In [210]:
```
## to fetch other locations name
data.Location.unique()
```

Out[210]:
```
array(['Albury', 'BadgerysCreek', 'Cobar', 'CoffsHarbour', 'Moree',
       'Newcastle', 'NorahHead', 'NorfolkIsland', 'Penrith', 'Richmond',
       'Sydney', 'SydneyAirport', 'WaggaWagga', 'Williamtown',
       'Wollongong', 'Canberra', 'Tuggeranong', 'MountGinini', 'Ballarat',
       'Bendigo', 'Sale', 'MelbourneAirport', 'Melbourne', 'Mildura',
       'Nhil', 'Portland', 'Watsonia', 'Dartmoor', 'Brisbane', 'Cairns',
       'GoldCoast', 'Townsville', 'Adelaide', 'MountGambier', 'Nuriootpa',
       'Woomera', 'Albany', 'Witchcliffe', 'PearceRAAF', 'PerthAirport',
       'Perth', 'SalmonGums', 'Walpole', 'Hobart', 'Launceston',
       'AliceSprings', 'Darwin', 'Katherine', 'Uluru'], dtype=object)
```

like this we can use random inputs(as we done above) to check how good the model is.

lets save trained model and its parameters, so that we can use that later

In [211]:
```
import joblib
```

In [214]:
```
Aust_predictor={
    'model':model,
    'imputer':imputer,
    'scaler':scaler,
    'encoder':encode,
    'input_cols':input_cols,
    'target_cols':target_col,
    'numeric_cols':numeric_cols,
    'categorical_cols':categorical,
    'encode_cols':encode_cols
}
```

In [215]:
```
joblib.dump(Aust_predictor,'aust_prediction.joblib')
```

Out[215]:
```
['aust_prediction.joblib']
```

In [216]:
```
model_1=joblib.load('aust_prediction.joblib')
```

In [219]:
```
model_1['model']
```

Out[219]:
```
LogisticRegression(solver='liblinear')
```

lets make some prediction using the model stored in the joblin

In [221]:
```
test_predict=model_1['model'].predict(x_test)
accuracy_score(test_target,test_predict)
```

Out[221]:
```
0.842045896538312
```

In [222]:
```
!git
```

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--ba
re]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone             Clone a repository into a new directory
   init              Create an empty Git repository or reinitialize an exis
ting one

work on the current change (see also: git help everyday)
   add               Add file contents to the index
   mv                Move or rename a file, a directory, or a symlink
   restore           Restore working tree files
   rm                Remove files from the working tree and from the index
   sparse-checkout   Initialize and modify the sparse-checkout

examine the history and state (see also: git help revisions)
   bisect            Use binary search to find the commit that introduced a
bug
   diff              Show changes between commits, commit and working tree,
etc
   grep              Print lines matching a pattern
   log               Show commit logs
   show              Show various types of objects
   status            Show the working tree status

grow, mark and tweak your common history
   branch            List, create, or delete branches
   commit            Record changes to the repository
   merge             Join two or more development histories together
   rebase            Reapply commits on top of another base tip
   reset             Reset current HEAD to the specified state
   switch            Switch branches
   tag               Create, list, delete or verify a tag object signed wit
h GPG

collaborate (see also: git help workflows)
   fetch             Download objects and refs from another repository
   pull              Fetch from and integrate with another repository or a
local branch
   push              Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.
```

In [ ]: