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:
>
> | Date | Location | MinTemp | MaxTemp | Rainfall | Evaporation | Sunshine | Cloud3pm | Temp9am | Temp3pm | RainToday | RainTomorrow |
> |---|---|---|---|---|---|---|---|---|---|---|---|
> | 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.
>
> 
>

EXERCISE: Before proceeding further, take a moment to think about how you can approach this problem. List five or more ideas that come to your mind below:

1. ???
2. ???
3. ???
4. ???
5. ???

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 *classfication* 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]

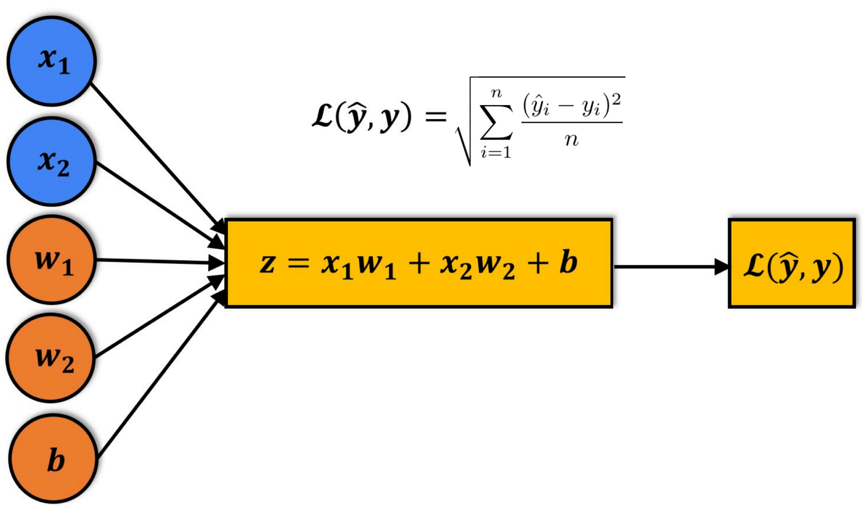
Can you think of some more regression problems?

> **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:



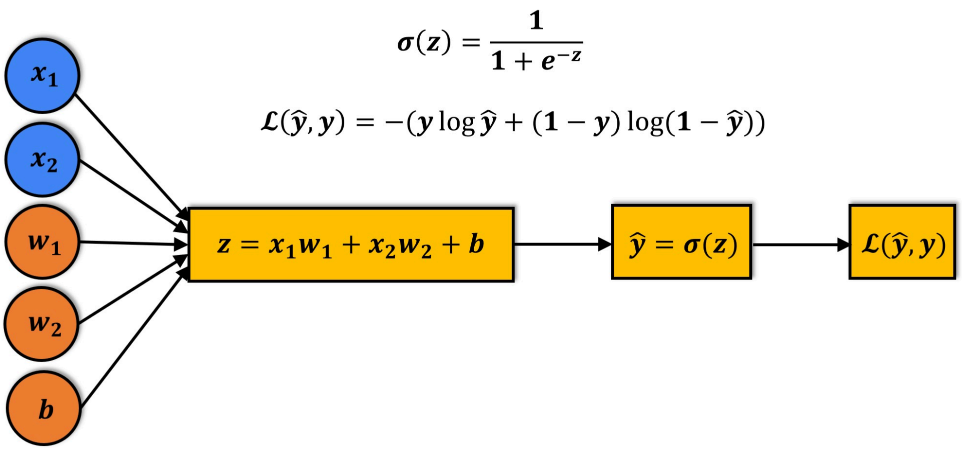
For a mathematical discussion of linear regression, watch [this YouTube playlist]

Logistic Regression for Solving Classification Problems

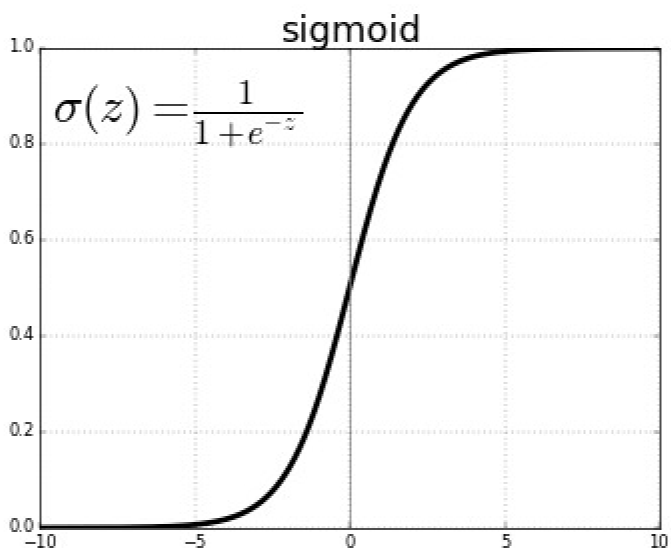
Logistic regression is a commonly used technique for solving binary classification problems. In a logistic regression model:

- 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:

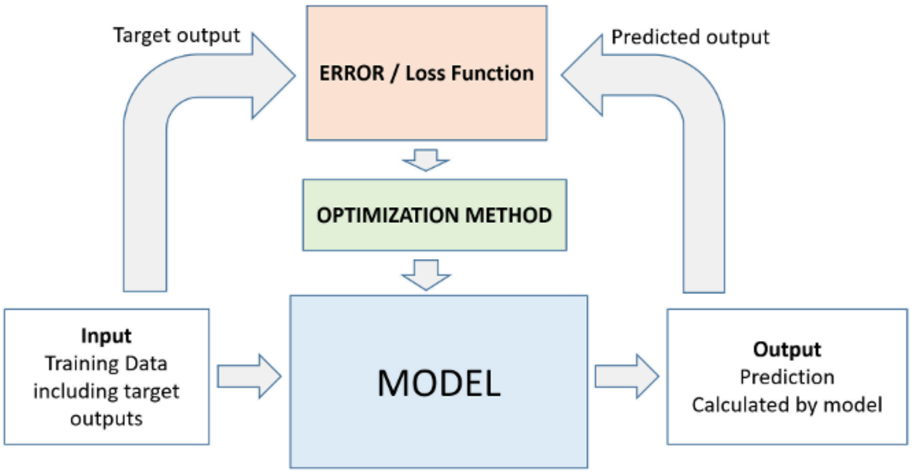


The output of the sigmoid function is called a logistic, hence the name *logistic regression*. For a mathematical discussion of logistic regression, sigmoid activation and cross entropy, check out this YouTube playlist. Logistic regression can also be applied to multi-class classification problems, with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

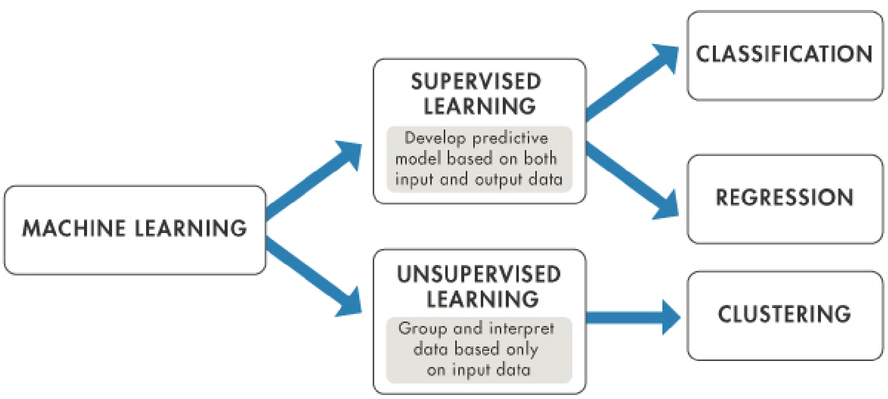
Machine Learning Workflow

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()
```

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)
```

```
Skipping, found downloaded files in "./weather-dataset-rattle-package" (use forc
```

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
```

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, its 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 interpertations can you draw from the above charts?

> **EXERCISE**: Visualize all the other columns of the dataset and study their relationship with the `RainToday` and `RainTomorrow` columns.

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Let's save our work before continuing.

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

```
import jovian
```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

(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
```

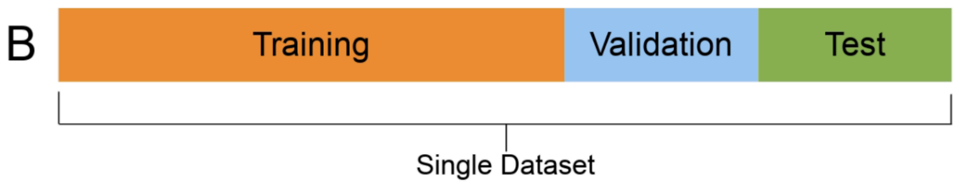
```
if use_sample:
    raw_df = raw_df.sample(frac=sample_fraction).copy()
```

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-

```
!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
```

```
val_df
```

```
test_df
```

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

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', 'WindG
```

```
target_col
```

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
```

```
train_targets
```

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()
```

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()
```

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

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
0	2	5.0	3.0	6	NaN
1	9	NaN	9.0	0	7.0
2	19	17.0	NaN	9	NaN

mean() →

	col1	col2	col3	col4	col5
0	2.0	5.0	3.0	6.0	7.0
1	9.0	11.0	9.0	0.0	7.0
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()
```

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()
```

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])
```

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

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

```
[np.float64(12.18482386562048),
 np.float64(23.235120301822324),
```

```
      np.float64(2.349974074310839),
      np.float64(5.472515506887154),
      np.float64(7.630539861047281),
      np.float64(39.97051988882308),
      np.float64(13.990496092519967),
      np.float64(18.631140782316862),
      np.float64(68.82683277087672),
      np.float64(51.44928834695453),
      np.float64(1017.6545771543717),
      np.float64(1015.2579625879797),
      np.float64(4.431160817585808),
      np.float64(4.499250233195188),
      np.float64(16.98706638787991),
      np.float64(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()
```

> **EXERCISE**: Apply some other imputation techniques and observe how they change the results of the model. You can learn more about other imputation techniques here: https://scikit-learn.org/stable/modules/impute.html

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()
```

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

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

```
?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])
```

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

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

```
Minimum:
[np.float64(-8.5),
 np.float64(-4.8),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(6.0),
```

```
 np.float64(0.0),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(980.5),
 np.float64(977.1),
 np.float64(0.0),
 np.float64(0.0),
 np.float64(-7.2),
 np.float64(-5.4)]
```

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

```
Maximum:
[np.float64(33.9),
 np.float64(48.1),
 np.float64(371.0),
 np.float64(145.0),
 np.float64(14.5),
 np.float64(135.0),
 np.float64(130.0),
 np.float64(87.0),
 np.float64(100.0),
 np.float64(100.0),
 np.float64(1041.0),
 np.float64(1039.6),
 np.float64(9.0),
 np.float64(9.0),
 np.float64(40.2),
 np.float64(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()
```

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

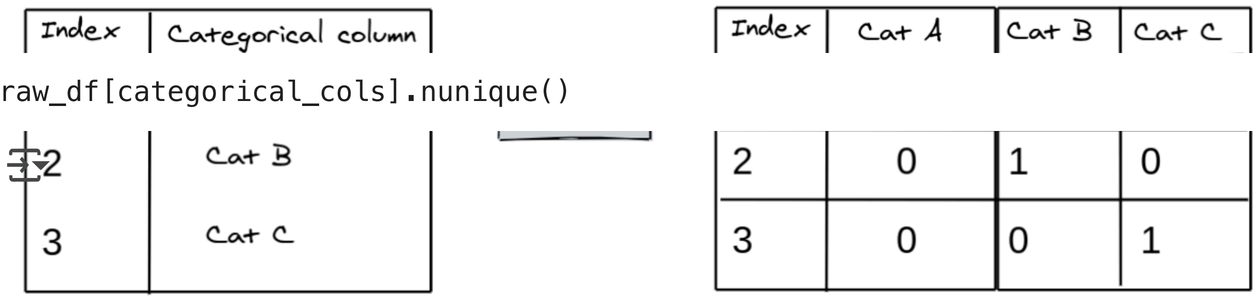
Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

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.




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

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

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

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

```
?OneHotEncoder
```

```
encoder = OneHotEncoder(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])
```

```
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_out(categorical_cols))
print(encoded_cols)
```

```
['Location_Adelaide', 'Location_Albany', 'Location_Albury', 'Location_AliceSprin
```

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`.

```
# One-hot encode categorical features
# Explicitly define categorical columns again to ensure correctness
categorical_cols = train_inputs.select_dtypes('object').columns.tolist()

# encoder = OneHotEncoder(sparse=False, handle_unknown='ignore').fit(raw_df[categori
encoder = OneHotEncoder(handle_unknown='ignore').fit(raw_df[categorical_cols]) # Cor
encoded_cols = list(encoder.get_feature_names_out(categorical_cols)) # Corrected bas

# train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols]) # O
train_inputs[encoded_cols] = pd.DataFrame(encoder.transform(train_inputs[categorical
# val_inputs[encoded_cols] = encoder.transform(val_inputs[categorical_cols]) # Origi
val_inputs[encoded_cols] = pd.DataFrame(encoder.transform(val_inputs[categorical_col
# test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols]) # Ori
test_inputs[encoded_cols] = pd.DataFrame(encoder.transform(test_inputs[categorical_c
```

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
```

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

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 200 ms, sys: 9.1 ms, total: 209 ms
Wall time: 173 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 1.78 s, sys: 786 ms, total: 2.56 s
Wall time: 1.95 s
```

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
```

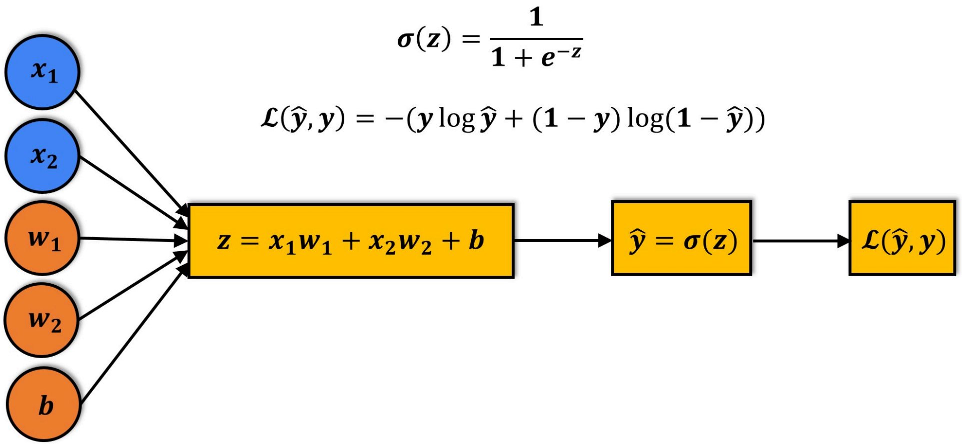
```
val_targets
```

Training a Logistic Regression Model

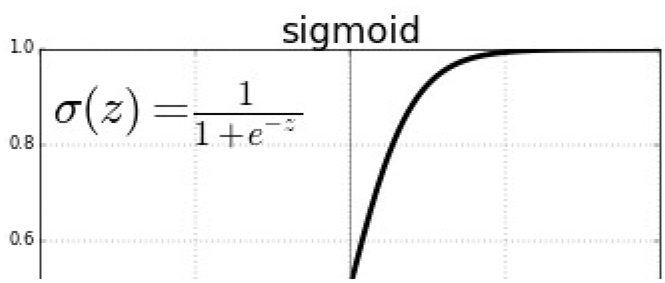
Logistic regression is a commonly used technique for solving binary classification problems. In a logistic regression model:

- 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:




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

```
?LogisticRegression
```

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

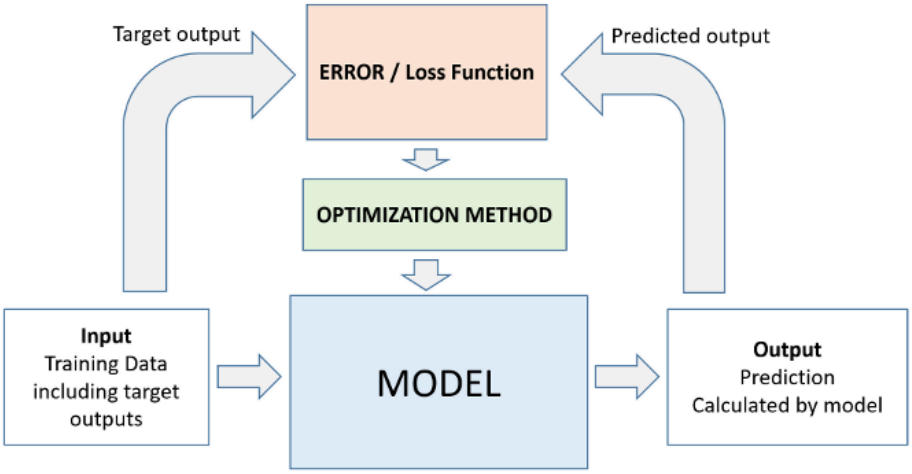
To train a logistic regression model, we can use the `LogisticRegression` class from Scikit-learn.

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

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

`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', '
```

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

```
[[0.8986324151652026, -2.8799288484272725, 3.1627783941533836, 0.854233199097419
```

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

```
[-2.44956109]
```

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
```

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

```
train_targets
```

We can output a probabilistic prediction using `predict_proba`.

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

```
array([[0.94401311, 0.05598689],
       [0.94074304, 0.05925696],
       [0.96093735, 0.03906265],
       ...,
       [0.98749147, 0.01250853],
       [0.98334725, 0.01665275],
       [0.87453712, 0.12546288]])
```

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

```
model.classes_
```

```
array(['No', 'Yes'], dtype=object)
```

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)
```

```
0.8519206433440829
```

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 (T**N**)	False Positives (F**P**) **Type I error**
	Positive **+**	False Negatives (F**N**) **Type II error**	True Positives (T**P**)

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

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

```
array([[0.94621341, 0.05378659],
       [0.4776585 , 0.5223415 ]])
```

Let's define a helper function to generate predictions, compute the accuracy score and plot a confusion matrix for a given st 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))
```

```
0.49898872034227926
```

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

```
0.7734344612991054
```

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

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

> **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?

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

> **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?

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

> **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?

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] jovian.commit() is no longer required on Google Colab. If you ran this
then just save this file in Colab using Ctrl+S/Cmd+S and it will be updated on J
Also, you can also delete this cell, it's no longer necessary.
```

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]) # O
new_input_df[encoded_cols] = pd.DataFrame(encoder.transform(new_input_df[categorical
```

```
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
```

```
array([0., 1.])
```

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)[0]
```

> **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()
```

```
Start coding or generate with AI.
```

```
Start coding or generate with AI.
```

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
import os
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')

# Load the dataset
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 >

# Create inputs and targets
input_cols = list(train_df.columns)[1:-1]  # Exclude Date and RainTomorrow
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()
categorical_cols = train_inputs.select_dtypes('object').columns.tolist()

# Impute missing numeric 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 using sparse=True and toarray()
encoder = OneHotEncoder(handle_unknown='ignore')  # sparse=True by default
encoder.fit(raw_df[categorical_cols])
encoded_cols = list(encoder.get_feature_names_out(categorical_cols))

# Transform and convert to DataFrame using .toarray()
train_encoded = pd.DataFrame(encoder.transform(train_inputs[categorical_cols]).toarr
                             columns=encoded_cols, index=train_inputs.index)
val_encoded = pd.DataFrame(encoder.transform(val_inputs[categorical_cols]).toarray()
                           columns=encoded_cols, index=val_inputs.index)
test_encoded = pd.DataFrame(encoder.transform(test_inputs[categorical_cols]).toarray
                            columns=encoded_cols, index=test_inputs.index)

# Drop original categorical columns
train_inputs.drop(columns=categorical_cols, inplace=True)
val_inputs.drop(columns=categorical_cols, inplace=True)
test_inputs.drop(columns=categorical_cols, inplace=True)

# Concatenate the one-hot encoded columns
train_inputs = pd.concat([train_inputs, train_encoded], axis=1)
val_inputs = pd.concat([val_inputs, val_encoded], axis=1)
test_inputs = pd.concat([test_inputs, test_encoded], axis=1)

# 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]
```

```
Skipping, found downloaded files in "./weather-dataset-rattle-package" (use forc
```

> **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
```