

SCIKIT-LEARN CLASSIFICATION TUTORIAL



Here we show how to use scikit-learn. The code for this example is [here](#). Download the data from Kaggle [here](#).

(This article is part of our [scikit-learn Guide](#). Use the right-hand menu to navigate.)

Which machine learning framework should you use?

Before we show you how scikit-learn works, it's worth discussing which ML framework to use. I put this up front because too many people starting data science think they must start with TensorFlow, but that is overkill for most of your problems. For low-level approaches, which TensorFlow specializes in, it is too complicated; there are easier approaches.

scikit-learn

TensorFlow

Keras

Spark ML

This general-purpose ML framework is both easy to use and can tackle most ML problems. It is very popular among data scientists. Even data scientists who use other frameworks often deploy scikit-learn utilities in part of their code.

TensorFlow is designed for one purpose: neural networks. It is very low level, which means you'll need a lot of knowledge about NumPy arrays and neural network theory. Google sells special CPUs, called TPUs, which are designed to process tensors at very large scale. So you can see it's designed for computing intensive tasks, like facial recognition, but most business problems are less complicated than facial recognition. We've written tutorials on how to use ML with TensorFlow & Keras [here](#).

If you want to use TensorFlow, then use Keras, as it acts as a front end, thus making it a lot easier. You'll be less likely to make mistakes that produce wrong answers. Keras also works in front of other popular ML frameworks, also making those easier to use. We explain [how to use Keras here](#).

scikit-learn is designed to run on one server. If you have a large amount of data, you might want to use Spark ML, as it's designed to run across a cluster. And Spark ML is easy-to-understand.

All the Python ML frameworks start pretty much the same, starting with the same tools:

- **Pandas.** This organizes csv, json, Spark, and other types of data into rows and columns. Pandas greatly simplifies all types of data, but its advanced features can get complicated.
- **NumPy.** This is tied closely to both Pandas and matplotlib. NumPy performs best when handling the most important machine learning task: the computationally expensive operation of multiplying matrices in multiple dimensions. As those grow, they can quickly run your computer out of memory.
- **Matplotlib.** This draws charts, like histograms, line charts, etc. Charting data is a good way to explore data while you work with the data, and they can illustrate your resulting conclusions at the end of your program.
- **Seaborn.** This framework, on top of matplotlib, is designed specifically for data science.

The data

The data we are looking at is glucose, body mass index, etc. taken from two sets of people: those who are diabetic and those who are not. That classification 1 (diabetic) and 0 (not) is in the **Outcome** column. The goal is to use that data to train a predictive model that will show given certain health

indicators whether or not a person is likely to have or will get diabetes.

The algorithm

There are a lot of ways to approach a classification problem, like logistic regression or even neural networks. Here we use the [Support Vector Machine \(SVM\)](#).

The first step is to read the data into a dataframe.

```
import pandas as pd
import seaborn as sns
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

```
data = pd.read_csv('/home/ubuntu/Downloads/diabetes.csv', delimiter=',')
```

Then, take a look at it.

```
data.head()
```

Machine learning requires that you split the data into **features** and **labels**.

- Features are the characteristics of what you are looking at, also known as the independent variables.
- Labels are what you are trying to predict, aka the **dependent variables**.

Classification means there are a finite set of outcomes. Here there are two, so you could call it a **binary classification problem**.

As you can see, the **outcome**, whether someone has diabetes or not, is the last column. So, the rest are features. It will be easy to split this data since the labels are on the end.

Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148	72	35	0	33.6
1	1	85	66	29	0	26.6
2	8	183	64	0	0	23.3
3	1	89	66	23	94	28.1
4	0	137	40	35	168	43.1

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

The Pandas **drop()** command means to create a new dataframe by taking an existing dataframe and dropping one or more columns. **axis=1** means we are referring to the columns and not the rows (which for Pandas is aka the **index**).

```
x = data.drop("Outcome", axis=1)
```

data is a Pandas **Series** and not a Pandas dataframe. This means it has one column only, but it still has the index column. **np.ravel()** will flatten that to an array.

```
y=np.ravel(data)
```

The standard procedure is to take the input data and create training and test datasets by splitting them by some amount. Here we pick 50%.

- **Training datasets** are used to train the model.
- **Test datasets** are used to make predictions based on that trained model.

We use x and y since the familiar equation for a line $y = mx + b$. For machine learning y is a vector and m and x are matrices, meaning an n -dimensional vector. b is bias, which is a single real number.

```
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size = 0.5,  
random_state=50)
```

Now we normalize the data. Basically, this calculates the value $((x - \mu) / \delta)$ where μ is the mean and δ is the standard deviation. This puts all the features on the same scale, which is a regular machine learning practice. In other words, it makes large numbers small so that all the numbers are about the same size.

```
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler().fit(x_train)
```

```
x_train = scaler.transform(x_train)
```

```
x_test = scaler.transform(x_test)
```

First, we declare the model. We are using a support vector machine.

```
from sklearn.svm import SVC  
svc_model = SVC()
```

Then we train it: it's that simple when you use scikit-learn. There's no other data manipulation required.

```
svc_model.fit(x_train, y_train)
```

The **fit()** function responds with this information.

```
SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,  
decision_function_shape='ovr', degree=3, gamma='auto_deprecated',  
kernel='rbf', max_iter=-1, probability=False, random_state=None,  
shrinking=True, tol=0.001, verbose=False)
```

Now we use the test data to create predictions.

```
y_predict = svc_model.predict(x_test)
```

Then we show how accurate those predictions are by creating what is called a **confusion matrix**. This is a visual way to see how many times the model was right versus how many times it was incorrect.

```
from sklearn.metrics import classification_report, confusion_matrix

cm = np.array(confusion_matrix(y_test, y_predict, labels=))

confusion = pd.DataFrame(cm, index=, columns=)
```

	Predicted Diabetes	Predicted Healthy
Diabetic	225	23
Not Diabetic	68	68

It's a little difficult to understand that display at first, so think of it like this:

Diabetic (Outcome = 1) True positive. Patient is diabetic and model correctly predicted that. False positive. Patient was not diabetic but model said patient was diabetic.

Not Diabetic (Outcome = 0) False positive. Patient was not diabetic, but model said patient was diabetic. True negative, patient not diabetic and model predicted that.

Here is a graphical way to show the same results using the powerful Seaborn extension to Matplotlib:

```
sns.heatmap(confusion,annot=True,fmt='g')
```

The classification report prints a summary of the model, showing a 77% precision. This means our model accurately predicts diabetes 77% of the time.

```
print(classification_report(y_test, y_predict))
```

	precision	recall	f1-score	support
0	0.77	0.91	0.83	248
1	0.75	0.50	0.60	136
micro avg	0.76	0.76	0.76	384
macro avg	0.76	0.70	0.72	384
weighted avg	0.76	0.76	0.75	384

The results were nearly the same as when we used [Keras as a neural network](#).