

Logistic Regression with Scikit Learn - Machine Learning with Python

This tutorial is a part of [Zero to Data Science Bootcamp by Jovian](#) and [Machine Learning with Python: Zero to GBMs](#)



The following topics are covered in this tutorial:

- Downloading a real-world dataset from Kaggle
- Exploratory data analysis and visualization
- Splitting a dataset into training, validation & test sets
- Filling/imputing missing values in numeric columns
- Scaling numeric features to a $(0, 1)$ range
- Encoding categorical columns as one-hot vectors
- Training a logistic regression model using Scikit-learn
- Evaluating a model using a validation set and test set
- Saving a model to disk and loading it back

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. You will be prompted to connect your Google Drive account so that this notebook can be placed into your drive for execution.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Problem Statement

This tutorial takes a practical and coding-focused approach. We'll learn how to apply *logistic regression* to a real-world dataset from [Kaggle](#):

QUESTION: The [Rain in Australia dataset](#) contains about 10 years of daily weather observations from numerous Australian weather stations. Here's a small sample from the dataset:

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	Cloud3pm	Temp9am	Temp3pm	RainToday	RainTomorrow
Date											
2008-09-21	Melbourne	6.5	19.8	0.4	4.2	10.6	3.0	13.0	19.4	No	No
2009-07-06	Sale	4.9	13.0	0.0	2.0	6.8	6.0	8.6	11.7	No	No
2010-11-20	GoldCoast	18.8	26.4	2.0	NaN	NaN	NaN	24.0	22.1	Yes	No
2010-11-22	PearceRAAF	19.4	27.4	1.8	NaN	10.7	3.0	24.4	25.8	Yes	No
2012-04-26	Nuriootpa	5.1	16.6	0.0	1.4	1.4	7.0	12.1	15.7	No	No
2013-07-06	Sydney	7.8	17.4	0.0	4.2	9.8	0.0	10.2	17.1	No	No
2014-04-22	Perth	7.7	23.7	0.0	4.0	10.5	1.0	16.7	21.8	No	No
2014-06-08	Wollongong	11.1	16.8	0.0	NaN	NaN	1.0	14.0	15.9	No	No
2016-04-13	Sale	10.8	19.0	0.0	NaN	NaN	1.0	16.1	18.1	No	No
2017-04-11	Albany	13.0	NaN	0.0	4.0	NaN	NaN	17.8	NaN	No	NaN

As a data scientist at the Bureau of Meteorology, you are tasked with creating a fully-automated system that can use today's weather data for a given location to predict whether it will rain at the location tomorrow.



Linear Regression vs. Logistic Regression

In the [previous tutorial](#), we attempted to predict a person's annual medical charges using *linear regression*. In this tutorial, we'll use *logistic regression*, which is better suited for *classification* problems like predicting whether it will rain tomorrow. Identifying whether a given problem is a *classification* or *regression* problem is an important first step in machine learning.

Classification Problems

Problems where each input must be assigned a discrete category (also called label or class) are known as *classification problems*.

Here are some examples of classification problems:

- [Rainfall prediction](#): Predicting whether it will rain tomorrow using today's weather data (classes are "Will Rain" and "Will Not Rain")
- [Breast cancer detection](#): Predicting whether a tumor is "benign" (noncancerous) or "malignant" (cancerous) using information like its radius, texture etc.
- [Loan Repayment Prediction](#) - Predicting whether applicants will repay a home loan based on factors like age, income, loan amount, no. of children etc.
- [Handwritten Digit Recognition](#) - Identifying which digit from 0 to 9 a picture of handwritten text represents.

Can you think of some more classification problems?

EXERCISE: Replicate the steps followed in this tutorial with each of the above datasets.

Classification problems can be binary (yes/no) or multiclass (picking one of many classes).

Regression Problems

Problems where a continuous numeric value must be predicted for each input are known as *regression problems*.

Here are some example of regression problems:

- [Medical Charges Prediction](#)
- [House Price Prediction](#)
- [Ocean Temperature Prediction](#)
- [Weather Temperature Prediction](#)

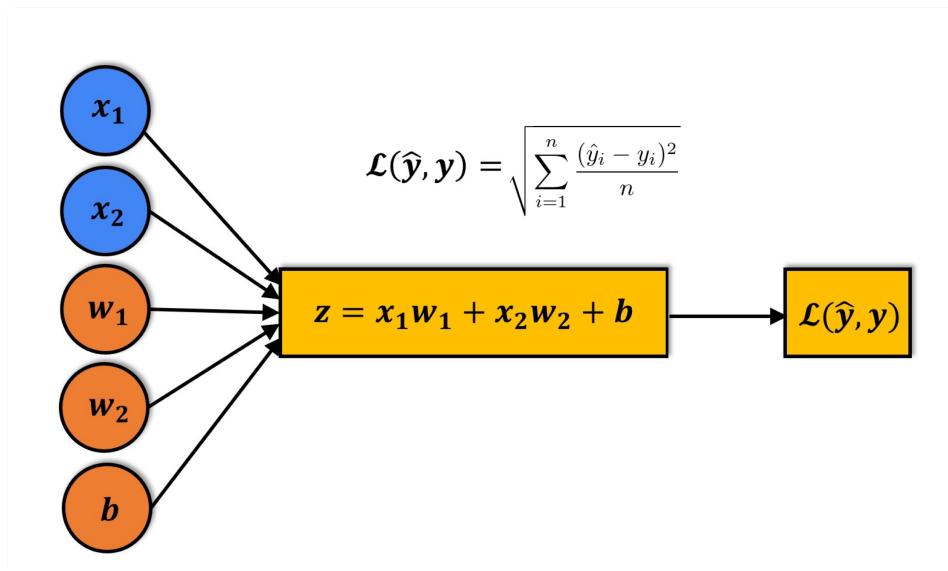
Can you think of some more regression problems?

EXERCISE: Replicate the steps followed in the [previous tutorial](#) with each of the above datasets.

Linear Regression for Solving Regression Problems

Linear regression is a commonly used technique for solving regression problems. In a linear regression model, the target is modeled as a linear combination (or weighted sum) of input features. The predictions from the model are evaluated using a loss function like the Root Mean Squared Error (RMSE).

Here's a visual summary of how a linear regression model is structured:



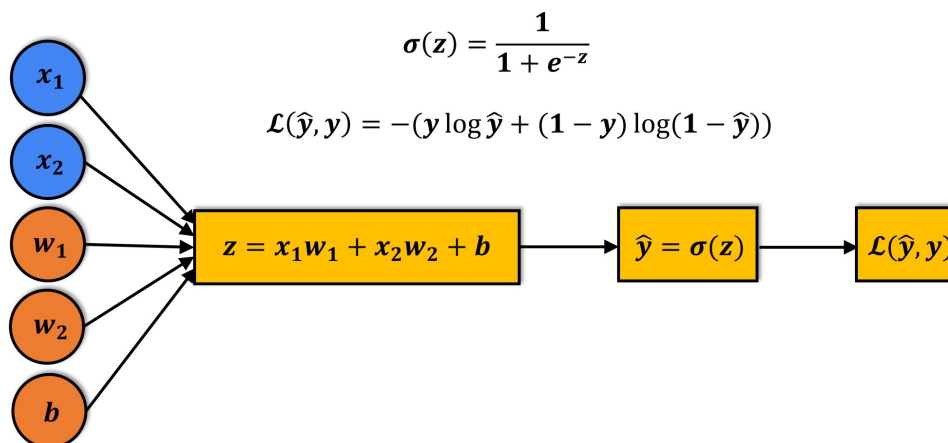
For a mathematical discussion of linear regression, watch [this YouTube playlist](#)

Logistic Regression for Solving Classification Problems

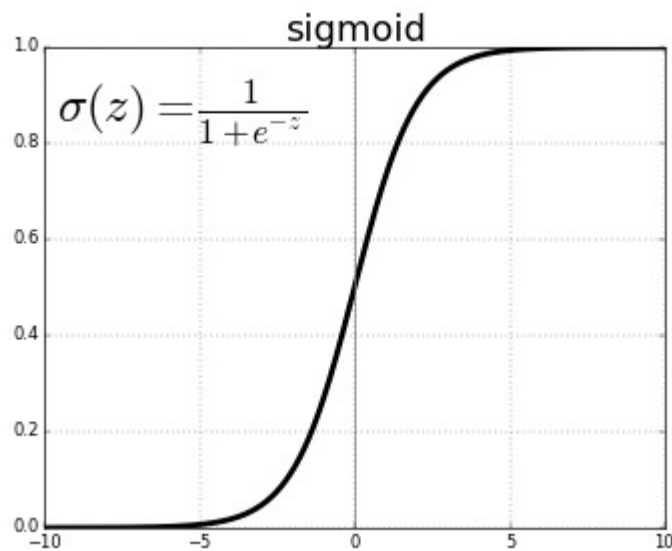
Logistic regression is a commonly used technique for solving binary classification problems. In a logistic regression model:

- we take linear combination (or weighted sum of the input features)
- we apply the sigmoid function to the result to obtain a number between 0 and 1
- this number represents the probability of the input being classified as "Yes"
- instead of RMSE, the cross entropy loss function is used to evaluate the results

Here's a visual summary of how a logistic regression model is structured ([source](#)):



The sigmoid function applied to the linear combination of inputs has the following formula:

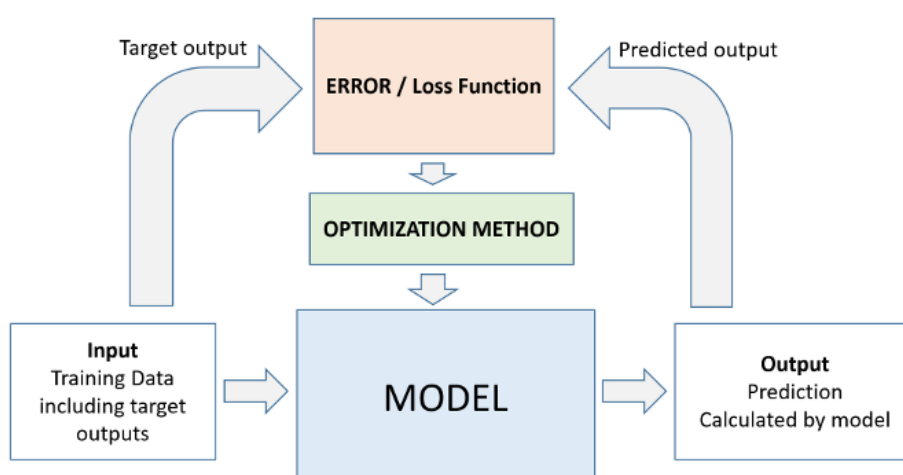


The output of the sigmoid function is called a logistic, hence the name *logistic regression*. For a mathematical discussion of logistic regression, sigmoid activation and cross entropy, check out [this YouTube playlist](#). Logistic regression can also be applied to multi-class classification problems, with a few modifications.

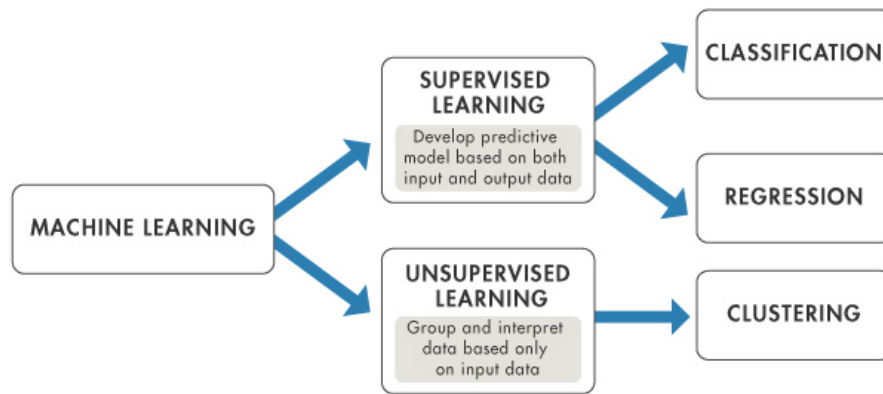
Machine Learning Workflow

Whether we're solving a regression problem using linear regression or a classification problem using logistic regression, the workflow for training a model is exactly the same:

1. We initialize a model with random parameters (weights & biases).
2. We pass some inputs into the model to obtain predictions.
3. We compare the model's predictions with the actual targets using the loss function.
4. We use an optimization technique (like least squares, gradient descent etc.) to reduce the loss by adjusting the weights & biases of the model
5. We repeat steps 1 to 4 till the predictions from the model are good enough.



Classification and regression are both supervised machine learning problems, because they use labeled data. Machine learning applied to unlabeled data is known as unsupervised learning ([image source](#)).



In this tutorial, we'll train a *logistic regression* model using the Rain in Australia dataset to predict whether or not it will rain at a location tomorrow, using today's data. This is a *binary classification* problem.

Let's install the `scikit-learn` library which we'll use to train our model.

```
!pip install scikit-learn --upgrade --quiet
```

Downloading the Data

We'll use the [opendatasets library](#) to download the data from Kaggle directly within Jupyter. Let's install and import `opendatasets`.

```
!pip install opendatasets --upgrade --quiet
```

```
import opendatasets as od
```

```
od.version()
```

```
'0.1.22'
```

The dataset can now be downloaded using `od.download`. When you execute `od.download`, you will be asked to provide your Kaggle username and API key. Follow these instructions to create an API key: <http://bit.ly/kaggle-creds>

```
dataset_url = 'https://www.kaggle.com/jsphyg/weather-dataset-rattle-package'
```

```
od.download(dataset_url)
```

Please provide your Kaggle credentials to download this dataset. Learn more:

<http://bit.ly/kaggle-creds>

Your Kaggle username: `atifahmed23`

Your Kaggle Key: `.....`

Downloading `weather-dataset-rattle-package.zip` to `./weather-dataset-rattle-package`

100%|██████████| 3.83M/3.83M [00:00<00:00, 60.4MB/s]

Once the above command is executed, the dataset is downloaded and extracted to the the directory `weather-dataset-rattle-package` .

```
import os
```

```
data_dir = './weather-dataset-rattle-package'
```

```
os.listdir(data_dir)
```

```
['weatherAUS.csv']
```

```
train_csv = data_dir + '/weatherAUS.csv'
```

Let's load the data from `weatherAUS.csv` using Pandas.

```
!pip install pandas --quiet
```

```
import pandas as pd
```

```
raw_df = pd.read_csv(train_csv)
```

```
raw_df
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	
...	
145455	2017-06-21	Uluru	2.8	23.4	0.0	NaN	NaN	E	31.0	
145456	2017-06-22	Uluru	3.6	25.3	0.0	NaN	NaN	NNW	22.0	
145457	2017-06-23	Uluru	5.4	26.9	0.0	NaN	NaN	N	37.0	
145458	2017-06-24	Uluru	7.8	27.0	0.0	NaN	NaN	SE	28.0	
145459	2017-06-25	Uluru	14.9	NaN	0.0	NaN	NaN	NaN	NaN	

145460 rows × 23 columns

The dataset contains over 145,000 rows and 23 columns. The dataset contains date, numeric and categorical columns. Our objective is to create a model to predict the value in the column `RainTomorrow` .

Let's check the data types and missing values in the various columns.

```
raw_df.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
```

While we should be able to fill in missing values for most columns, it might be a good idea to discard the rows where the value of `RainTomorrow` or `RainToday` is missing to make our analysis and modeling simpler (since one of them is the target variable, and the other is likely to be very closely related to the target variable).

```
raw_df.dropna(subset=['RainToday', 'RainTomorrow'], inplace=True)
```


How would you deal with the missing values in the other columns?

Exploratory Data Analysis and Visualization

Before training a machine learning model, it's always a good idea to explore the distributions of various columns and see how they are related to the target column. Let's explore and visualize the data using the Plotly, Matplotlib and Seaborn libraries. Follow these tutorials to learn how to use these libraries:

- <https://jovian.ai/aakashns/python-matplotlib-data-visualization>
- <https://jovian.ai/aakashns/interactive-visualization-plotly>
- <https://jovian.ai/aakashns/dataviz-cheatsheet>

```
!pip install plotly matplotlib seaborn --quiet
```

```
import plotly.express as px
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (10, 6)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

```
px.histogram(raw_df, x='Location', title='Location vs. Rainy Days', color='RainToday')
```

```
px.histogram(raw_df,  
             x='Temp3pm',  
             title='Temperature at 3 pm vs. Rain Tomorrow',  
             color='RainTomorrow')
```

```
px.histogram(raw_df,  
             x='RainTomorrow',  
             color='RainToday',  
             title='Rain Tomorrow vs. Rain Today')
```

```
px.scatter(raw_df.sample(2000),  
           title='Min Temp. vs Max Temp.',  
           x='MinTemp',  
           y='MaxTemp',  
           color='RainToday')
```

```
px.scatter(raw_df.sample(2000),  
           title='Temp (3 pm) vs. Humidity (3 pm)',
```

```
x='Temp3pm',  
y='Humidity3pm',  
color='RainTomorrow')
```

What interpretations can you draw from the above charts?

Let's save our work before continuing.

```
!pip install jovian --upgrade --quiet
```

```
import jovian
```

(Optional) Working with a Sample

When working with massive datasets containing millions of rows, it's a good idea to work with a sample initially, to quickly set up your model training notebook. If you'd like to work with a sample, just set the value of `use_sample` to `True`.

```
#use_sample = False
```

```
#sample_fraction = 0.1
```

```
#if use_sample:
#     raw_df = raw_df.sample(frac=sample_fraction).copy()
```

Make sure to set `use_sample` to `False` and re-run the notebook end-to-end once you're ready to use the entire dataset.

Training, Validation and Test Sets

While building real-world machine learning models, it is quite common to split the dataset into three parts:

1. **Training set** - used to train the model, i.e., compute the loss and adjust the model's weights using an optimization technique.
2. **Validation set** - used to evaluate the model during training, tune model hyperparameters (optimization technique, regularization etc.), and pick the best version of the model. Picking a good validation set is essential for training models that generalize well. [Learn more here.](#)
3. **Test set** - used to compare different models or approaches and report the model's final accuracy. For many datasets, test sets are provided separately. The test set should reflect the kind of data the model will encounter in the real-world, as closely as feasible.



As a general rule of thumb you can use around 60% of the data for the training set, 20% for the validation set and 20% for the test set. If a separate test set is already provided, you can use a 75%-25% training-validation split.

When rows in the dataset have no inherent order, it's common practice to pick random subsets of rows for creating test and validation sets. This can be done using the `train_test_split` utility from `scikit-learn`. Learn more about it here: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

```
!pip install scikit-learn --upgrade --quiet
```

```
from sklearn.model_selection import train_test_split
```

```
train_val_df, test_df = train_test_split(raw_df, test_size=0.2, random_state=42)
train_df, val_df = train_test_split(train_val_df, test_size=0.25, random_state=42)
```

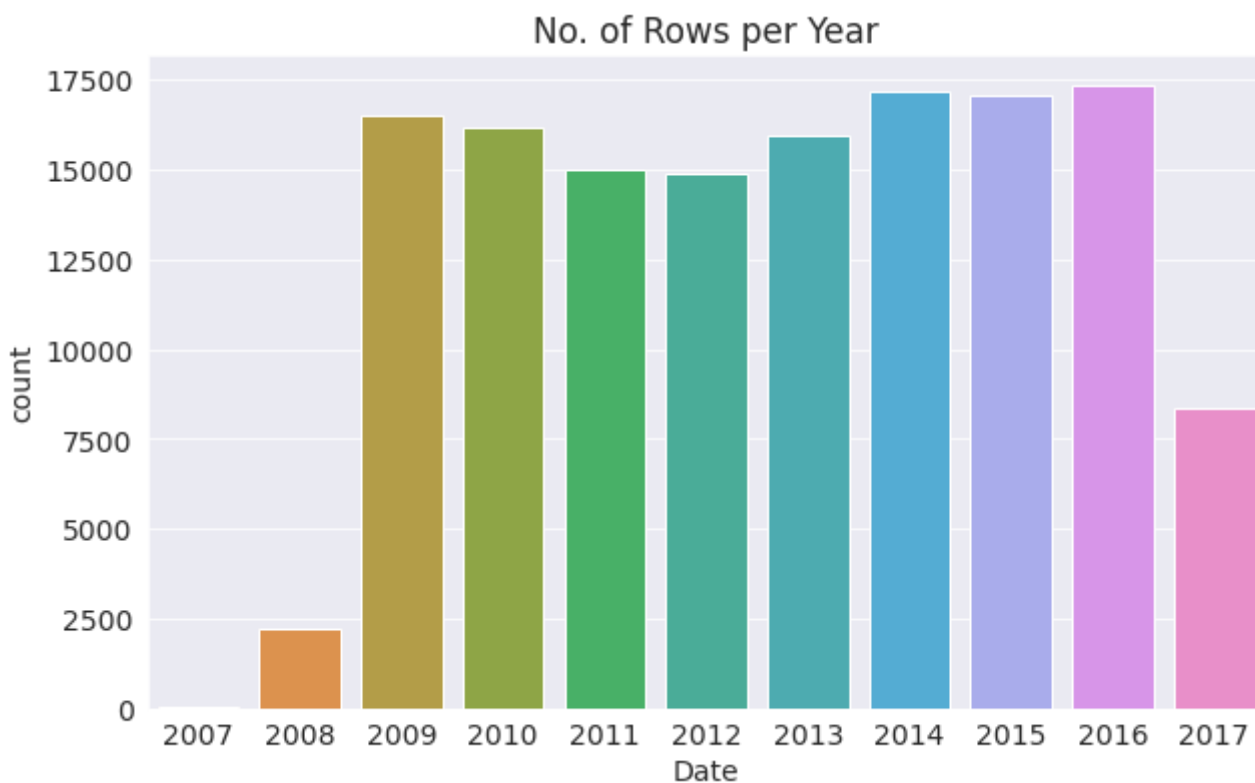
```
print('train_df.shape :', train_df.shape)
print('val_df.shape :', val_df.shape)
print('test_df.shape :', test_df.shape)
```

```
train_df.shape : (84471, 23)
val_df.shape : (28158, 23)
test_df.shape : (28158, 23)
```

However, while working with dates, it's often a better idea to separate the training, validation and test sets with time, so that the model is trained on data from the past and evaluated on data from the future.

For the current dataset, we can use the Date column in the dataset to create another column for year. We'll pick the last two years for the test set, and one year before it for the validation set.

```
plt.title('No. of Rows per Year')
sns.countplot(x=pd.to_datetime(raw_df.Date).dt.year);
```



```
year = pd.to_datetime(raw_df.Date).dt.year
```

```
train_df = raw_df[year < 2015]
val_df = raw_df[year == 2015]
test_df = raw_df[year > 2015]
```

```
print('train_df.shape :', train_df.shape)
print('val_df.shape :', val_df.shape)
print('test_df.shape :', test_df.shape)
```

```
train_df.shape : (97988, 23)
```

```
val_df.shape : (17089, 23)
```

```
test_df.shape : (25710, 23)
```

While not a perfect 60-20-20 split, we have ensured that the test validation and test sets both contain data for all 12 months of the year.

```
train_df
```

Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
------	----------	---------	---------	----------	-------------	----------	-------------	---------------	-------

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	
...	
144548	2014-12-27	Uluru	16.9	33.2	0.0	NaN	NaN	SSE	43.0	
144549	2014-12-28	Uluru	15.1	36.8	0.0	NaN	NaN	NE	31.0	
144550	2014-12-29	Uluru	17.3	37.8	0.0	NaN	NaN	ESE	39.0	
144551	2014-12-30	Uluru	20.1	38.5	0.0	NaN	NaN	ESE	43.0	
144552	2014-12-31	Uluru	22.5	39.6	0.0	NaN	NaN	WNW	76.0	

97988 rows × 23 columns

val_df

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
2133	2015-01-01	Albury	11.4	33.5	0.0	NaN	NaN	WSW	30.0	
2134	2015-01-02	Albury	15.5	39.6	0.0	NaN	NaN	NE	56.0	
2135	2015-01-03	Albury	17.1	38.3	0.0	NaN	NaN	NNE	48.0	
2136	2015-01-04	Albury	26.0	33.1	0.0	NaN	NaN	NNE	41.0	
2137	2015-01-05	Albury	19.0	35.2	0.0	NaN	NaN	E	33.0	
...	
144913	2015-12-27	Uluru	20.5	34.7	0.0	NaN	NaN	E	52.0	
144914	2015-12-28	Uluru	18.0	36.4	0.0	NaN	NaN	ESE	54.0	
144915	2015-12-29	Uluru	17.5	37.1	0.0	NaN	NaN	E	56.0	
144916	2015-12-30	Uluru	20.0	38.9	0.0	NaN	NaN	E	59.0	
144917	2015-12-31	Uluru	19.3	37.4	0.0	NaN	NaN	SE	56.0	

17089 rows × 23 columns

```
test_df
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
2498	2016-01-01	Albury	20.4	37.6	0.0	NaN	NaN	ENE	54.0	
2499	2016-01-02	Albury	20.9	33.6	0.4	NaN	NaN	SSE	50.0	
2500	2016-01-03	Albury	18.4	23.1	2.2	NaN	NaN	ENE	48.0	
2501	2016-01-04	Albury	17.3	23.7	15.6	NaN	NaN	SSE	39.0	
2502	2016-01-05	Albury	15.5	22.9	6.8	NaN	NaN	ENE	31.0	
...	
145454	2017-06-20	Uluru	3.5	21.8	0.0	NaN	NaN	E	31.0	
145455	2017-06-21	Uluru	2.8	23.4	0.0	NaN	NaN	E	31.0	
145456	2017-06-22	Uluru	3.6	25.3	0.0	NaN	NaN	NNW	22.0	
145457	2017-06-23	Uluru	5.4	26.9	0.0	NaN	NaN	N	37.0	
145458	2017-06-24	Uluru	7.8	27.0	0.0	NaN	NaN	SE	28.0	

25710 rows × 23 columns

Let's save our work before continuing.

Identifying Input and Target Columns

Often, not all the columns in a dataset are useful for training a model. In the current dataset, we can ignore the `Date` column, since we only want to weather conditions to make a prediction about whether it will rain the next day.

Let's create a list of input columns, and also identify the target column.

```
input_cols = list(train_df.columns)[1:-1]
target_col = 'RainTomorrow'
```

```
print(input_cols)
```

```
['Location', 'MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine',
'WindGustDir', 'WindGustSpeed', 'WindDir9am', 'WindDir3pm', 'WindSpeed9am',
'WindSpeed3pm', 'Humidity9am', 'Humidity3pm', 'Pressure9am', 'Pressure3pm', 'Cloud9am',
'Cloud3pm', 'Temp9am', 'Temp3pm', 'RainToday']
```

```
target_col
```


'RainTomorrow'

We can now create inputs and targets for the training, validation and test sets for further processing and model training.

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

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

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

train_inputs

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am
0	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	W
1	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	NNW
2	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	W
3	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	SE
4	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	ENE
...
144548	Uluru	16.9	33.2	0.0	NaN	NaN	SSE	43.0	ESE
144549	Uluru	15.1	36.8	0.0	NaN	NaN	NE	31.0	ENE
144550	Uluru	17.3	37.8	0.0	NaN	NaN	ESE	39.0	ESE
144551	Uluru	20.1	38.5	0.0	NaN	NaN	ESE	43.0	ESE
144552	Uluru	22.5	39.6	0.0	NaN	NaN	WNW	76.0	ENE

97988 rows × 21 columns

train_targets

```
0      No
1      No
2      No
3      No
4      No
...
144548  No
144549  No
144550  No
144551  No
144552  No
```

Name: RainTomorrow, Length: 97988, dtype: object

Let's also identify which of the columns are numerical and which ones are categorical. This will be useful later, as we'll need to convert the categorical data to numbers for training a logistic regression model.

```
!pip install numpy --quiet
```

```
import numpy as np
```

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

Let's view some statistics for the numeric columns.

```
train_inputs[numeric_cols].describe()
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am
count	97674.000000	97801.000000	97988.000000	61657.000000	57942.000000	91160.000000	97114.000000
mean	12.007831	23.022202	2.372935	5.289991	7.609004	40.215873	14.092263
std	6.347175	6.984397	8.518819	3.952010	3.788813	13.697967	8.984203
min	-8.500000	-4.100000	0.000000	0.000000	0.000000	6.000000	0.000000
25%	7.500000	17.900000	0.000000	2.600000	4.800000	31.000000	7.000000
50%	11.800000	22.400000	0.000000	4.600000	8.500000	39.000000	13.000000
75%	16.600000	27.900000	0.800000	7.200000	10.600000	48.000000	19.000000
max	33.900000	48.100000	371.000000	82.400000	14.300000	135.000000	87.000000

Do the ranges of the numeric columns seem reasonable? If not, we may have to do some data cleaning as well.

Let's also check the number of categories in each of the categorical columns.

```
train_inputs[categorical_cols].nunique()
```

```
Location      49
WindGustDir    16
WindDir9am     16
WindDir3pm     16
RainToday      2
dtype: int64
```

Let's save our work before continuing.

Imputing Missing Numeric Data

Machine learning models can't work with missing numerical data. The process of filling missing values is called imputation.

	col1	col2	col3	col4	col5			col1	col2	col3	col4	col5	
0	2	5.0	3.0	6	NaN	mean()		0	2.0	5.0	3.0	6.0	7.0
1	9	NaN	9.0	0	7.0			1	9.0	11.0	9.0	0.0	7.0
2	19	17.0	NaN	9	NaN			2	19.0	17.0	6.0	9.0	7.0

There are several techniques for imputation, but we'll use the most basic one: replacing missing values with the average value in the column using the `SimpleImputer` class from `sklearn.impute`.

```
from sklearn.impute import SimpleImputer
```

```
imputer = SimpleImputer(strategy = 'mean')
```

Before we perform imputation, let's check the no. of missing values in each numeric column.

```
raw_df[numeric_cols].isna().sum()
```

```
MinTemp      468
MaxTemp      307
Rainfall      0
Evaporation  59694
Sunshine     66805
WindGustSpeed 9105
WindSpeed9am  1055
WindSpeed3pm  2531
Humidity9am   1517
Humidity3pm   3501
Pressure9am   13743
Pressure3pm   13769
Cloud9am      52625
Cloud3pm      56094
Temp9am       656
Temp3pm      2624
dtype: int64
```

These values are spread across the training, test and validation sets. You can also check the no. of missing values individually for `train_inputs`, `val_inputs` and `test_inputs`.

```
train_inputs[numeric_cols].isna().sum()
```

```
MinTemp      314
MaxTemp      187
Rainfall      0
Evaporation   36331
Sunshine     40046
WindGustSpeed 6828
WindSpeed9am  874
WindSpeed3pm 1069
```

Humidity9am	1052
Humidity3pm	1116
Pressure9am	9112
Pressure3pm	9131
Cloud9am	34988
Cloud3pm	36022
Temp9am	574
Temp3pm	596

dtype: int64

```
val_inputs[numeric_cols].isna().sum()
```

MinTemp	34
MaxTemp	29
Rainfall	0
Evaporation	8335
Sunshine	9038
WindGustSpeed	875
WindSpeed9am	71
WindSpeed3pm	362
Humidity9am	212
Humidity3pm	661
Pressure9am	1977
Pressure3pm	1977
Cloud9am	6506
Cloud3pm	6933
Temp9am	22
Temp3pm	471

dtype: int64

```
test_inputs[numeric_cols].isna().sum()
```

MinTemp	120
MaxTemp	91
Rainfall	0
Evaporation	15028
Sunshine	17721
WindGustSpeed	1402
WindSpeed9am	110
WindSpeed3pm	1100
Humidity9am	253
Humidity3pm	1724
Pressure9am	2654
Pressure3pm	2661
Cloud9am	11131
Cloud3pm	13139
Temp9am	60
Temp3pm	1557

dtype: int64

The first step in imputation is to `fit` the imputer to the data i.e. compute the chosen statistic (e.g. mean) for each column in the dataset.

```
imputer.fit(raw_df[numeric_cols])
```

```
SimpleImputer()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

SimpleImputer

```
SimpleImputer()
```

After calling `fit`, the computed statistic for each column is stored in the `statistics_` property of `imputer`.

```
list(imputer.statistics_)
```

```
[12.18482386562048,  
23.235120301822324,  
2.349974074310839,  
5.472515506887154,  
7.630539861047281,  
39.97051988882308,  
13.990496092519967,  
18.631140782316862,  
68.82683277087672,  
51.44928834695453,  
1017.6545771543717,  
1015.2579625879797,  
4.431160817585808,  
4.499250233195188,  
16.98706638787991,  
21.69318269001107]
```

The missing values in the training, test and validation sets can now be filled in using the `transform` method of `imputer`.

```
train_inputs[numeric_cols] = imputer.transform(train_inputs[numeric_cols])  
val_inputs[numeric_cols] = imputer.transform(val_inputs[numeric_cols])  
test_inputs[numeric_cols] = imputer.transform(test_inputs[numeric_cols])
```

The missing values are now filled in with the mean of each column.

```
train_inputs[numeric_cols].isna().sum()
```

MinTemp	0
MaxTemp	0
Rainfall	0
Evaporation	0

```

Sunshine      0
WindGustSpeed 0
WindSpeed9am  0
WindSpeed3pm  0
Humidity9am   0
Humidity3pm   0
Pressure9am   0
Pressure3pm   0
Cloud9am      0
Cloud3pm      0
Temp9am       0
Temp3pm       0
dtype: int64

```

Scaling Numeric Features

Another good practice is to scale numeric features to a small range of values e.g. $(0, 1)$ or $(-1, 1)$. Scaling numeric features ensures that no particular feature has a disproportionate impact on the model's loss. Optimization algorithms also work better in practice with smaller numbers.

The numeric columns in our dataset have varying ranges.

```
raw_df[numeric_cols].describe()
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am
count	140319.000000	140480.000000	140787.000000	81093.000000	73982.000000	131682.000000	139732.000000
mean	12.184824	23.23512	2.349974	5.472516	7.630540	39.970520	13.990400
std	6.403879	7.11450	8.465173	4.189132	3.781729	13.578201	8.886200
min	-8.500000	-4.80000	0.000000	0.000000	0.000000	6.000000	0.000000
25%	7.600000	17.90000	0.000000	2.600000	4.900000	31.000000	7.000000
50%	12.000000	22.60000	0.000000	4.800000	8.500000	39.000000	13.000000
75%	16.800000	28.30000	0.800000	7.400000	10.700000	48.000000	19.000000
max	33.900000	48.10000	371.000000	145.000000	14.500000	135.000000	130.000000

Let's use `MinMaxScaler` from `sklearn.preprocessing` to scale values to the $(0, 1)$ range.

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
```

First, we `fit` the scaler to the data i.e. compute the range of values for each numeric column.

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

```
MinMaxScaler()
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

MinMaxScaler

```
MinMaxScaler()
```

We can now inspect the minimum and maximum values in each column.

```
print('Minimum:')  
list(scaler.data_min_)
```

Minimum:

```
[-8.5,  
 -4.8,  
 0.0,  
 0.0,  
 0.0,  
 6.0,  
 0.0,  
 0.0,  
 0.0,  
 0.0,  
 980.5,  
 977.1,  
 0.0,  
 0.0,  
 -7.2,  
 -5.4]
```

```
print('Maximum:')  
list(scaler.data_max_)
```

Maximum:

```
[33.9,  
 48.1,  
 371.0,  
 145.0,  
 14.5,  
 135.0,  
 130.0,  
 87.0,  
 100.0,  
 100.0,  
 1041.0,  
 1039.6,  
 9.0,  
 9.0,
```

```
40.2,  
46.7]
```

We can now separately scale the training, validation and test sets using the `transform` method of `scaler`.

```
train_inputs[numeric_cols] = scaler.transform(train_inputs[numeric_cols])  
val_inputs[numeric_cols] = scaler.transform(val_inputs[numeric_cols])  
test_inputs[numeric_cols] = scaler.transform(test_inputs[numeric_cols])
```

We can now verify that values in each column lie in the range (0, 1)


```
train_inputs[numeric_cols].describe()
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am
count	97988.000000	97988.000000	97988.000000	97988.000000	97988.000000	97988.000000	97988.000000
mean	0.483689	0.525947	0.006396	0.036949	0.525366	0.265107	0.108395
std	0.149458	0.131904	0.022962	0.021628	0.200931	0.102420	0.068800
min	0.000000	0.013233	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.377358	0.429112	0.000000	0.026207	0.517241	0.193798	0.053846
50%	0.478774	0.514178	0.000000	0.037741	0.526244	0.255814	0.100000
75%	0.591981	0.618147	0.002156	0.038621	0.634483	0.310078	0.146154
max	1.000000	1.000000	1.000000	0.568276	0.986207	1.000000	0.669231

Learn more about scaling techniques here: <https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/>

Encoding Categorical Data

Since machine learning models can only be trained with numeric data, we need to convert categorical data to numbers. A common technique is to use one-hot encoding for categorical columns.

Index	Categorical column		Index	Cat A	Cat B	Cat C
1	Cat A		1	1	0	0
2	Cat B		2	0	1	0
3	Cat C		3	0	0	1

One hot encoding involves adding a new binary (0/1) column for each unique category of a categorical column.

```
raw_df[categorical_cols].nunique()
```

```
Location      49  
WindGustDir   16  
WindDir9am    16  
WindDir3pm    16
```



```
RainToday      2
dtype: int64
```

We can perform one hot encoding using the `OneHotEncoder` class from `sklearn.preprocessing`.

```
from sklearn.preprocessing import OneHotEncoder
```

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore')
```

First, we `fit` the encoder to the data i.e. identify the full list of categories across all categorical columns.

```
encoder.fit(raw_df[categorical_cols])
```

```
OneHotEncoder(handle_unknown='ignore', sparse=False)
```

In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook.
On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.
`OneHotEncoder`

```
OneHotEncoder(handle_unknown='ignore', sparse=False)
```

```
encoder.categories_
```

```
[array(['Adelaide', 'Albany', 'Albury', 'AliceSprings', 'BadgerysCreek',
       'Ballarat', 'Bendigo', 'Brisbane', 'Cairns', 'Canberra', 'Cobar',
       'CoffsHarbour', 'Dartmoor', 'Darwin', 'GoldCoast', 'Hobart',
       'Katherine', 'Launceston', 'Melbourne', 'MelbourneAirport',
       'Mildura', 'Moree', 'MountGambier', 'MountGinini', 'Newcastle',
       'Nhil', 'NorahHead', 'NorfolkIsland', 'Nuriootpa', 'PearceRAAF',
       'Penrith', 'Perth', 'PerthAirport', 'Portland', 'Richmond', 'Sale',
       'SalmonGums', 'Sydney', 'SydneyAirport', 'Townsville',
       'Tuggeranong', 'Uluru', 'WaggaWagga', 'Walpole', 'Watsonia',
       'Williamtown', 'Witchcliffe', 'Wollongong', 'Woomera'],
      dtype=object),
 array(['E', 'ENE', 'ESE', 'N', 'NE', 'NNE', 'NNW', 'NW', 'S', 'SE', 'SSE',
       'SSW', 'SW', 'W', 'WNW', 'WSW', nan], dtype=object),
 array(['E', 'ENE', 'ESE', 'N', 'NE', 'NNE', 'NNW', 'NW', 'S', 'SE', 'SSE',
       'SSW', 'SW', 'W', 'WNW', 'WSW', nan], dtype=object),
 array(['E', 'ENE', 'ESE', 'N', 'NE', 'NNE', 'NNW', 'NW', 'S', 'SE', 'SSE',
       'SSW', 'SW', 'W', 'WNW', 'WSW', nan], dtype=object),
 array(['No', 'Yes'], dtype=object)]
```

The encoder has created a list of categories for each of the categorical columns in the dataset.

We can generate column names for each individual category using `get_feature_names`.

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

```
[ 'Location_Adelaide', 'Location_Albury', 'Location_AliceSprings',
'Location_BadgerysCreek', 'Location_Ballarat', 'Location_Bendigo', 'Location_Brisbane',
'Location_Cairns', 'Location_Canberra', 'Location_Cobar', 'Location_CoffsHarbour',
'Location_Dartmoor', 'Location_Darwin', 'Location_GoldCoast', 'Location_Hobart',
'Location_Katherine', 'Location_Launceston', 'Location_Melbourne',
'Location_MelbourneAirport', 'Location_Mildura', 'Location_Moree',
'Location_MountGambier', 'Location_MountGinini', 'Location_Newcastle', 'Location_Nhil',
'Location_NorahHead', 'Location_NorfolkIsland', 'Location_Nuriootpa',
'Location_PearceRAAF', 'Location_Penrith', 'Location_Perth', 'Location_PerthAirport',
'Location_Portland', 'Location_Richmond', 'Location_Sale', 'Location_SalmonGums',
'Location_Sydney', 'Location_SydneyAirport', 'Location_Townsville',
'Location_Tuggeranong', 'Location_Uluru', 'Location_WaggaWagga', 'Location_Walpole',
'Location_Watsonia', 'Location_Williamtown', 'Location_Witchcliffe',
'Location_Wollongong', 'Location_Woomera', 'WindGustDir_E', 'WindGustDir_ENE',
'WindGustDir_ESE', 'WindGustDir_N', 'WindGustDir_NE', 'WindGustDir_NNE',
'WindGustDir_NNW', 'WindGustDir_NW', 'WindGustDir_S', 'WindGustDir_SE',
'WindGustDir_SSE', 'WindGustDir_SSW', 'WindGustDir_SW', 'WindGustDir_W',
'WindGustDir_WNW', 'WindGustDir_WSW', 'WindGustDir_nan', 'WindDir9am_E',
'WindDir9am_ENE', 'WindDir9am_ESE', 'WindDir9am_N', 'WindDir9am_NE', 'WindDir9am_NNE',
'WindDir9am_NNW', 'WindDir9am_NW', 'WindDir9am_S', 'WindDir9am_SE', 'WindDir9am_SSE',
'WindDir9am_SSW', 'WindDir9am_SW', 'WindDir9am_W', 'WindDir9am_WNW', 'WindDir9am_WSW',
'WindDir9am_nan', 'WindDir3pm_E', 'WindDir3pm_ENE', 'WindDir3pm_ESE', 'WindDir3pm_N',
'WindDir3pm_NE', 'WindDir3pm_NNE', 'WindDir3pm_NNW', 'WindDir3pm_NW', 'WindDir3pm_S',
'WindDir3pm_SE', 'WindDir3pm_SSE', 'WindDir3pm_SSW', 'WindDir3pm_SW', 'WindDir3pm_W',
'WindDir3pm_WNW', 'WindDir3pm_WSW', 'WindDir3pm_nan', 'RainToday_No', 'RainToday_Yes']
```

/opt/conda/lib/python3.9/site-packages/sklearn/utils/deprecation.py:87: FutureWarning:

Function `get_feature_names` is deprecated; `get_feature_names` is deprecated in 1.0 and will be removed in 1.2. Please use `get_feature_names_out` instead.

All of the above columns will be added to `train_inputs`, `val_inputs` and `test_inputs`.

To perform the encoding, we use the `transform` method of `encoder`.

```
train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols])
val_inputs[encoded_cols] = encoder.transform(val_inputs[categorical_cols])
test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols])
```

/opt/conda/lib/python3.9/site-packages/pandas/core/frame.py:3678: PerformanceWarning:

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

/opt/conda/lib/python3.9/site-packages/pandas/core/frame.py:3678: PerformanceWarning:

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

/opt/conda/lib/python3.9/site-packages/pandas/core/frame.py:3678: PerformanceWarning:

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

We can verify that these new columns have been added to our training, test and validation sets.

```
pd.set_option('display.max_columns', None)
```

test_inputs

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am
2498	Albury	0.681604	0.801512	0.000000	0.037741	0.526244	ENE	0.372093	NaN
2499	Albury	0.693396	0.725898	0.001078	0.037741	0.526244	SSE	0.341085	SSE
2500	Albury	0.634434	0.527410	0.005930	0.037741	0.526244	ENE	0.325581	ESE
2501	Albury	0.608491	0.538752	0.042049	0.037741	0.526244	SSE	0.255814	SE
2502	Albury	0.566038	0.523629	0.018329	0.037741	0.526244	ENE	0.193798	SE
...
145454	Uluru	0.283019	0.502836	0.000000	0.037741	0.526244	E	0.193798	ESE
145455	Uluru	0.266509	0.533081	0.000000	0.037741	0.526244	E	0.193798	SE
145456	Uluru	0.285377	0.568998	0.000000	0.037741	0.526244	NNW	0.124031	SE
145457	Uluru	0.327830	0.599244	0.000000	0.037741	0.526244	N	0.240310	SE
145458	Uluru	0.384434	0.601134	0.000000	0.037741	0.526244	SE	0.170543	SSE

25710 rows × 123 columns

Let's save our work before continuing.

```
jovian.commit()
```

[jovian] Updating notebook "ahmedatif655/python-sklearn-logistic-regression" on <https://jovian.ai>

[jovian] Committed successfully! <https://jovian.ai/ahmedatif655/python-sklearn-logistic-regression>

'<https://jovian.ai/ahmedatif655/python-sklearn-logistic-regression>'

Saving Processed Data to Disk

It can be useful to save processed data to disk, especially for really large datasets, to avoid repeating the preprocessing steps every time you start the Jupyter notebook. The parquet format is a fast and efficient format for saving and loading Pandas dataframes.

```
print('train_inputs:', train_inputs.shape)
print('train_targets:', train_targets.shape)
print('val_inputs:', val_inputs.shape)
print('val_targets:', val_targets.shape)
print('test_inputs:', test_inputs.shape)
print('test_targets:', test_targets.shape)
```

```
train_inputs: (97988, 123)
train_targets: (97988,)
val_inputs: (17089, 123)
val_targets: (17089,)
test_inputs: (25710, 123)
test_targets: (25710,)
```

```
!pip install pyarrow --quiet
```

```
train_inputs.to_parquet('train_inputs.parquet')
val_inputs.to_parquet('val_inputs.parquet')
test_inputs.to_parquet('test_inputs.parquet')
```

```
%%time
pd.DataFrame(train_targets).to_parquet('train_targets.parquet')
pd.DataFrame(val_targets).to_parquet('val_targets.parquet')
pd.DataFrame(test_targets).to_parquet('test_targets.parquet')
```

```
CPU times: user 26.8 ms, sys: 4.95 ms, total: 31.8 ms
Wall time: 30.1 ms
```

We can read the data back using `pd.read_parquet`.

```
%%time

train_inputs = pd.read_parquet('train_inputs.parquet')
val_inputs = pd.read_parquet('val_inputs.parquet')
test_inputs = pd.read_parquet('test_inputs.parquet')

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

```
CPU times: user 282 ms, sys: 162 ms, total: 444 ms
Wall time: 231 ms
```

Let's verify that the data was loaded properly.

```
print('train_inputs:', train_inputs.shape)
print('train_targets:', train_targets.shape)
print('val_inputs:', val_inputs.shape)
print('val_targets:', val_targets.shape)
print('test_inputs:', test_inputs.shape)
print('test_targets:', test_targets.shape)
```

```
train_inputs: (97988, 123)
train_targets: (97988,)
val_inputs: (17089, 123)
val_targets: (17089,)
test_inputs: (25710, 123)
test_targets: (25710,)
```

val_inputs

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am
2133	Albury	0.469340	0.724008	0.0	0.037741	0.526244	WSW	0.186047	ESE
2134	Albury	0.566038	0.839319	0.0	0.037741	0.526244	NE	0.387597	ESE
2135	Albury	0.603774	0.814745	0.0	0.037741	0.526244	NNE	0.325581	NE
2136	Albury	0.813679	0.716446	0.0	0.037741	0.526244	NNE	0.271318	ESE
2137	Albury	0.648585	0.756144	0.0	0.037741	0.526244	E	0.209302	SSE
...
144913	Uluru	0.683962	0.746692	0.0	0.037741	0.526244	E	0.356589	ESE
144914	Uluru	0.625000	0.778828	0.0	0.037741	0.526244	ESE	0.372093	E
144915	Uluru	0.613208	0.792060	0.0	0.037741	0.526244	E	0.387597	E
144916	Uluru	0.672170	0.826087	0.0	0.037741	0.526244	E	0.410853	E
144917	Uluru	0.655660	0.797732	0.0	0.037741	0.526244	SE	0.387597	ESE

17089 rows × 123 columns

val_targets

2133	No
2134	No
2135	No
2136	No
2137	No
...	..
144913	No
144914	No
144915	No
144916	No

144917 No

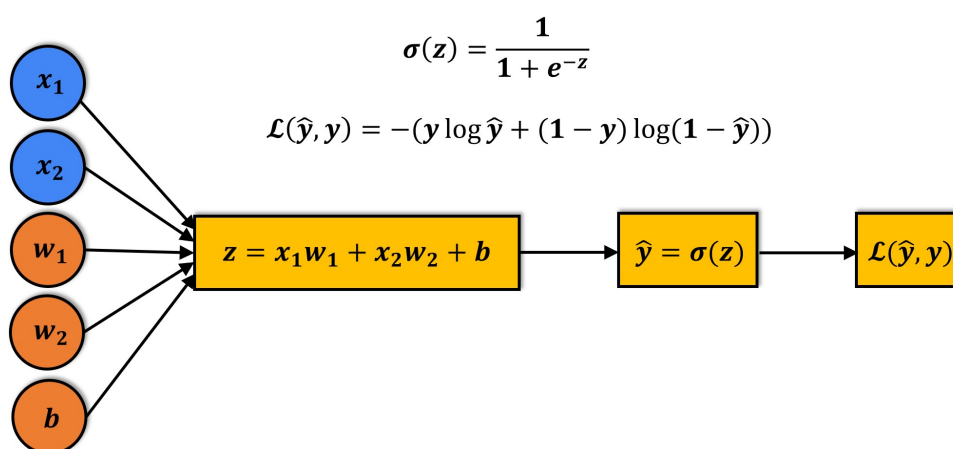
Name: RainTomorrow, Length: 17089, dtype: object

Training a Logistic Regression Model

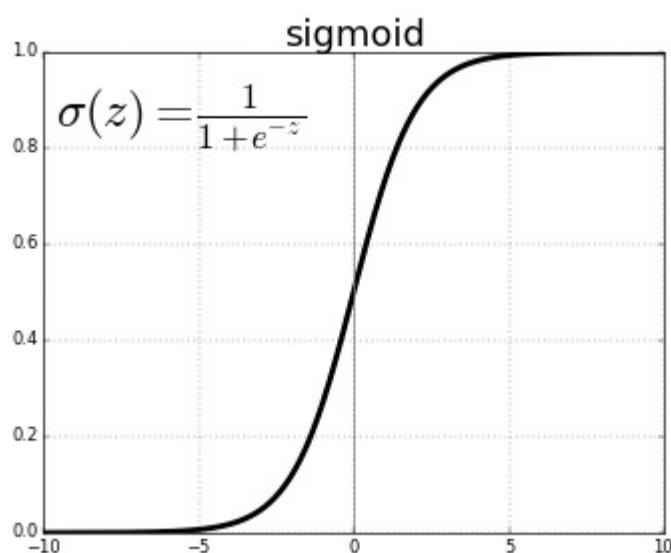
Logistic regression is a commonly used technique for solving binary classification problems. In a logistic regression model:

- we take linear combination (or weighted sum of the input features)
- we apply the sigmoid function to the result to obtain a number between 0 and 1
- this number represents the probability of the input being classified as "Yes"
- instead of RMSE, the cross entropy loss function is used to evaluate the results

Here's a visual summary of how a logistic regression model is structured ([source](#)):



The sigmoid function applied to the linear combination of inputs has the following formula:



To train a logistic regression model, we can use the `LogisticRegression` class from Scikit-learn.

```
from sklearn.linear_model import LogisticRegression
```

```
model = LogisticRegression(solver='liblinear')
```

We can train the model using `model.fit`.

```
model.fit(train_inputs[numeric_cols + encoded_cols], train_targets)
```

```
LogisticRegression(solver='liblinear')
```

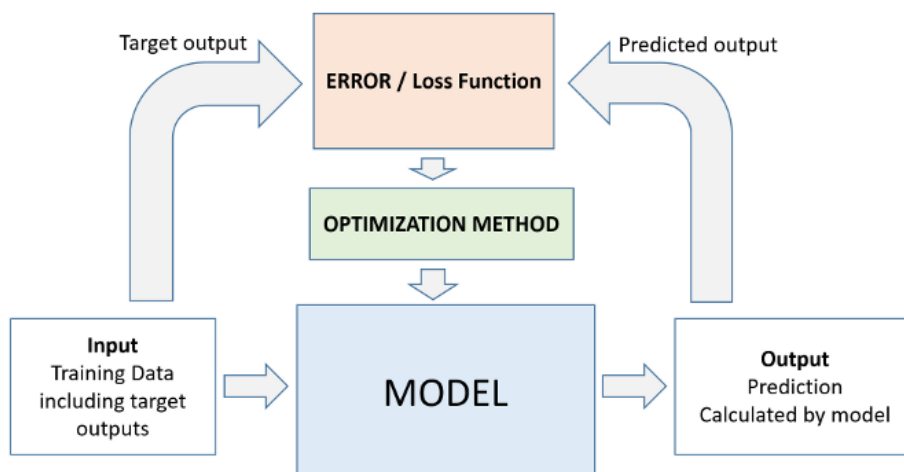
In a Jupyter environment, please rerun this cell to show the HTML representation or trust the notebook. On GitHub, the HTML representation is unable to render, please try loading this page with nbviewer.org.

LogisticRegression

```
LogisticRegression(solver='liblinear')
```

`model.fit` uses the following workflow for training the model ([source](#)):

1. We initialize a model with random parameters (weights & biases).
2. We pass some inputs into the model to obtain predictions.
3. We compare the model's predictions with the actual targets using the loss function.
4. We use an optimization technique (like least squares, gradient descent etc.) to reduce the loss by adjusting the weights & biases of the model
5. We repeat steps 1 to 4 till the predictions from the model are good enough.



For a mathematical discussion of logistic regression, sigmoid activation and cross entropy, check out [this YouTube playlist](#). Logistic regression can also be applied to multi-class classification problems, with a few modifications.

Let's check the weights and biases of the trained model.

```
print(numeric_cols + encoded_cols)
```

```
['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine', 'WindGustSpeed',  
'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm', 'Pressure9am',  
'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am', 'Temp3pm', 'Location_Adelaide',  
'Location_Albury', 'Location_AliceSprings',  
'Location_BadgerysCreek', 'Location_Ballarat', 'Location_Bendigo', 'Location_Brisbane',  
'Location_Cairns', 'Location_Canberra', 'Location_Cobar', 'Location_CoffsHarbour',  
'Location_Dartmoor', 'Location_Darwin', 'Location_GoldCoast', 'Location_Hobart',
```

```
'Location_Katherine', 'Location_Launceston', 'Location_Melbourne',
'Location_MelbourneAirport', 'Location_Mildura', 'Location_Moree',
'Location_MountGambier', 'Location_MountGinini', 'Location_Newcastle', 'Location_Nhil',
'Location_NorahHead', 'Location_NorfolkIsland', 'Location_Nuriootpa',
'Location_PearceRAAF', 'Location_Penrith', 'Location_Perth', 'Location_PerthAirport',
'Location_Portland', 'Location_Richmond', 'Location_Sale', 'Location_SalmonGums',
'Location_Sydney', 'Location_SydneyAirport', 'Location_Townsville',
'Location_Tuggeranong', 'Location_Uluru', 'Location_WaggaWagga', 'Location_Walpole',
'Location_Watsonia', 'Location_Williamtown', 'Location_Witchcliffe',
'Location_Wollongong', 'Location_Woomera', 'WindGustDir_E', 'WindGustDir_ENE',
'WindGustDir_ESE', 'WindGustDir_N', 'WindGustDir_NE', 'WindGustDir_NNE',
'WindGustDir_NNW', 'WindGustDir_NW', 'WindGustDir_S', 'WindGustDir_SE',
'WindGustDir_SSE', 'WindGustDir_SSW', 'WindGustDir_SW', 'WindGustDir_W',
'WindGustDir_WNW', 'WindGustDir_WSW', 'WindGustDir_nan', 'WindDir9am_E',
'WindDir9am_ENE', 'WindDir9am_ESE', 'WindDir9am_N', 'WindDir9am_NE', 'WindDir9am_NNE',
'WindDir9am_NNW', 'WindDir9am_NW', 'WindDir9am_S', 'WindDir9am_SE', 'WindDir9am_SSE',
'WindDir9am_SSW', 'WindDir9am_SW', 'WindDir9am_W', 'WindDir9am_WNW', 'WindDir9am_WSW',
'WindDir9am_nan', 'WindDir3pm_E', 'WindDir3pm_ENE', 'WindDir3pm_ESE', 'WindDir3pm_N',
'WindDir3pm_NE', 'WindDir3pm_NNE', 'WindDir3pm_NNW', 'WindDir3pm_NW', 'WindDir3pm_S',
'WindDir3pm_SE', 'WindDir3pm_SSE', 'WindDir3pm_SSW', 'WindDir3pm_SW', 'WindDir3pm_W',
'WindDir3pm_WNW', 'WindDir3pm_WSW', 'WindDir3pm_nan', 'RainToday_No', 'RainToday_Yes']
```

```
print(model.coef_.tolist())
```

```
[0.8986312609474285, -2.879915092658212, 3.1627776264933574, 0.8542459382973824,
-1.6713938157543706, 6.76440050922128, -0.9423213591718845, -1.42842865463832,
0.3228930743770261, 5.9953154987618165, 5.46385655502931, -9.176804472374327,
-0.16229721680217385, 1.2876597708076856, 0.4747154320042143, 2.02142997364059,
0.6016502699240072, -0.5524818275855837, 0.47814286015408314, 0.007669467204548213,
0.3468145806348842, -0.35227732307979903, 0.1797107840572217, 0.44048649804106504,
-0.01398167816568222, 0.02894383291420655, 0.2581473534927079, -0.02120563206261895,
-0.04279484239836155, -0.4831416037756491, -0.1375633826357322, -0.5760597487717791,
-0.7875230536596454, -0.25540396702719076, -0.3288841032379834, -0.5690045891451422,
0.08183001655827722, 0.013382478857057546, 0.06412743460364478, -0.9020545807835312,
-0.44432966651080874, 0.008517224362121965, -0.4606123862367159, -0.46551812525644326,
-0.06949902988832511, 0.19115875431972928, 0.4504759478826444, 0.6081210437708318,
0.42731381800032103, -0.02833128317486651, 0.2515468486227883, -0.3216057062471844,
0.4249559063158988, -0.059037309568432346, -0.11320039897732705, -0.7283770160157691,
0.36645271606590557, 0.18359037024685046, 0.18397509471495396, 0.186604155984671,
-0.24926950217802196, 0.01794813204026981, 0.7034029609707069, -0.8000214863101158,
-0.1923442975471845, -0.16179791354781203, -0.15898620264542218, -0.06310033590951818,
-0.2235507857968916, -0.23239291679777038, -0.3211527410742425, -0.16556887395871991,
-0.1588490215452739, -0.11097605966472121, -0.03252737030424638, -0.05062769301399886,
-0.09419907134852218, -0.0908177872772501, -0.22315549084119995, -0.25852836372215743,
```



```
-0.20081553189918253, 0.09749216884851723, -0.3187737020817201, -0.0293565743180952,  
-0.3286137550314391, 0.029865254960669677, -0.02106594187179869, 0.14405292856066781,  
-0.06176893659197386, -0.056974532765033455, -0.4085360588512133, -0.30812356188439455,  
-0.4054348438541035, -0.19482956984204197, -0.060291451422106714, -0.08692075188432295,  
-0.05711258488208302, -0.01612934921555332, -0.26954055952487277, -0.23985373450037073,  
-0.15250075425147527, -0.23483410396076096, 0.03819649909879108, -0.31141324929030056,  
-0.08024483340788163, 0.2845459191687802, 0.2218508419248476, -0.2654491111515652,  
-0.2411741260412407, -0.3668808289837676, -0.31686271657059184, -0.3700217827404518,  
-0.18037040623755315, -0.03349904762557947, -0.2759756376242295, 0.07493308169351313,  
-1.4735166839848708, -0.9760373065076474]]
```

```
print(model.intercept_)
```

```
[-2.44955399]
```

Each weight is applied to the value in a specific column of the input. Higher the weight, greater the impact of the column on the prediction.

Making Predictions and Evaluating the Model

We can now use the trained model to make predictions on the training, test

```
X_train = train_inputs[numeric_cols + encoded_cols]  
X_val = val_inputs[numeric_cols + encoded_cols]  
X_test = test_inputs[numeric_cols + encoded_cols]
```

```
train_preds = model.predict(X_train)
```

```
train_preds
```

```
train_targets
```

We can output a probabilistic prediction using `predict_proba`.

```
train_probs = model.predict_proba(X_train)  
train_probs
```

The numbers above indicate the probabilities for the target classes "No" and "Yes".

```
model.classes_
```

We can test the accuracy of the model's predictions by computing the percentage of matching values in `train_preds` and `train_targets`.

This can be done using the `accuracy_score` function from `sklearn.metrics`.

```
from sklearn.metrics import accuracy_score
```

```
accuracy_score(train_targets, train_preds)
```

The model achieves an accuracy of 85.1% on the training set. We can visualize the breakdown of correctly and incorrectly classified inputs using a confusion matrix.

		Predicted	
		Negative (N) -	Positive (P) +
Actual	Negative -	True Negatives (TN)	False Positives (FP) Type I error
	Positive +	False Negatives (FN) Type II error	True Positives (TP)

```
from sklearn.metrics import confusion_matrix
```

```
confusion_matrix(train_targets, train_preds, normalize='true')
```

Let's define a helper function to generate predictions, compute the accuracy score and plot a confusion matrix for a given set of inputs.

```
def predict_and_plot(inputs, targets, name=''):
    preds = model.predict(inputs)

    accuracy = accuracy_score(targets, preds)
    print("Accuracy: {:.2f}%".format(accuracy * 100))

    cf = confusion_matrix(targets, preds, normalize='true')
    plt.figure()
    sns.heatmap(cf, annot=True)
    plt.xlabel('Prediction')
    plt.ylabel('Target')
    plt.title('{} Confusion Matrix'.format(name));

    return preds
```

```
train_preds = predict_and_plot(X_train, train_targets, 'Training')
```

Let's compute the model's accuracy on the validation and test sets too.

```
val_preds = predict_and_plot(X_val, val_targets, 'Validatiaon')
```

```
test_preds = predict_and_plot(X_test, test_targets, 'Test')
```

The accuracy of the model on the test and validation set are above 84%, which suggests that our model generalizes well to data it hasn't seen before.

But how good is 84% accuracy? While this depends on the nature of the problem and on business requirements, a good way to verify whether a model has actually learned something useful is to compare its results to a "random" or "dumb" model.

Let's create two models: one that guesses randomly and another that always return "No". Both of these models completely ignore the inputs given to them.

```
def random_guess(inputs):  
    return np.random.choice(["No", "Yes"], len(inputs))
```

```
def all_no(inputs):  
    return np.full(len(inputs), "No")
```

Let's check the accuracies of these two models on the test set.

```
accuracy_score(test_targets, random_guess(X_test))
```

```
accuracy_score(test_targets, all_no(X_test))
```

Our random model achieves an accuracy of 50% and our "always No" model achieves an accuracy of 77%.

Thankfully, our model is better than a "dumb" or "random" model! This is not always the case, so it's a good practice to benchmark any model you train against such baseline models.

EXERCISE: Initialize the `LogisticRegression` model with different arguments and try to achieve a higher accuracy. The arguments used for initializing the model are called hyperparameters (to differentiate them from weights and biases - parameters that are learned by the model during training). You can find the full list of arguments here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

EXERCISE: Train a logistic regression model using just the numeric columns from the dataset. Does it perform better or worse than the model trained above?

EXERCISE: Train a logistic regression model using just the categorical columns from the dataset. Does it perform better or worse than the model trained above?

EXERCISE: Train a logistic regression model without feature scaling. Also try a different strategy for missing data imputation. Does it perform better or worse than the model trained above?

Let's save our work before continuing.

```
jovian.commit()
```

Making Predictions on a Single Input

Once the model has been trained to a satisfactory accuracy, it can be used to make predictions on new data. Consider the following dictionary containing data collected from the Katherine weather department today.

```
new_input = {'Date': '2021-06-19',
             'Location': 'Katherine',
             'MinTemp': 23.2,
             'MaxTemp': 33.2,
             'Rainfall': 10.2,
             'Evaporation': 4.2,
             'Sunshine': np.nan,
             'WindGustDir': 'NNW',
             'WindGustSpeed': 52.0,
             'WindDir9am': 'NW',
             'WindDir3pm': 'NNE',
             'WindSpeed9am': 13.0,
             'WindSpeed3pm': 20.0,
             'Humidity9am': 89.0,
             'Humidity3pm': 58.0,
```

```
'Pressure9am': 1004.8,  
'Pressure3pm': 1001.5,  
'Cloud9am': 8.0,  
'Cloud3pm': 5.0,  
'Temp9am': 25.7,  
'Temp3pm': 33.0,  
'RainToday': 'Yes'}
```

The first step is to convert the dictionary into a Pandas dataframe, similar to `raw_df` . This can be done by passing a list containing the given dictionary to the `pd.DataFrame` constructor.

```
new_input_df = pd.DataFrame([new_input])
```

```
new_input_df
```

We've now created a Pandas dataframe with the same columns as `raw_df` (except `RainTomorrow` , which needs to be predicted). The dataframe contains just one row of data, containing the given input.

We must now apply the same transformations applied while training the model:

1. Imputation of missing values using the `imputer` created earlier
2. Scaling numerical features using the `scaler` created earlier
3. Encoding categorical features using the `encoder` created earlier

```
new_input_df[numeric_cols] = imputer.transform(new_input_df[numeric_cols])  
new_input_df[numeric_cols] = scaler.transform(new_input_df[numeric_cols])  
new_input_df[encoded_cols] = encoder.transform(new_input_df[categorical_cols])
```

```
X_new_input = new_input_df[numeric_cols + encoded_cols]  
X_new_input
```

We can now make a prediction using `model.predict` .

```
prediction = model.predict(X_new_input)[0]
```

```
prediction
```

Our model predicts that it will rain tomorrow in Katherine! We can also check the probability of the prediction.

```
prob = model.predict_proba(X_new_input)[0]
```

```
prob
```

Looks like our model isn't too confident about its prediction!

Let's define a helper function to make predictions for individual inputs.

```
def predict_input(single_input):
    input_df = pd.DataFrame([single_input])
    input_df[numeric_cols] = imputer.transform(input_df[numeric_cols])
    input_df[numeric_cols] = scaler.transform(input_df[numeric_cols])
    input_df[encoded_cols] = encoder.transform(input_df[categorical_cols])
    X_input = input_df[numeric_cols + encoded_cols]
    pred = model.predict(X_input)[0]
    prob = model.predict_proba(X_input)[0][list(model.classes_).index(pred)]
    return pred, prob
```

We can now use this function to make predictions for individual inputs.

```
new_input = {'Date': '2021-06-19',
             'Location': 'Launceston',
             'MinTemp': 23.2,
             'MaxTemp': 33.2,
             'Rainfall': 10.2,
             'Evaporation': 4.2,
             'Sunshine': np.nan,
             'WindGustDir': 'NNW',
             'WindGustSpeed': 52.0,
             'WindDir9am': 'NW',
             'WindDir3pm': 'NNE',
             'WindSpeed9am': 13.0,
             'WindSpeed3pm': 20.0,
             'Humidity9am': 89.0,
             'Humidity3pm': 58.0,
             'Pressure9am': 1004.8,
             'Pressure3pm': 1001.5,
             'Cloud9am': 8.0,
             'Cloud3pm': 5.0,
             'Temp9am': 25.7,
             'Temp3pm': 33.0,
             'RainToday': 'Yes'}
```

```
predict_input(new_input)
```

EXERCISE: Try changing the values in `new_input` and observe how the predictions and probabilities change. Try different values of location, temperature, humidity, pressure etc. Try to get an *intuitive feel* of which columns have the greatest effect on the result of the model.

```
raw_df.Location.unique()
```

Let's save our work before continuing.

```
jovian.commit()
```

Saving and Loading Trained Models

We can save the parameters (weights and biases) of our trained model to disk, so that we needn't retrain the model from scratch each time we wish to use it. Along with the model, it's also important to save imputers, scalers, encoders and even column names. Anything that will be required while generating predictions using the model should be saved.

We can use the `joblib` module to save and load Python objects on the disk.

```
import joblib
```

Let's first create a dictionary containing all the required objects.

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

We can now save this to a file using `joblib.dump`

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

The object can be loaded back using `joblib.load`

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

Let's use the loaded model to make predictions on the original test set.

```
test_preds2 = aussie_rain2['model'].predict(X_test)  
accuracy_score(test_targets, test_preds2)
```

As expected, we get the same result as the original model.

Let's save our work before continuing. We can upload our trained models to Jovian using the `outputs` argument.

```
jovian.commit(outputs=['aussie_rain.joblib'])
```

Putting it all Together

While we've covered a lot of ground in this tutorial, the number of lines of code for processing the data and training the model is fairly small. Each step requires no more than 3-4 lines of code.

Data Preprocessing

```
import opendatasets as od
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder

# Download the dataset
od.download('https://www.kaggle.com/jsphyg/weather-dataset-rattle-package')
raw_df = pd.read_csv('weather-dataset-rattle-package/weatherAUS.csv')
raw_df.dropna(subset=['RainToday', 'RainTomorrow'], inplace=True)

# Create training, validation and test sets
year = pd.to_datetime(raw_df.Date).dt.year
train_df, val_df, test_df = raw_df[year < 2015], raw_df[year == 2015], raw_df[year > 2015]

# Create inputs and targets
input_cols = list(train_df.columns)[1:-1]
target_col = 'RainTomorrow'
train_inputs, train_targets = train_df[input_cols].copy(), train_df[target_col].copy()
val_inputs, val_targets = val_df[input_cols].copy(), val_df[target_col].copy()
test_inputs, test_targets = test_df[input_cols].copy(), test_df[target_col].copy()

# Identify numeric and categorical columns
numeric_cols = train_inputs.select_dtypes(include=np.number).columns.tolist()[:-1]
categorical_cols = train_inputs.select_dtypes('object').columns.tolist()

# Impute missing numerical values
imputer = SimpleImputer(strategy = 'mean').fit(raw_df[numeric_cols])
train_inputs[numeric_cols] = imputer.transform(train_inputs[numeric_cols])
val_inputs[numeric_cols] = imputer.transform(val_inputs[numeric_cols])
test_inputs[numeric_cols] = imputer.transform(test_inputs[numeric_cols])

# Scale numeric features
scaler = MinMaxScaler().fit(raw_df[numeric_cols])
train_inputs[numeric_cols] = scaler.transform(train_inputs[numeric_cols])
val_inputs[numeric_cols] = scaler.transform(val_inputs[numeric_cols])
test_inputs[numeric_cols] = scaler.transform(test_inputs[numeric_cols])
```



```
# One-hot encode categorical features
```

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore').fit(raw_df[categorical_cols])
encoded_cols = list(encoder.get_feature_names(categorical_cols))
train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols])
val_inputs[encoded_cols] = encoder.transform(val_inputs[categorical_cols])
test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols])
```

```
# Save processed data to disk
```

```
train_inputs.to_parquet('train_inputs.parquet')
val_inputs.to_parquet('val_inputs.parquet')
test_inputs.to_parquet('test_inputs.parquet')
pd.DataFrame(train_targets).to_parquet('train_targets.parquet')
pd.DataFrame(val_targets).to_parquet('val_targets.parquet')
pd.DataFrame(test_targets).to_parquet('test_targets.parquet')
```

```
# Load processed data from disk
```

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

EXERCISE: Try to explain each line of code in the above cell in your own words. Scroll back to relevant sections of the notebook if needed.

Model Training and Evaluation

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import joblib
```

```
# Select the columns to be used for training/prediction
```

```
X_train = train_inputs[numeric_cols + encoded_cols]
X_val = val_inputs[numeric_cols + encoded_cols]
X_test = test_inputs[numeric_cols + encoded_cols]
```

```
# Create and train the model
```

```
model = LogisticRegression(solver='liblinear')
model.fit(X_train, train_targets)
```

```
# Generate predictions and probabilities
```

```
train_preds = model.predict(X_train)
train_probs = model.predict_proba(X_train)
accuracy_score(train_targets, train_preds)
```

```
# Helper function to predict, compute accuracy & plot confusion matrix
```

```
def predict_and_plot(inputs, targets, name=''):
    preds = model.predict(inputs)
```

```

accuracy = accuracy_score(targets, preds)
print("Accuracy: {:.2f}%".format(accuracy * 100))
cf = confusion_matrix(targets, preds, normalize='true')
plt.figure()
sns.heatmap(cf, annot=True)
plt.xlabel('Prediction')
plt.ylabel('Target')
plt.title('{} Confusion Matrix'.format(name));
return preds

# Evaluate on validation and test set
val_preds = predict_and_plot(X_val, val_targets, 'Validation')
test_preds = predict_and_plot(X_test, test_targets, 'Test')

# Save the trained model & load it back
aussie_rain = {'model': model, 'imputer': imputer, 'scaler': scaler, 'encoder': encoder,
               'input_cols': input_cols, 'target_col': target_col, 'numeric_cols': numeric_cols,
               'categorical_cols': categorical_cols, 'encoded_cols': encoded_cols}
joblib.dump(aussie_rain, 'aussie_rain.joblib')
aussie_rain2 = joblib.load('aussie_rain.joblib')

```

EXERCISE: Try to explain each line of code in the above cell in your own words. Scroll back to relevant sections of the notebook if needed.

Prediction on Single Inputs

```

def predict_input(single_input):
    input_df = pd.DataFrame([single_input])
    input_df[numeric_cols] = imputer.transform(input_df[numeric_cols])
    input_df[numeric_cols] = scaler.transform(input_df[numeric_cols])
    input_df[encoded_cols] = encoder.transform(input_df[categorical_cols])
    X_input = input_df[numeric_cols + encoded_cols]
    pred = model.predict(X_input)[0]
    prob = model.predict_proba(X_input)[0][list(model.classes_).index(pred)]
    return pred, prob

new_input = {'Date': '2021-06-19',
              'Location': 'Launceston',
              'MinTemp': 23.2,
              'MaxTemp': 33.2,
              'Rainfall': 10.2,
              'Evaporation': 4.2,
              'Sunshine': np.nan,
              'WindGustDir': 'NNW',
              'WindGustSpeed': 52.0,
              'WindDir9am': 'NW',
              'WindDir3pm': 'NNE',
              'WindSpeed9am': 13.0,
              'WindSpeed3pm': 20.0,

```

```
'Humidity9am': 89.0,  
'Humidity3pm': 58.0,  
'Pressure9am': 1004.8,  
'Pressure3pm': 1001.5,  
'Cloud9am': 8.0,  
'Cloud3pm': 5.0,  
'Temp9am': 25.7,  
'Temp3pm': 33.0,  
'RainToday': 'Yes'}
```

```
predict_input(new_input)
```

Let's save our work using Jovian.

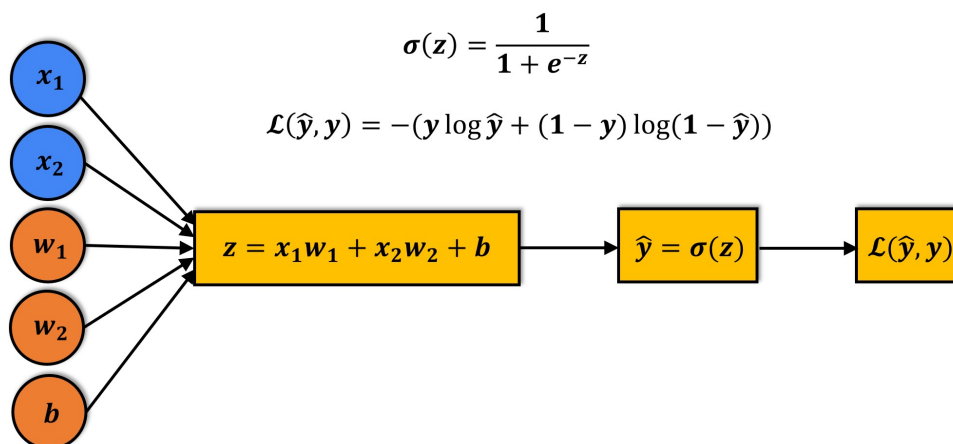
```
jovian.commit()
```

Summary and References

Logistic regression is a commonly used technique for solving binary classification problems. In a logistic regression model:

- we take linear combination (or weighted sum of the input features)
- we apply the sigmoid function to the result to obtain a number between 0 and 1
- this number represents the probability of the input being classified as "Yes"
- instead of RMSE, the cross entropy loss function is used to evaluate the results

Here's a visual summary of how a logistic regression model is structured ([source](#)):



To train a logistic regression model, we can use the `LogisticRegression` class from Scikit-learn. We covered the following topics in this tutorial:

- Downloading a real-world dataset from Kaggle
- Exploratory data analysis and visualization
- Splitting a dataset into training, validation & test sets
- Filling/imputing missing values in numeric columns
- Scaling numeric features to a (0, 1) range
- Encoding categorical columns as one-hot vectors

- Training a logistic regression model using Scikit-learn
- Evaluating a model using a validation set and test set
- Saving a model to disk and loading it back

Check out the following resources to learn more:

- <https://www.youtube.com/watch?v=-la3q9d7AKQ&list=PLNeKWBMsAzboR8vvhnlanxCNr2V7ITuXy&index=1>
- <https://www.kaggle.com/prashant111/extensive-analysis-eda-fe-modelling>
- <https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction#Baseline>
- <https://jovian.ai/aakashns/03-logistic-regression>

Try training logistic regression models on the following datasets:

- [Breast cancer detection](#): Predicting whether a tumor is "benign" (noncancerous) or "malignant" (cancerous) using information like its radius, texture etc.
- [Loan Repayment Prediction](#) - Predicting whether applicants will repay a home loan based on factors like age, income, loan amount, no. of children etc.
- [Handwritten Digit Recognition](#) - Identifying which digit from 0 to 9 a picture of handwritten text represents.