Machine Learning with Python

Python is a popular platform used for research and development of production systems. It is a vast language with number of modules, packages and libraries that provides multiple ways of achieving a task.

Python and its libraries like NumPy, SciPy, Scikit-Learn, Matplotlib are used in data science and data analysis. They are also extensively used for creating scalable machine learning algorithms. Python implements popular machine learning techniques such as Classification, Regression, Recommendation, and Clustering.

Python offers ready-made framework for performing data mining tasks on large volumes of data effectively in lesser time. It includes several implementations achieved through algorithms such as linear regression, logistic regression, Naïve Bayes, k-means, K nearest neighbor, and Random Forest.

**Python in Machine Learning**

Python has libraries that enables developers to use optimized algorithms. It implements popular machine learning techniques such as recommendation, classification, and clustering. Therefore, it is necessary to have a brief introduction to machine learning before we move further.

**What is Machine Learning?**

Data science, machine learning and artificial intelligence are some of the top trending topics in the tech world today. Data mining and Bayesian analysis are trending and this is adding the demand for machine learning. This tutorial is your entry into the world of machine learning.

Machine learning is a discipline that deals with programming the systems so as to make them automatically learn and improve with experience. Here, learning implies recognizing and understanding the input data and taking informed decisions based on the supplied data. It is very difficult to consider all the decisions based on all possible inputs. To solve this problem, algorithms are developed that build knowledge from a specific data and past experience by applying the principles of statistical science, probability, logic, mathematical optimization, reinforcement learning, and control theory.

**Applications of Machine Learning Algorithms**

The developed machine learning algorithms are used in various applications such as −

* Vision processing
* Language processing
* Forecasting things like stock market trends, weather
* Pattern recognition
* Games
* Data mining
* Expert systems
* Robotics

**Steps Involved in Machine Learning**

A machine learning project involves the following steps −

* Defining a Problem
* Preparing Data
* Evaluating Algorithms
* Improving Results
* Presenting Results

The best way to get started using Python for machine learning is to work through a project end-to-end and cover the key steps like loading data, summarizing data, evaluating algorithms and making some predictions. This gives you a replicable method that can be used dataset after dataset. You can also add further data and improve the results.

**Libraries and Packages**

To understand machine learning, you need to have basic knowledge of Python programming. In addition, there are a number of libraries and packages generally used in performing various machine learning tasks as listed below −

        **numpy** − is used for its N-dimensional array objects

        **pandas** − is a data analysis library that includes dataframes

        **matplotlib** − is 2D plotting library for creating graphs and plots

        **scikit-learn** − the algorithms used for data analysis and data mining tasks

        **seaborn** − a data visualization library based on matplotlib

**Installation**

You can install software for machine learning in any of the two methods as discussed here −

**Method 1**

Download and install Python separately from **python.org** on various operating systems as explained below −

To install Python after downloading, double click the **.exe** (for Windows) or **.pkg** (for Mac) file and follow the instructions on the screen.

For Linux OS, check if Python is already installed by using the following command at the prompt −

$ python --version. ...

If Python 2.7 or later is not installed, install Python with the distribution's package manager. Note that the command and package name varies.

On Debian derivatives such as Ubuntu, you can use **apt** −

$ sudo apt-get install python3

Now, open the command prompt and run the following command to verify that Python is installed correctly −

$ python3 --version

Python 3.6.2

Similarly, we can download and install necessary libraries like numpy, matplotlib etc. individually using installers like **pip**. For this purpose, you can use the commands shown here −

$pip install numpy

$pip install matplotlib

$pip install pandas

$pip install seaborn

**Method 2**

Alternatively, to install Python and other scientific computing and machine learning packages simultaneously, we should install **Anaconda** distribution. It is a Python implementation for Linux, Windows and OSX, and comprises various machine learning packages like numpy, scikit-learn, and matplotlib. It also includes **Jupyter Notebook**, an interactive Python environment. We can install Python 2.7 or any 3.x version as per our requirement.

To download the free Anaconda Python distribution from Continuum Analytics, you can do the following −

Visit the official site of Continuum Analytics and its download page. Note that the installation process may take 15-20 minutes as the installer contains Python, associated packages, a code editor, and some other files. Depending on your operating system, choose the installation process as explained here −

**For Windows** − Select the **Anaconda for Windows** section and look in the column with Python 2.7 or 3.x. You can find that there are two versions of the installer, one for 32-bit Windows, and one for 64-bit Windows. Choose the relevant one.

**For Mac OS** − Scroll to the **Anaconda for OS X section**. Look in the column with Python 2.7 or 3.x. Note that here there is only one version of the installer: the 64-bit version.

**For Linux OS** − We select the "Anaconda for Linux" section. Look in the column with Python 2.7 or 3.x.

Note that you have to ensure that Anaconda’s Python distribution installs into a single directory, and does not affect other Python installations, if any, on your system.

To work with graphs and plots, we will need these Python library packages - **matplotlib** and **seaborn**.

If you are using Anaconda Python, your system already has numpy, matplotlib, pandas, seaborn, etc. installed. We start the Anaconda Navigator to access either Jupyter Note book or Spyder IDE of python.

After opening either of them, type the following commands −

import numpy

import matplotlib

Now, we need to check if installation is successful. For this, go to the command line and type in the following command −

$ python

Python 3.6.3 |Anaconda custom (32-bit)| (default, Oct 13 2017, 14:21:34)

[GCC 7.2.0] on linux

Next, you can import the required libraries and print their versions as shown −

>>>import numpy

>>>print numpy.\_\_version\_\_

1.14.2

>>> import matplotlib

>>> print (matplotlib.\_\_version\_\_)

2.1.2

>> import pandas

>>> print (pandas.\_\_version\_\_)

0.22.0

>>> import seaborn

>>> print (seaborn.\_\_version\_\_)

0.8.1

**Machine Learning (ML)** is an automated learning with little or no human intervention. It involves programming computers so that they learn from the available inputs. The main purpose of machine learning is to explore and construct algorithms that can learn from the previous data and make predictions on new input data.

The **input** to a learning algorithm is training data, representing experience, and the **output** is any expertise, which usually takes the form of another algorithm that can perform a task. The input data to a machine learning system can be numerical, textual, audio, visual, or multimedia. The corresponding output data of the system can be a floating-point number, for instance, the velocity of a rocket, an integer representing a category or a class, for example, a pigeon or a sunflower from image recognition.

In this chapter, we will learn about the training data our programs will access and how learning process is automated and how the success and performance of such machine learning algorithms is evaluated.

**Concepts of Learning**

Learning is the process of converting experience into expertise or knowledge.

Learning can be broadly classified into three categories, as mentioned below, based on the nature of the learning data and interaction between the learner and the environment.

* Supervised Learning
* Unsupervised Learning
* Semi-supervised Learning

Similarly, there are four categories of machine learning algorithms as shown below −

* Supervised learning algorithm
* Unsupervised learning algorithm
* Semi-supervised learning algorithm
* Reinforcement learning algorithm

However, the most commonly used ones are **supervised** and **unsupervised learning**.

**Supervised Learning**

Supervised learning is commonly used in real world applications, such as face and speech recognition, products or movie recommendations, and sales forecasting. Supervised learning can be further classified into two types - **Regression** and **Classification**.

**Regression** trains on and predicts a continuous-valued response, for example predicting real estate prices.

**Classification** attempts to find the appropriate class label, such as analyzing positive/negative sentiment, male and female persons, benign and malignant tumors, secure and unsecure loans etc.

In supervised learning, learning data comes with description, labels, targets or desired outputs and the objective is to find a general rule that maps inputs to outputs. This kind of learning data is called **labeled data**. The learned rule is then used to label new data with unknown outputs.

Supervised learning involves building a machine learning model that is based on **labeled samples**. For example, if we build a system to estimate the price of a plot of land or a house based on various features, such as size, location, and so on, we first need to create a database and label it. We need to teach the algorithm what features correspond to what prices. Based on this data, the algorithm will learn how to calculate the price of real estate using the values of the input features.

Supervised learning deals with learning a function from available training data. Here, a learning algorithm analyzes the training data and produces a derived function that can be used for mapping new examples. There are many **supervised learning algorithms** such as Logistic Regression, Neural networks, Support Vector Machines (SVMs), and Naive Bayes classifiers.

Common **examples** of supervised learning include classifying e-mails into spam and not-spam categories, labeling webpages based on their content, and voice recognition.

**Unsupervised Learning**

Unsupervised learning is used to detect anomalies, outliers, such as fraud or defective equipment, or to group customers with similar behaviors for a sales campaign. It is the opposite of supervised learning. There is no labeled data here.

When learning data contains only some indications without any description or labels, it is up to the coder or to the algorithm to find the structure of the underlying data, to discover hidden patterns, or to determine how to describe the data. This kind of learning data is called **unlabeled data**.

Suppose that we have a number of data points, and we want to classify them into several groups. We may not exactly know what the criteria of classification would be. So, an unsupervised learning algorithm tries to classify the given dataset into a certain number of groups in an optimum way.

Unsupervised learning algorithms are extremely powerful tools for analyzing data and for identifying patterns and trends. They are most commonly used for clustering similar input into logical groups. Unsupervised learning algorithms include Kmeans, Random Forests, Hierarchical clustering and so on.

**Semi-supervised Learning**

If some learning samples are labeled, but some other are not labeled, then it is semi-supervised learning. It makes use of a large amount of **unlabeled data for training** and a small amount of **labeled data for testing**. Semi-supervised learning is applied in cases where it is expensive to acquire a fully labeled dataset while more practical to label a small subset. For example, it often requires skilled experts to label certain remote sensing images, and lots of field experiments to locate oil at a particular location, while acquiring unlabeled data is relatively easy.

**Reinforcement Learning**

Here learning data gives feedback so that the system adjusts to dynamic conditions in order to achieve a certain objective. The system evaluates its performance based on the feedback responses and reacts accordingly. The best known instances include self-driving cars and chess master algorithm AlphaGo.

**Purpose of Machine Learning**

Machine learning can be seen as a branch of AI or Artificial Intelligence, since, the ability to change experience into expertise or to detect patterns in complex data is a mark of human or animal intelligence.

As a field of science, machine learning shares common concepts with other disciplines such as statistics, information theory, game theory, and optimization.

As a subfield of information technology, its objective is to program machines so that they will learn.

However, it is to be seen that, the purpose of machine learning is not building an automated duplication of intelligent behavior, but using the power of computers to complement and supplement human intelligence. For example, machine learning programs can scan and process huge databases detecting patterns that are beyond the scope of human perception.

In the real world, we usually come across lots of raw data which is not fit to be readily processed by machine learning algorithms. We need to preprocess the raw data before it is fed into various machine learning algorithms. This chapter discusses various techniques for preprocessing data in Python machine learning.

**Data Preprocessing**

In this section, let us understand how we preprocess data in Python.

Initially, open a file with a **.py** extension, for example **prefoo.py** file, in a text editor like notepad.

Then, add the following piece of code to this file −

import numpy as np

from sklearn import preprocessing

#We imported a couple of packages. Let's create some sample data and add the line to this file:

input\_data = np.array([[3, -1.5, 3, -6.4], [0, 3, -1.3, 4.1], [1, 2.3, -2.9, -4.3]])

We are now ready to operate on this data.

**Preprocessing Techniques**

Data can be preprocessed using several techniques as discussed here −

**Mean removal**

It involves removing the mean from each feature so that it is centered on zero. Mean removal helps in removing any bias from the features.

You can use the following code for mean removal −

data\_standardized = preprocessing.scale(input\_data)

print "\nMean = ", data\_standardized.mean(axis = 0)

print "Std deviation = ", data\_standardized.std(axis = 0)

Now run the following command on the terminal −

$ python prefoo.py

You can observe the following output −

Mean = [ 5.55111512e-17 -3.70074342e-17 0.00000000e+00 -1.85037171e-17]

Std deviation = [1. 1. 1. 1.]

Observe that in the output, mean is almost 0 and the standard deviation is 1.

**Scaling**

The values of every feature in a data point can vary between random values. So, it is important to scale them so that this matches specified rules.

You can use the following code for scaling −

data\_scaler = preprocessing.MinMaxScaler(feature\_range = (0, 1))

data\_scaled = data\_scaler.fit\_transform(input\_data)

print "\nMin max scaled data = ", data\_scaled

Now run the code and you can observe the following output −

Min max scaled data = [ [ 1. 0. 1. 0. ]

                        [ 0. 1. 0.27118644 1. ]

                        [ 0.33333333 0.84444444 0. 0.2 ]

]

Note that all the values have been scaled between the given range.

**Normalization**

Normalization involves adjusting the values in the feature vector so as to measure them on a common scale. Here, the values of a feature vector are adjusted so that they sum up to 1. We add the following lines to the prefoo.py file −

You can use the following code for normalization −

data\_normalized = preprocessing.normalize(input\_data, norm  = 'l1')

print "\nL1 normalized data = ", data\_normalized

Now run the code and you can observe the following output −

L1 normalized data = [  [ 0.21582734 -0.10791367 0.21582734 -0.46043165]

                        [ 0. 0.35714286 -0.1547619 0.48809524]

                        [ 0.0952381 0.21904762 -0.27619048 -0.40952381]

]

Normalization is used to ensure that data points do not get boosted due to the nature of their features.

**Binarization**

Binarization is used to convert a numerical feature vector into a Boolean vector. You can use the following code for binarization −

data\_binarized = preprocessing.Binarizer(threshold=1.4).transform(input\_data)

print "\nBinarized data =", data\_binarized

Now run the code and you can observe the following output −

Binarized data = [[ 1. 0. 1. 0.]

                  [ 0. 1. 0. 1.]

                  [ 0. 1. 0. 0.]

                 ]

This technique is helpful when we have prior knowledge of the data.

**One Hot Encoding**

It may be required to deal with numerical values that are few and scattered, and you may not need to store these values. In such situations you can use One Hot Encoding technique.

If the number of distinct values is ***k***, it will transform the feature into a *k-dimensional* vector where only one value is *1* and all other values are *0*.

You can use the following code for one hot encoding −

encoder = preprocessing.OneHotEncoder()

encoder.fit([  [0, 2, 1, 12],

               [1, 3, 5, 3],

               [2, 3, 2, 12],

               [1, 2, 4, 3]

])

encoded\_vector = encoder.transform([[2, 3, 5, 3]]).toarray()

print "\nEncoded vector =", encoded\_vector

Now run the code and you can observe the following output −

Encoded vector = [[ 0. 0. 1. 0. 1. 0. 0. 0. 1. 1. 0.]]

In the example above, let us consider the third feature in each feature vector. The values are 1, 5, 2, and 4.

There are four separate values here, which means the one-hot encoded vector will be of length 4. If we want to encode the value 5, it will be a vector [0, 1, 0, 0]. Only one value can be 1 in this vector. The second element is 1, which indicates that the value is 5.

**Label Encoding**

In supervised learning, we mostly come across a variety of labels which can be in the form of numbers or words. If they are numbers, then they can be used directly by the algorithm. However, many times, labels need to be in readable form. Hence, the training data is usually labelled with words.

Label encoding refers to changing the word labels into numbers so that the algorithms can understand how to work on them. Let us understand in detail how to perform label encoding −

Create a new Python file, and import the preprocessing package −

from sklearn import preprocessing

label\_encoder = preprocessing.LabelEncoder()

input\_classes = ['suzuki', 'ford', 'suzuki', 'toyota', 'ford', 'bmw']

label\_encoder.fit(input\_classes)

print "\nClass mapping:"

for i, item in enumerate(label\_encoder.classes\_):

print item, '-->', i

Now run the code and you can observe the following output −

Class mapping:

bmw --> 0

ford --> 1

suzuki --> 2

toyota --> 3

As shown in above output, the words have been changed into 0-indexed numbers. Now, when we deal with a set of labels, we can transform them as follows −

labels = ['toyota', 'ford', 'suzuki']

encoded\_labels = label\_encoder.transform(labels)

print "\nLabels =", labels

print "Encoded labels =", list(encoded\_labels)

Now run the code and you can observe the following output −

Labels = ['toyota', 'ford', 'suzuki']

Encoded labels = [3, 1, 2]

This is efficient than manually maintaining mapping between words and numbers. You can check by transforming numbers back to word labels as shown in the code here −

encoded\_labels = [3, 2, 0, 2, 1]

decoded\_labels = label\_encoder.inverse\_transform(encoded\_labels)

print "\nEncoded labels =", encoded\_labels

print "Decoded labels =", list(decoded\_labels)

Now run the code and you can observe the following output −

Encoded labels = [3, 2, 0, 2, 1]

Decoded labels = ['toyota', 'suzuki', 'bmw', 'suzuki', 'ford']

From the output, you can observe that the mapping is preserved perfectly.

**Data Analysis**

This section discusses data analysis in Python machine learning in detail −

**Loading the Dataset**

We can load the data directly from the UCI Machine Learning repository. Note that here we are using **pandas** to load the data. We will also use pandas next to explore the data both with descriptive statistics and data visualization. Observe the following code and note that we are specifying the names of each column when loading the data.

import pandas

data = ‘pima\_indians.csv’

names = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', ‘Outcome’]

dataset = pandas.read\_csv(data, names = names)

When you run the code, you can observe that the dataset loads and is ready to be analyzed. Here, we have downloaded the pima\_indians.csv file and moved it into our working directory and loaded it using the local file name.

**Summarizing the Dataset**

Summarizing the data can be done in many ways as follows −

* Check dimensions of the dataset
* List the entire data
* View the statistical summary of all attributes
* Breakdown of the data by the class variable

**Dimensions of Dataset**

You can use the following command to check how many instances (rows) and attributes (columns) the data contains with the shape property.

print(dataset.shape)

Then, for the code that we have discussed, we can see 769 instances and 6 attributes −

(769, 6)

**List the Entire Data**

You can view the entire data and understand its summary −

print(dataset.head(20))

This command prints the first 20 rows of the data as shown −

Sno Pregnancies Glucose BloodPressure SkinThickness Insulin Outcome

1        6        148         72           35          0       1

2        1         85         66           29          0       0

3        8        183         64            0          0       1

4        1         89         66           23         94       0

5        0        137         40           35        168       1

6        5        116         74            0          0       0

7        3         78         50           32         88       1

8       10        115          0            0          0       0

9        2        197         70           45        543       1

10       8        125         96            0          0       1

11       4        110         92            0          0       0

12      10        168         74            0          0       1

13      10        139         80            0          0       0

14       1        189         60           23        846       1

15       5        166         72           19        175       1

16       7        100          0            0          0       1

17       0        118         84           47        230       1

18       7        107         74            0          0       1

19       1        103         30           38         83       0

**View the Statistical Summary**

You can view the statistical summary of each attribute, which includes the count, unique, top and freq, by using the following command.

print(dataset.describe())

The above command gives you the following output that shows the statistical summary of each attribute −

         Pregnancies Glucose BloodPressur SkinThckns Insulin Outcome

count       769       769       769         769       769     769

unique       18       137        48          52       187       3

top           1       100        70           0         0       0

freq        135        17        57         227       374     500

**Breakdown the Data by Class Variable**

You can also look at the number of instances (rows) that belong to each outcome as an absolute count, using the command shown here −

print(dataset.groupby('Outcome').size())

Then you can see the number of outcomes of instances as shown −

Outcome

0         500

1         268

Outcome     1

dtype: int64

**Data Visualization**

You can visualize data using two types of plots as shown −

        Univariate plots to understand each attribute

        Multivariate plots to understand the relationships between attributes

**Univariate Plots**

Univariate plots are plots of each individual variable. Consider a case where the input variables are numeric, and we need to create box and whisker plots of each. You can use the following code for this purpose.

import pandas

import matplotlib.pyplot as plt

data = 'iris\_df.csv'

names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']

dataset = pandas.read\_csv(data, names=names)

dataset.plot(kind='box', subplots=True, layout=(2,2), sharex=False, sharey=False)

plt.show()

You can see the output with a clearer idea of the distribution of the input attributes as shown −

**Box and Whisker Plots**

You can create a histogram of each input variable to get an idea of the distribution using the commands shown below −

#histograms

dataset.hist()

plt().show()

From the output, you can see that two of the input variables have a Gaussian distribution. Thus these plots help in giving an idea about the algorithms that we can use in our program.

**Multivariate Plots**

Multivariate plots help us to understand the interactions between the variables.

**Scatter Plot Matrix**

First, let’s look at scatterplots of all pairs of attributes. This can be helpful to spot structured relationships between input variables.

from pandas.plotting import scatter\_matrix

scatter\_matrix(dataset)

plt.show()

You can observe the output as shown −

Observe that in the output there is a diagonal grouping of some pairs of attributes. This indicates a high correlation and a predictable relationship.

**Training Data**

The observations in the training set form the experience that the algorithm uses to learn. In supervised learning problems, each observation consists of an observed output variable and one or more observed input variables.

**Test Data**

The test set is a set of observations used to evaluate the performance of the model using some performance metric. It is important that no observations from the training set are included in the test set. If the test set does contain examples from the training set, it will be difficult to assess whether the algorithm has learned to generalize from the training set or has simply memorized it.

A program that generalizes well will be able to effectively perform a task with new data. In contrast, a program that memorizes the training data by learning an overly complex model could predict the values of the response variable for the training set accurately, but will fail to predict the value of the response variable for new examples. Memorizing the training set is called **over-fitting**. A program that memorizes its observations may not perform its task well, as it could memorize relations and structures that are noise or coincidence. Balancing memorization and generalization, or over-fitting and under-fitting, is a problem common to many machine learning algorithms. **Regularization**may be applied to many models to reduce over-fitting.

In addition to the training and test data, a third set of observations, called a **validation** or **hold-out set**, is sometimes required. The validation set is used to tune variables called **hyper parameters**, which control how the model is learned. The program is still evaluated on the test set to provide an estimate of its performance in the real world; its performance on the validation set should not be used as an estimate of the model's real-world performance since the program has been tuned specifically to the validation data. It is common to partition a single set of supervised observations into training, validation, and test sets. There are no requirements for the sizes of the partitions, and they may vary according to the amount of data available. It is common to allocate 50 percent or more of the data to the training set, 25 percent to the test set, and the remainder to the validation set.

Some training sets may contain only a few hundred observations; others may include millions. Inexpensive storage, increased network connectivity, the ubiquity of sensor-packed smartphones, and shifting attitudes towards privacy have contributed to the contemporary state of big data, or training sets with millions or billions of examples.

However, machine learning algorithms also follow the maxim "garbage in, garbage out." A student who studies for a test by reading a large, confusing textbook that contains many errors will likely not score better than a student who reads a short but well-written textbook. Similarly, an algorithm trained on a large collection of noisy, irrelevant, or incorrectly labeled data will not perform better than an algorithm trained on a smaller set of data that is more representative of problems in the real world.

Many supervised training sets are prepared manually, or by semi-automated processes. Creating a large collection of supervised data can be costly in some domains. Fortunately, several datasets are bundled with **scikit-learn**, allowing developers to focus on experimenting with models instead.

During development, and particularly when training data is scarce, a practice called **cross-validation** can be used to train and validate an algorithm on the same data. In cross-validation, the training data is partitioned. The algorithm is trained using all but one of the partitions, and tested on the remaining partition. The partitions are then rotated several times so that the algorithm is trained and evaluated on all of the data.

Consider for example that the original dataset is partitioned into five subsets of equal size, labeled A through E. Initially, the model is trained on partitions B through E, and tested on partition A. In the next iteration, the model is trained on partitions A, C, D, and E, and tested on partition B. The partitions are rotated until models have been trained and tested on all of the partitions. Cross-validation provides a more accurate estimate of the model's performance than testing a single partition of the data.

**Performance Measures − Bias and Variance**

Many metrics can be used to measure whether or not a program is learning to perform its task more effectively. For supervised learning problems, many performance metrics measure the number of prediction errors.

There are two fundamental causes of prediction error for a model -**bias** and **variance**. Assume that you have many training sets that are all unique, but equally representative of the population. A model with a high bias will produce similar errors for an input regardless of the training set it was trained with; the model biases its own assumptions about the real relationship over the relationship demonstrated in the training data. A model with high variance, conversely, will produce different errors for an input depending on the training set that it was trained with. A model with high bias is inflexible, but a model with high variance may be so flexible that it models the noise in the training set. That is, a model with high variance over-fits the training data, while a model with high bias under-fits the training data.

Ideally, a model will have both low bias and variance, but efforts to decrease one will frequently increase the other. This is known as the **bias-variance trade-off**. We may have to consider the bias-variance tradeoffs of several models introduced in this tutorial. Unsupervised learning problems do not have an error signal to measure; instead, performance metrics for unsupervised learning problems measure some attributes of the structure discovered in the data. Most performance measures can only be worked out for a specific type of task.

Machine learning systems should be evaluated using performance measures that represent the costs of making errors in the real world. While this looks trivial, the following example illustrates the use of a performance measure that is right for the task in general but not for its specific application.

**Accuracy, Precision and Recall**

Consider a classification task in which a machine learning system observes tumors and has to predict whether these tumors are benign or malignant. **Accuracy**, or the fraction of instances that were classified correctly, is an obvious measure of the program's performance. While accuracy does measure the program's performance, it does not make distinction between malignant tumors that were classified as being benign, and benign tumors that were classified as being malignant. In some applications, the costs incurred on all types of errors may be the same. In this problem, however, failing to identify malignant tumors is a more serious error than classifying benign tumors as being malignant by mistake.

We can measure each of the possible prediction outcomes to create different snapshots of the classifier's performance. When the system correctly classifies a tumor as being malignant, the prediction is called a **true positive**. When the system incorrectly classifies a benign tumor as being malignant, the prediction is a **false positive**. Similarly, a **false negative** is an incorrect prediction that the tumor is benign, and a **true negative** is a correct prediction that a tumor is benign. These four outcomes can be used to calculate several common measures of classification performance, like accuracy, precision, recall and so on.

Accuracy is calculated with the following formula −

**ACC = (TP + TN)/(TP + TN + FP + FN)**

Where, TP is the number of true positives

TN is the number of true negatives

FP is the number of false positives

FN is the number of false negatives.

**Precision** is the fraction of the tumors that were predicted to be malignant that are actually malignant. Precision is calculated with the following formula −

**PREC = TP/(TP + FP)**

**Recall** is the fraction of malignant tumors that the system identified. Recall is calculated with the following formula −

**R = TP/(TP + FN)**

In this example, precision measures the fraction of tumors that were predicted to be malignant that are actually malignant. Recall measures the fraction of truly malignant tumors that were detected. The precision and recall measures could reveal that a classifier with impressive accuracy actually fails to detect most of the malignant tumors. If most tumors are benign, even a classifier that never predicts malignancy could have high accuracy. A different classifier with lower accuracy and higher recall might be better suited to the task, since it will detect more of the malignant tumors. Many other performance measures for classification can also be used.