mlnuggets

Logistic regression in Python with Scikit-learn


Brian Mutea

10 min read

Table of Contents

Subscribe

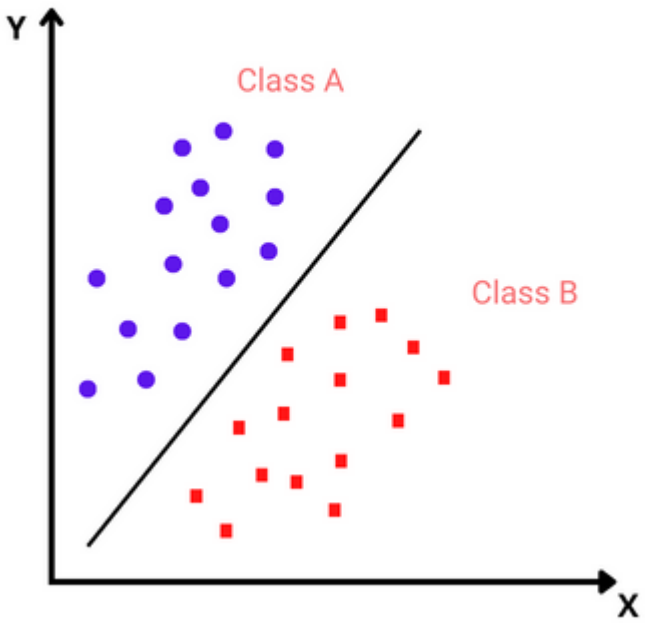
In linear regression, we tried to understand the relationship between one or more predictor variables and a continuous response variable. This article will explore logistic regression, where the response variable will be discrete or **categorical**.

What is classification?

Classification is a supervised machine learning problem of predicting which category or class a particular observation belongs to based on its features.

For instance, one popular classification problem is Image classification. We may want to classify images into different classes: dog, cat, donkey, and human.

Based on pre-classified images of dogs and cats, a classification model can be trained using algorithms like **Convolutional Neural Networks (CNN)** to classify the images into their respective categories.



Classification: Image by author

There are two types of classification problems:

- **Binomial** or **binary classification:** has exactly two classes to choose from.
- **Multinomial** or **Multiclass classification:** has three or more classes to choose from.

Some examples of classification algorithms:

- **Logistic regression**
- Decision trees
- Random forest
- Artificial neural networks
- XGBoost

What is Logistic regression?

Logistic regression is a supervised **classification** model known as the **logit** model. It estimates the **probability** of something occurring, like 'will buy' or 'will not buy,' based on a dataset of independent variables. The outcome should be a categorical or a discrete value. The outcome can be either a 0 and 1, true and false, yes and no, and so on.

The model does not give an exact 0 and 1 but a value between 0 and 1. Unlike linear regression, which fits a regression line, logistic regression fits an 'S'-shaped logistic function(**Sigmoid function**).

Logistic function(Sigmoid function)

Since logistic regression is a binary classification technique, the values predicted should fall close to either 0 or 1. This is why a sigmoid function is convenient. In mathematical terms:

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

Image by author

Where:

- **p(x)** is the predicted probability that the output for a given $\mathbf{x}$ is equal to 1.
- **z** is the linear function since logistic regression is a linear classifier which translates to:
  - $\mathbf{z = b_0 + b_1x_1 + ... + b_rx_r}$
    Where:
    - $\mathbf{b_0}$, $\mathbf{b_1}$ ...$\mathbf{b_r}$ are the model's **predicted weights** or **coefficients**.
    - **x** the feature values.

Note that the **z** can be defined as the log of the probability of something happening(1 = p(x) = will buy) divided by the probability of something not happening(0 = 1-p(x) = will not buy). For this reason, **z** is referred to as the **log-odds or natural logarithm of odds**.

The **odds** mean the probability of success over the probability of failure.

$$z = log\left(\frac{p(x)}{1 - p(x)}\right)$$

Log-odds: Image by author

For example, let's imagine we are trying to build a model to predict the probability of a tumor spreading given its size in centimeters. After plotting the dataset, we can use linear regression to model the status p(x) as a function with the sigmoid function.

So let's say after fitting the curve, we get the following values:

- **b_0 = -5.47**
- **b_1 = 1.87**

Log-odds would be:

```
z = -5.47 + (1.87 x 3)
```

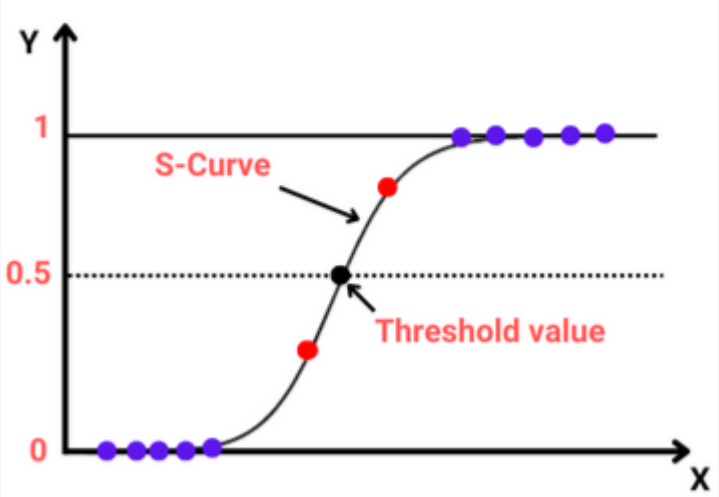
Given a tumor size of 3, we can check the probability with the sigmoid function as:

$$P(spread) = p(3) = \frac{1}{1 + e^{-(-5.47+1.87x3)}} = 0.53$$

Image by author

The probability that the tumor of size 3cm spreads is 0.53, equal to 53%.

💡 In logistic regression, we use a **threshold value** that defines the probability of either 0 or 1. For instance, we can set a threshold value of 0.5 where values above the threshold take the value 1, and those below take the value 0. So in our example above, The probability that a tumor of size 3cm will spread takes 1.



Sigmoid function: Image by author

Maximum Likelihood Estimation(MLE)

We maximize the **log-likelihood function (LLF)** to get the best coefficients or the predicted weights. This involves finding the best fit sigmoid curve that provides the optimal coefficients, and this method is called **Maximum Likelihood Estimation.**

$$LLF = \sum \left(y_i log\left(p\left(x_i\right)\right) + \left(1 - y_i\right) log(1 - p(x_i))\right)$$

Image by author

While y_i = 0, the LLF for that observation is equal to log(1-p($\mathbf{x}_i$)), and if $p(\mathbf{x}_i)$ is close to y_i = 0, the log(1-p($\mathbf{x}_i$)) is close to 0. The main goal is to maximize the LLF. So If $p(\mathbf{x}_i)$ is distant from 0, then the log(1-p($\mathbf{x}_i$)) drops significantly, and that's not what we want.

Types of logistic regression

So far, we have discussed one type of **binary** type of logistic regression where the outcome is a 0/1, True/False, and so on. There are two more types:

- **Multinomial logistic regression**: This type of regression has three or more unordered types of dependent variables, such as cats/dogs/donkeys.
- **Ordinal logistic regression**: Has three or more ordered dependent variables such as poor/average/ good or high/medium/average.

Assumptions of logistic regression

Logistic regression assumes that:

- The response variable is **binary** or **dichotomous**.
- The observations or independent variables have very little or **no multicollinearity.**
- There are **no extreme outliers.**
- There is a **linear relationship** between the predictor variables and the log-odds of the response variable.
- **Large sample sizes** for a more reliable analysis.

Logistic regression with Scikit-learn

To implement logistic regression with Scikit-learn, you need to understand the Scikit-learn modeling process and linear regression.

The steps for building a logistic regression include:

- **Import** the packages, classes, and functions.
- **Load** the data.
- Exploratory Data Analysis(**EDA**).
- **Transform** the data if necessary.
- **Fit** the classification model.
- **Evaluate** the performance model.

Importing packages

First, you need to import Seaborn for visualization, NumPy, and Pandas. In addition, import:

- `LogisticRegression` for fitting the model.
- `confusion_matrix` and `classification_report` for evaluating the model.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report
```

Loading the dataset

Import the Social Network Ads dataset from Kaggle. The data is a CSV file with data that will help us build a logistic regression model to show which users

purchased or did not purchase a product.

```
social_N_data = pd.read_csv('Social_Network_Ads.csv')
pd.concat([social_N_data.head(), social_N_data.tail()])
```

Out[92]:

	User ID	Gender	Age	EstimatedSalary	Purchased
0	15624510	Male	19	19000	0
1	15810944	Male	35	20000	0
2	15668575	Female	26	43000	0
3	15603246	Female	27	57000	0
4	15804002	Male	19	76000	0
395	15691863	Female	46	41000	1
396	15706071	Male	51	23000	1
397	15654296	Female	50	20000	1
398	15755018	Male	36	33000	0
399	15594041	Female	49	36000	1

Exploratory Data Analysis

Analyzing the data first is key to understanding its characteristics. We will begin with checking the missing values.

```
social_N_data.isnull().any()
```

Out[28]:

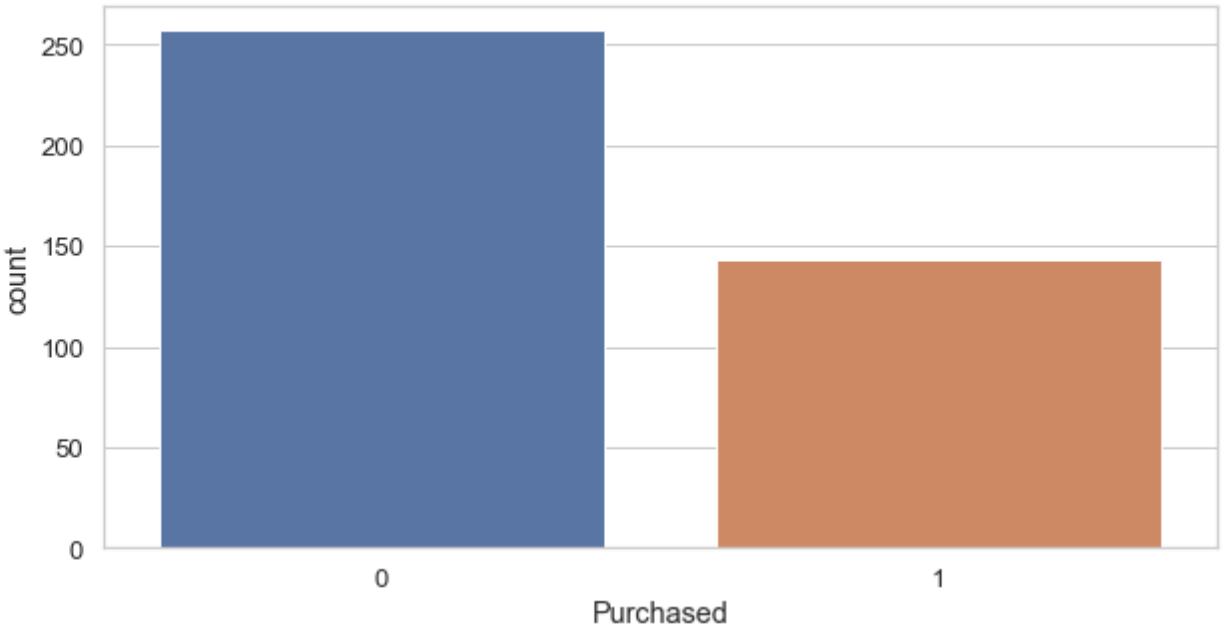
```
User ID             False
Gender              False
Age                 False
EstimatedSalary     False
Purchased           False
dtype: bool
```

No null values in the dataset.

Check for the total number of those who purchased and those who did not purchase:

```
sns.countplot(x='Purchased', data=social_N_data)
```

```
Out[94]: <AxesSubplot:xlabel='Purchased', ylabel='count'>
```

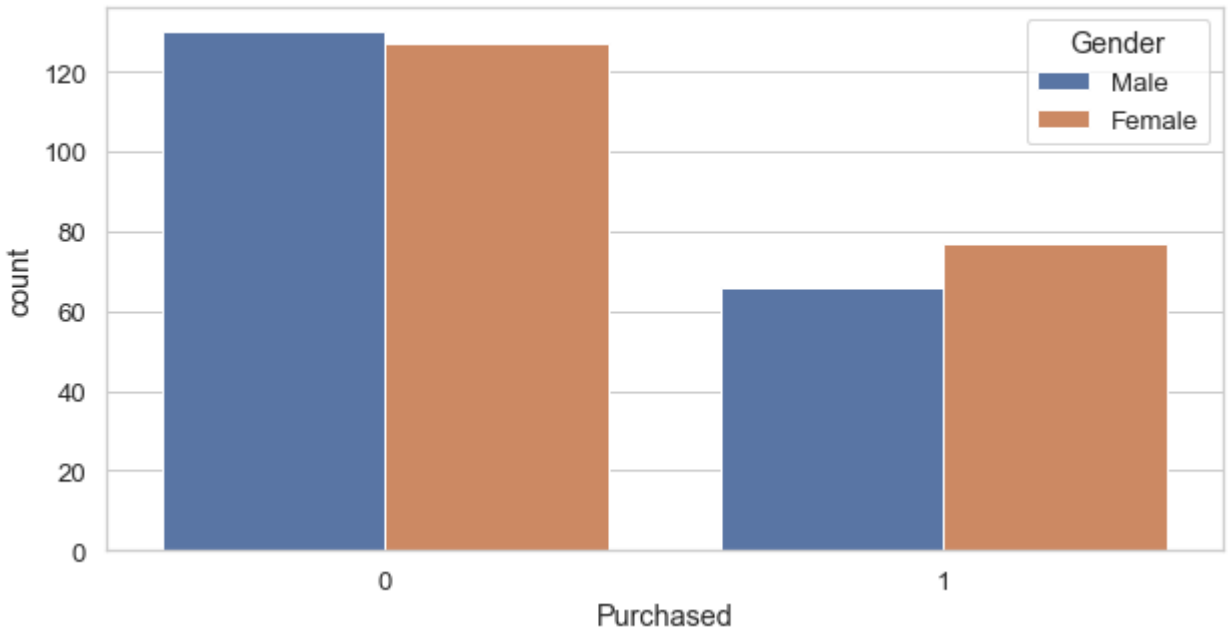


Zero indicates those who did not purchase and 1 for those who bought.

Check for how many males and females purchased the product:

```
sns.countplot(x='Purchased', hue='Gender', data=social_N_data)
```

```
Out[98]: <AxesSubplot:xlabel='Purchased', ylabel='count'>
```

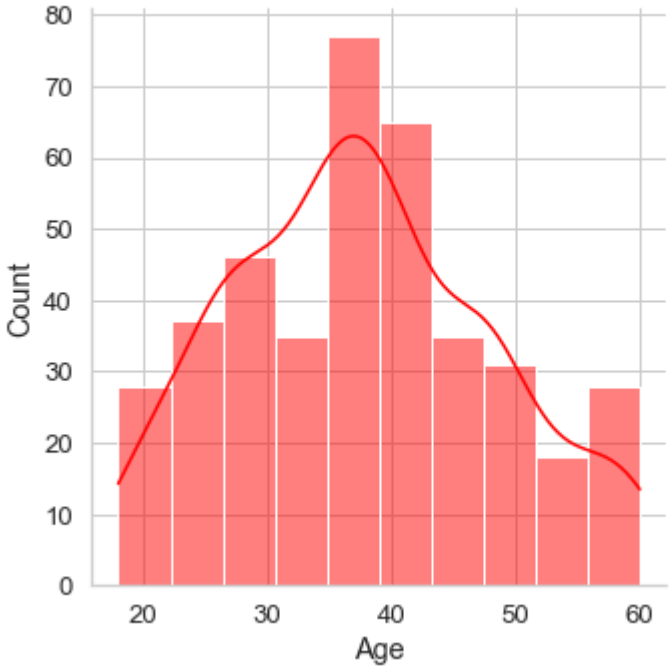


From the plot, we can see that most people who did not purchase are male, and the majority of those who purchased are female.

We can also check the age distribution in the dataset:

```
sns.displot(x='Age', data=social_N_data, color='red', kde=True)
```

```
Out[105]: <seaborn.axisgrid.FacetGrid at 0x1501f3c7040>
```



Cleaning the data

We will use the `Gender` , `Age` , and `EstimatedSalary` columns from the dataset for the logistic regression. This means that we do not require the `UserID` column. Thus we will drop it.

```
social_N_data.drop('User ID', axis=1, inplace=True)
```

Out[112]:

	Gender	Age	EstimatedSalary	Purchased
0	Male	19	19000	0
1	Male	35	20000	0
2	Female	26	43000	0
3	Female	27	57000	0
4	Male	19	76000	0
...	...	...	...	...
395	Female	46	41000	1
396	Male	51	23000	1
397	Female	50	20000	1
398	Male	36	33000	0
399	Female	49	36000	1

400 rows × 4 columns

Changing categorical data into dummies

Let's look at the `info` of the dataset to get a general idea of what it contains.

```
social_N_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 400 entries, 0 to 399
Data columns (total 5 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   User ID          400 non-null    int64
 1   Gender           400 non-null    object
 2   Age              400 non-null    int64
 3   EstimatedSalary  400 non-null    int64
 4   Purchased        400 non-null    int64
dtypes: int64(4), object(1)
memory usage: 15.8+ KB
```

The `Gender` variable is categorical. For the model to work, we will convert it into dummy variables using the Pandas `get_dummies` or the `oneHotEncoder` method.

Change `Gender` to dummy variable and drop the first dummy to prevent multicollinearity:

```
gender = pd.get_dummies(social_N_data['Gender'], drop_first=True)
```

```
social_N_data.drop('Gender',axis=1,inplace=True)
```

```
social_N_data = pd.concat([social_N_data,gender], axis=1)
```

```
In [20]: social_N_data.head(8)
```

Out[20]:

	Age	EstimatedSalary	Purchased	Male
0	19	19000	0	1
1	35	20000	0	1
2	26	43000	0	0
3	27	57000	0	0
4	19	76000	0	1
5	27	58000	0	1
6	27	84000	0	0
7	32	150000	1	0

When the Male value is 1, it means the gender is male, and when the value is 0, the gender is female. We did not require both the Female and Gender variables in the dataset, as one can be used to predict the other.

Splitting the data into independent(X) and dependent(y) variables

Split the data into independent and dependent variables.

```
X = social_N_data.iloc[:,[0,1,3]] # Age, EstimatedSalary and Male
X.head()
```

Out[54]:

	Age	EstimatedSalary	Male
0	19	19000	1
1	35	20000	1
2	26	43000	0
3	27	57000	0
4	19	76000	1

```
y = social_N_data.iloc[:, 2] # Purchased
```

```
Out[51]: 0      0
         1      0
         2      0
         3      0
         4      0
               ..
         395    1
         396    1
         397    1
         398    0
         399    1
         Name: Purchased, Length: 400, dtype: int64
```

Feature scaling

Feature scaling is a method used to normalize the range of independent variables. The method enables the independent variables to be in the same range.

When working with large datasets, scaling plays a significant role in improving the performance of the model.

In the data, we will import the `StandardScaler` from Scikit-learn `preprocessing` module and use it to transform the data. For instance, there is a big difference between the values of the Age variable and those of EstimatedSalary.

```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
X = sc.fit_transform(X)
X
```

```
Out[61]: array([[-1.78179743, -1.49004624,  1.02020406],
                [-0.25358736, -1.46068138,  1.02020406],
                [-1.11320552, -0.78528968, -0.98019606],
                ...,
                [ 1.17910958, -1.46068138, -0.98019606],
                [-0.15807423, -1.07893824,  1.02020406],
                [ 1.08359645, -0.99084367, -0.98019606]])
```

Splitting the dataset into train and test sets

Split the dataset into training and testing sets using the `train_test_split` function.

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, ran

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)

'''
(280, 3)
```

```
(120, 3)
(280,)
(120,)
'''
```

Fitting the logistic regression model and predicting test results

Now that the dataset is well prepared, we can train the model by importing the `LogisticRegression` class of the Scikit-learn `linear_model` module.

Training is done by calling the `fit` method and pass the training data.

```
from sklearn.linear_model import LogisticRegression
classifier = LogisticRegression()
classifier.fit(X_train, y_train)
```

```
Out[68]: LogisticRegression()
```

The model is now trained on the training set. Let's perform prediction on the test set using the `predict` method.

```
y_pred = classifier.predict(X_test)
```

Let's create a Pandas DataFrame and compare the predicted and actual values.

```
result = pd.DataFrame({'Actual' : y_test, 'Predicted' : y_pred})
result
```

Out[27]:

	Actual	Predicted
398	0	0
125	0	0
328	1	1
339	1	1
172	0	0
...	...	...
91	0	0
322	0	0
248	0	0
186	0	0
395	1	0

120 rows × 2 columns

The `coef_` and `intercept_` attributes give the model coefficient and intercept.

```
classifier.coef_
# array([[2.36839196, 1.42929561, 0.20973787]])
classifier.intercept_
# array([-1.1352347])
```

Evaluating the model

There are various ways of checking the performance of the model.

Using `predict_proba`

It returns the matrix of probabilities that the predicted output is equal to zero or one.

```
print(classifier.predict_proba(X))
```

```
Out[38]: array([[9.99305047e-01, 6.94953436e-04],
                [9.73650395e-01, 2.63496050e-02],
                [9.93938614e-01, 6.06138563e-03],
                [9.86425967e-01, 1.35740330e-02],
                [9.92449933e-01, 7.55006749e-03],
                [9.78634880e-01, 2.13651200e-02],
                [9.59015410e-01, 4.09845897e-02],
                [3.21160956e-01, 6.78839044e-01],
                [9.95160538e-01, 4.83946203e-03],
                [8.94774017e-01, 1.05225983e-01],
                [9.71990649e-01, 2.80093506e-02],
                [9.91181077e-01, 8.81892308e-03],
                [9.85694423e-01, 1.43055770e-02],
                [9.87533664e-01, 1.24663357e-02],
                [9.92255565e-01, 7.74443464e-03],
                [9.20459783e-01, 7.95402167e-02],
                [6.64896819e-01, 3.35103181e-01],
```

From the matrix, each row represents a single observation. The first column is the probability that the product is not purchased(1-p(x)), and the second column is the probability that the product is purchased(p(x)).

Using confusion matrix

From the Scikit-learn `metrics` module, we import `confusion_matrix`. The confusion matrix is the number of correct and incorrect predictions column-wise, showing the following values:

- **True negatives(TN)** in the upper-left position.
- **False negatives(FN)** in the lower-left position.
- **False positives(FP)** in the upper-right position.
- **True positives(TP)** in the lower-right position.

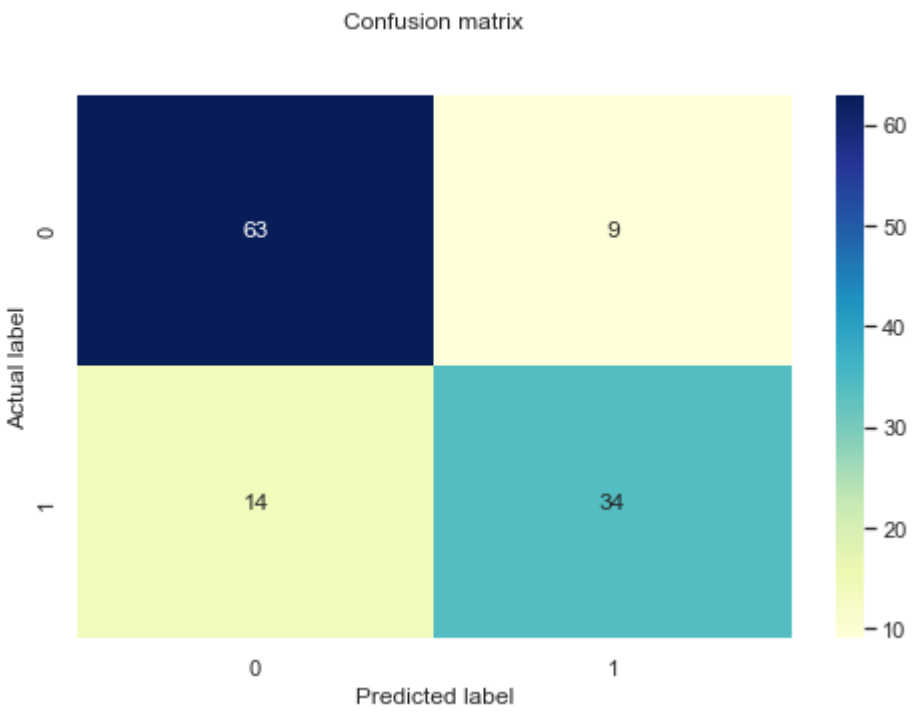
```
from sklearn.metrics import confusion_matrix
cf_matrix = confusion_matrix(y_test, y_pred)
cf_matrix
```

```
Out[42]: array([[63,  9],
                [14, 34]], dtype=int64)
```

The output of the confusion matrix is a 2*2 matrix since the model is a binary classification. Let's visualize it better using a heatmap and explain.

```
sns.heatmap(pd.DataFrame(cf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

```
Out[32]: Text(0.5, 21.5, 'Predicted label')
```



From the `confusion_matrix`, we have the following observations:

- **63 TN predictions:** zeros predicted correctly.
- **14 FN predictions:** ones wrongly predicted as zeros.
- **9 FP predictions:** zeros that were wrongly predicted as ones.
- **34 TP predictions:** ones predicted correctly.

To calculate the model's **accuracy** from the confusion matrix, we divide the sum of TN and TP by the sum of all the predictions.

$$Accuracy = \frac{TN + TP}{TN + TP + FP + FN}$$

Image by author

```
Accuracy = (63 + 34)/(63 + 34 + 9 + 14)
Accuracy
# 0.8083333333333333

# Also same result from sklearn accuracy_score
from sklearn.metrics import accuracy_score

accuracy_score(y_test,y_pred)
#0.8083333333333333
```

The accuracy of our model is about 80% which is ideal.

Confusion matrix metrics

The `classification_report` gives a more comprehensive report of the model's performance.

```
target_names = ['will NOT PURCHASE', 'will PURCHASE']
print(classification_report(y_test, y_pred,target_names=target_names))
```

	precision	recall	f1-score	support
will NOT PURCHASE	0.82	0.88	0.85	72
will PURCHASE	0.79	0.71	0.75	48
accuracy			0.81	120
macro avg	0.80	0.79	0.80	120
weighted avg	0.81	0.81	0.81	120

Classification report

Complete code for logistic regression

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix, classification_report, accuracy

social_N_data = pd.read_csv('Social_Network_Ads.csv')
pd.concat([social_N_data.head(), social_N_data.tail()])

#CHECK FOR NULL VALUES
social_N_data.isnull().any()

# CLEAN THE DATA
social_N_data.drop('User ID', axis=1, inplace=True)

# CHANGE CATEGORICAL VARIABLE TO DUMMIES
social_N_data.info()
```

```
gender = pd.get_dummies(social_N_data['Gender'], drop_first=True)
social_N_data.drop('Gender',axis=1,inplace=True)
social_N_data = pd.concat([social_N_data,gender], axis=1)

# SPLIT DATA TO INDEPENDENT AND DEPENDENT VARIABLES
X = social_N_data.iloc[:,[0,1,3]] # Age, EstimatedSalary and Male
y = social_N_data.iloc[:, 2] # Purchased

# FEATURE SCALING
sc = StandardScaler()
X = sc.fit_transform(X)

# SPLIT DATA TO TRAIN AND TEST SET
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, ran

# FIT/TRAIN MODEL
classifier = LogisticRegression()
classifier.fit(X_train, y_train)

# PREDICTIONS
y_pred = classifier.predict(X_test)
result = pd.DataFrame({'Actual' : y_test, 'Predicted' : y_pred})
print(result)

# EVALUATE MODEL
# predic_proba()
# print(classifier.predict_proba(X) # uncheck if needed
#confusion matrix
cf_matrix = confusion_matrix(y_test, y_pred)
print('Confusion Matrix \n', cf_matrix)

sns.heatmap(pd.DataFrame(cf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')

print('Accuracy of model')
print(accuracy_score(y_test,y_pred) * 100, '%')
#0.8083333333333333
```

```
# classification report
target_names = ['will NOT PURCHASE', 'will PURCHASE']
print('Classification report: \n', classification_report(y_test, y_pred,targe
```

. . .

Final thoughts

Logistic regression is direct and friendly to implement. Hopefully, you can now analyze various datasets using the logistic regression technique. Generally, we have covered:

- Logistic regression in relation to the classification.
- Logistic or Sigmoid function.
- How logistic regression uses MLE to predict outcomes.
- Assumptions of logistic regression.
- Step-by-step implementation of logistic regression.

. . .

The Complete Data Science and Machine Learning Bootcamp on Udemy is a great next step if you want to keep exploring the data science and machine learning field.

Follow us on LinkedIn, Twitter, and GitHub, and subscribe to our *blog, so you don't miss a new issue.*

Data Science

Brian Mutea

Software Engineer | Data Scientist with an appreciable passion for building models that fix problems and sharing knowledge.

Discussion

Community guidelines

0 comments

Start the conversation

Become a member of **mlnuggets** to start commenting.

Sign up now

Already a member? Sign in

YOU MIGHT ALSO LIKE

Top 20 Pandas Functions You Aren't Using, Which You Should Be Using Public

This blog post will explore 20 powerful and unique Pandas functions that can significantly enhance your data analysis workflow. We will be using the famous Iris dataset as an example to demonstrate each function. The Iris dataset contains four features: Sepal Length, Sepal Width,...

Muhammad Anas Data Science

Entropy, information gain, and Gini impurity(Decision tree splitting criteria) Public

Decision trees are supervised machine-learning models used to solve classification and regression problems. They help to make decisions by breaking down a problem with a bunch of if-else-then-like evaluations that result in a tree-like structure. For quality and viable decisions to be made, a...

Brian Mutea

Data Science

FEATURED POSTS

How to Train YOLOv5 Object Detection on Custom Data

How to Build Large Language Model Applications with PaLM API and LangChain

How to Build LLM Applications With LangChain and Openai

AUTHORS →

Derrick Mwiti

Google Developer Expert - Machine Learning

Brian Mutea

Software Engineer | Data Scientist with an appreciable passion for building models that fix problems and sharing knowledge.

Kamanda Wycliffe

Data Scientist and Ethical Hacker with interest in data modeling and data-driven cyber security systems.

MLNUGGETS

Machine Learning Technical Deep Dives.

NAVIGATION

Privacy

Contact us

Terms

About

NEWSLETTER

Your email address

Subscribe