**Difference between Machine learning and Artificial Intelligence**

Artificial Intelligence and Machine Learning are the terms of computer science. This article discusses some points on the basis of which we can differentiate between these two terms.

**Overview**

[**Artificial Intelligence**](https://www.geeksforgeeks.org/artificial-intelligence-an-introduction/): The word Artificial Intelligence comprises of two words “Artificial” and “Intelligence”. Artificial refers to something which is made by human or non natural thing and Intelligence means ability to understand or think. There is a misconception that Artificial Intelligence is a system, but it is not a system .AI is implemented in the system. There can be so many definition of AI, one definition can be ***“It is the study of how to train the computers so that computers can do things which at present human can do better.”***Therefore It is a intelligence where we want to add all the capabilities to machine that human contain.

[**Machine Learning**](https://www.geeksforgeeks.org/getting-started-machine-learning/) : Machine Learning is the learning in which machine can learn by its own without being explicitly programmed. It is an application of AI that provide system the ability to automatically learn and improve from experience. Here we can generate a program by integrating input and output of that program. One of the simple definition of the Machine Learning is ***“Machine Learning is said to learn from experience E w.r.t some class of task T and a performance measure P if learners performance at the task in the class as measured by P improves with experiences.”***

**The key difference between AI and ML are:**

|  |  |
| --- | --- |
| **ARTIFICIAL INTELLIGENCE** | **MACHINE LEARNING** |
| AI stands for Artificial intelligence, where intelligence is defined acquisition of knowledge intelligence is defined as a ability to acquire and apply knowledge. | ML stands for Machine Learning which is defined as the acquisition of knowledge or skill |
| The aim is to increase chance of success and not accuracy. | The aim is to increase accuracy, but it does not care about success |
| It work as a computer program that does smart work | It is a simple concept machine takes data and learn from data. |
| The goal is to simulate natural intelligence to solve complex problem | The goal is to learn from data on certain task to maximize the performance of machine on this task. |
| AI is decision making. | ML allows system to learn new things from data. |
| It leads to develop a system to mimic human to respond behave in a circumstances. | It involves in creating self learning algorithms. |
| AI will go for finding the optimal solution. | ML will go for only solution for that whether it is optimal or not. |
| AI leads to intelligence or wisdom. | ML leads to knowledge. |

Finding Correlation Between Many Variables (Multidimensional Dataset) with Python

I will find correlation on a dataset with the following 19 columns (features/attributes) and 1000 rows (samples/observations/instances):

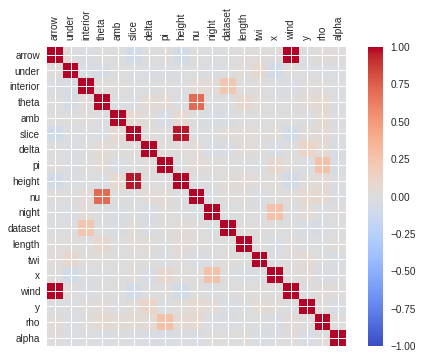
[‘arrow’, ‘under’, ‘interior’, ‘theta’, ‘amb’, ‘slice’, ‘delta’, ‘pi’, ‘height’, ‘nu’, ‘night’, ‘dataset’, ‘length’, ‘twi’, ‘x’, ‘wind’, ‘y’, ‘rho’, ‘alpha’]

The “corr()” method evaluates the correlation between all the features, then it can be graphed with a color coding:

import numpy as np  
import pandas as pd  
import matplotlib.pyplot as plt

data = pd.read\_csv('[https://www.dropbox.com/s/4jgheggd1dak5pw/data\_visualization.csv?raw=1'](https://www.dropbox.com/s/4jgheggd1dak5pw/data_visualization.csv?raw=1%27), index\_col=0)

corr = data.corr()  
fig = plt.figure()  
ax = fig.add\_subplot(111)  
cax = ax.matshow(corr,cmap='coolwarm', vmin=-1, vmax=1)  
fig.colorbar(cax)  
ticks = np.arange(0,len(data.columns),1)  
ax.set\_xticks(ticks)  
plt.xticks(rotation=90)  
ax.set\_yticks(ticks)  
ax.set\_xticklabels(data.columns)  
ax.set\_yticklabels(data.columns)  
plt.show()



Correlation between variables of the dataset

On this example, when there is no correlation between 2 variables (when correlation is 0 or near 0) the color is gray. The darkest red means there is a perfect positive correlation, while the darkest blue means there is a perfect negative correlation.

When evaluating the correlation between all the features, the The “corr()” method includes the correlation of each feature with itself, which is always 1, so that is why this type of graph always has the red diagonal from the upper left to the lower right. Other than the diagonal, the rest of the squares show correlation between different features, making it really easy to find that “wind” and “arrow” are highly correlated, “height” and “slice” are highly correlated, “nu” and “theta” are have a correlation of about 0.5.

Conclusion: the “corr()” is very easy to use and very powerful for the early stages of data analysis (data preparation), by doing a graph of its results using matplotlib or any other python plotting utility, you will get a better idea of the data so you can make decisions for the next steps of data preparation and data analysis.

Linear Regression (Python Implementation)

Linear regression is a statistical approach for modelling relationship between a dependent variable with a given set of independent variables.

**Note:** In this article, we refer dependent variables as **response** and independent variables as **features** for simplicity.

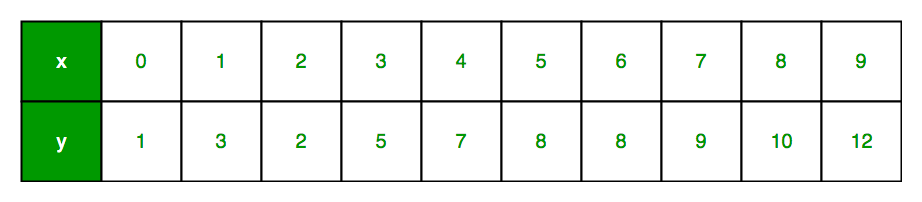
In order to provide a basic understanding of linear regression, we start with the most basic version of linear regression, i.e. **Simple linear regression**.

**Simple Linear Regression**

Simple linear regression is an approach for predicting a **response** using a **single feature**.

It is assumed that the two variables are linearly related. Hence, we try to find a linear function that predicts the response value(y) as accurately as possible as a function of the feature or independent variable(x).

Let us consider a dataset where we have a value of response y for every feature x:



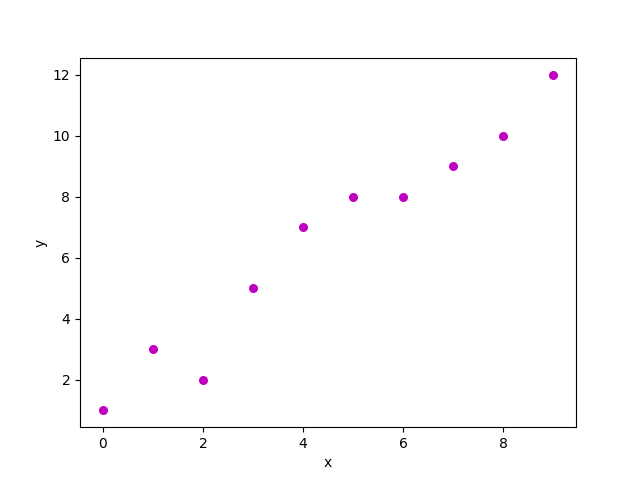
For generality, we define:

x as **feature vector**, i.e x = [x\_1, x\_2, …., x\_n],

y as **response vector**, i.e y = [y\_1, y\_2, …., y\_n]

for **n** observations (in above example, n=10).

A scatter plot of above dataset looks like:-



Now, the task is to find a **line which fits best** in above scatter plot so that we can predict the response for any new feature values. (i.e a value of x not present in dataset)

This line is called **regression line**.

The equation of regression line is represented as:

Here,

* h(x\_i) represents the **predicted response value** for ith observation.
* b\_0 and b\_1 are regression coefficients and represent **y-intercept** and **slope** of regression line respectively.

To create our model, we must “learn” or estimate the values of regression coefficients b\_0 and b\_1. And once we’ve estimated these coefficients, we can use the model to predict responses!

In this article, we are going to use the **Least Squares technique**.

Now consider:

Here, e\_i is **residual error** in ith observation.  
So, our aim is to minimize the total residual error.

We define the squared error or cost function, J as:

and our task is to find the value of b\_0 and b\_1 for which J(b\_0,b\_1) is minimum!

Without going into the mathematical details, we present the result here:

where SS\_xy is the sum of cross-deviations of y and x:

and SS\_xx is the sum of squared deviations of x:

Note: The complete derivation for finding least squares estimates in simple linear regression can be found [here](https://www.amherst.edu/system/files/media/1287/SLR_Leastsquares.pdf).

import numpy as np

import matplotlib.pyplot as plt

def estimate\_coef(x, y):

    # number of observations/points

    n = np.size(x)

    # mean of x and y vector

    m\_x, m\_y = np.mean(x), np.mean(y)

    # calculating cross-deviation and deviation about x

    SS\_xy = np.sum(y\*x - n\*m\_y\*m\_x)

    SS\_xx = np.sum(x\*x - n\*m\_x\*m\_x)

    # calculating regression coefficients

    b\_1 = SS\_xy / SS\_xx

    b\_0 = m\_y - b\_1\*m\_x

    return(b\_0, b\_1)

def plot\_regression\_line(x, y, b):

    # plotting the actual points as scatter plot

    plt.scatter(x, y, color = "m",

               marker = "o", s = 30)

    # predicted response vector

    y\_pred = b[0] + b[1]\*x

    # plotting the regression line

    plt.plot(x, y\_pred, color = "g")

    # putting labels

    plt.xlabel('x')

    plt.ylabel('y')

    # function to show plot

    plt.show()

def main():

    # observations

    x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

    y = np.array([1, 3, 2, 5, 7, 8, 8, 9, 10, 12])

    # estimating coefficients

    b = estimate\_coef(x, y)

    print("Estimated coefficients:\nb\_0 = {}  \

          \nb\_1 = {}".format(b[0], b[1]))

    # plotting regression line

    plot\_regression\_line(x, y, b)

if \_\_name\_\_ == "\_\_main\_\_":

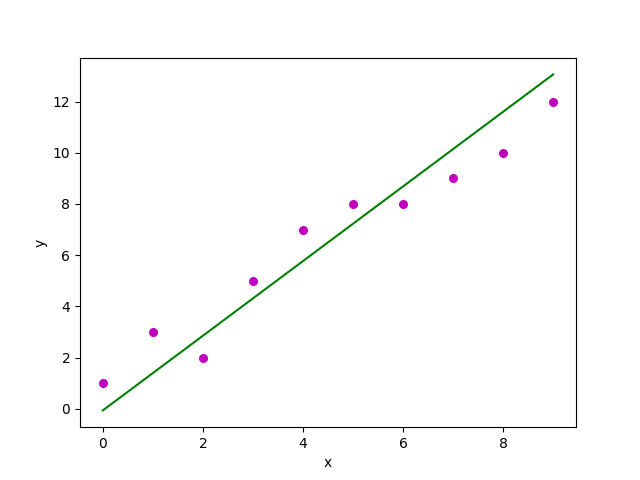
    main()

Output of above piece of code is:

Estimated coefficients:

b\_0 = -0.0586206896552

b\_1 = 1.45747126437

And graph obtained looks like this:  


**Multiple linear regression**

Multiple linear regression attempts to model the relationship between **two or more features** and a response by fitting a linear equation to observed data.

Clearly, it is nothing but an extension of Simple linear regression.

Consider a dataset with **p** features(or independent variables) and one response(or dependent variable).  
Also, the dataset contains **n** rows/observations.

We define:

X (**feature matrix**) = a matrix of size **n X p** where x\_{ij} denotes the values of jth feature for ith observation.

So,

and

y (**response vector**) = a vector of size **n** where y\_{i} denotes the value of response for ith observation.

The **regression line** for **p** features is represented as:  
  
where h(x\_i) is **predicted response value** for ith observation and b\_0, b\_1, …, b\_p are the **regression coefficients**.

Also, we can write:

where e\_i represents **residual error** in ith observation.

We can generalize our linear model a little bit more by representing feature matrix **X** as:  
  
So now, the linear model can be expressed in terms of matrices as:

where,

and

Now, we determine **estimate of b**, i.e. b’ using **Least Squares method**.

As already explained, Least Squares method tends to determine b’ for which total residual error is minimized.

We present the result directly here:  
  
where ‘ represents the transpose of the matrix while -1 represents the matrix inverse.

Knowing the least square estimates, b’, the multiple linear regression model can now be estimated as:

where y’ is **estimated response vector**.

**Note:** The complete derivation for obtaining least square estimates in multiple linear regression can be found [here](https://isites.harvard.edu/fs/docs/icb.topic515975.files/OLSDerivation.pdf).

Given below is the implementation of multiple linear regression technique on the [Boston house pricing dataset](https://archive.ics.uci.edu/ml/datasets/Housing) dataset using Scikit-learn.

import matplotlib.pyplot as plt

import numpy as np

from sklearn import datasets, linear\_model, metrics

# load the boston dataset

boston = datasets.load\_boston(return\_X\_y=False)

# defining feature matrix(X) and response vector(y)

X = boston.data

y = boston.target

# splitting X and y into training and testing sets

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.4,

                                                    random\_state=1)

# create linear regression object

reg = linear\_model.LinearRegression()

# train the model using the training sets

reg.fit(X\_train, y\_train)

# regression coefficients

print('Coefficients: \n', reg.coef\_)

# variance score: 1 means perfect prediction

print('Variance score: {}'.format(reg.score(X\_test, y\_test)))

# plot for residual error

## setting plot style

plt.style.use('fivethirtyeight')

## plotting residual errors in training data

plt.scatter(reg.predict(X\_train), reg.predict(X\_train) - y\_train,

            color = "green", s = 10, label = 'Train data')

## plotting residual errors in test data

plt.scatter(reg.predict(X\_test), reg.predict(X\_test) - y\_test,

            color = "blue", s = 10, label = 'Test data')

## plotting line for zero residual error

plt.hlines(y = 0, xmin = 0, xmax = 50, linewidth = 2)

## plotting legend

plt.legend(loc = 'upper right')

## plot title

plt.title("Residual errors")

## function to show plot

plt.show()

Output of above program looks like this:

Coefficients:

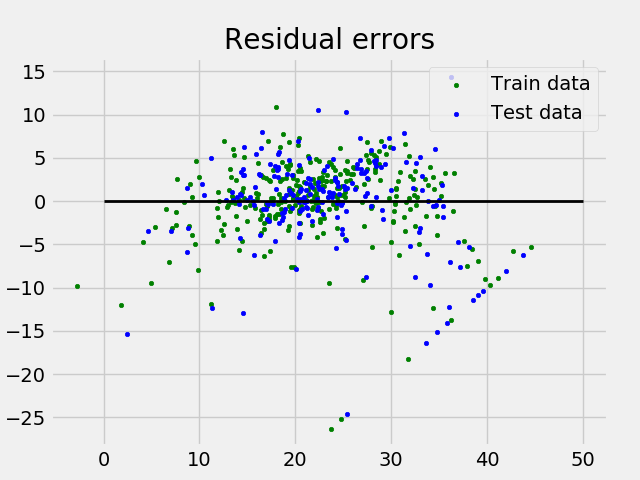
[ -8.80740828e-02 6.72507352e-02 5.10280463e-02 2.18879172e+00

-1.72283734e+01 3.62985243e+00 2.13933641e-03 -1.36531300e+00

2.88788067e-01 -1.22618657e-02 -8.36014969e-01 9.53058061e-03

-5.05036163e-01]

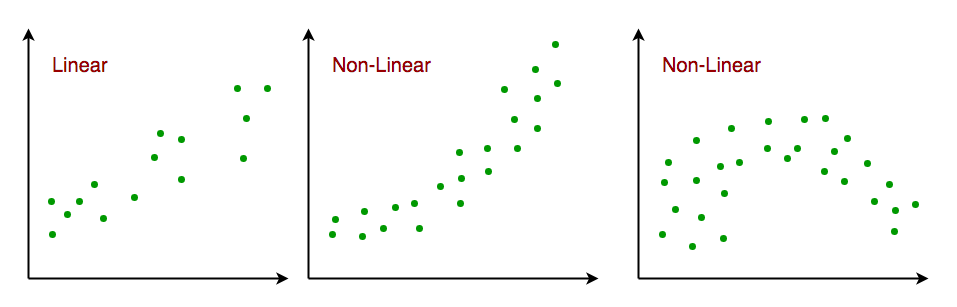
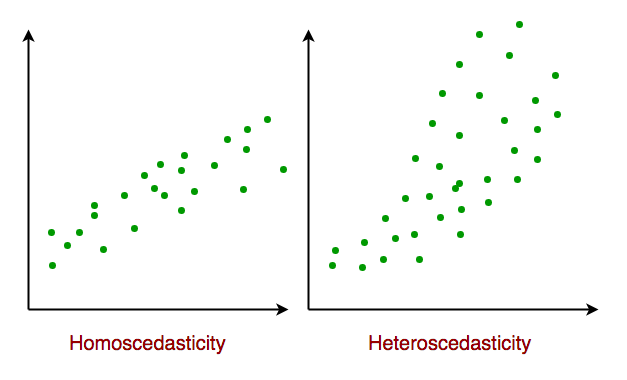
Variance score: 0.720898784611

and **Residual Error plot** looks like this:  


In above example, we determine accuracy score using **Explained Variance Score**.  
We define:  
explained\_variance\_score = 1 – Var{y – y’}/Var{y}  
where y’ is the estimated target output, y the corresponding (correct) target output, and Var is Variance, the square of the standard deviation.  
The best possible score is 1.0, lower values are worse.

**Assumptions**

Given below are the basic assumptions that a linear regression model makes regarding a dataset on which it is applied:

* **Linear relationship**: Relationship between response and feature variables should be linear. The linearity assumption can be tested using scatter plots. As shown below, 1st figure represents linearly related variables where as variables in 2nd and 3rd figure are most likely non-linear. So, 1st figure will give better predictions using linear regression.  
  
* **Little or no multi-collinearity**: It is assumed that there is little or no multicollinearity in the data. Multicollinearity occurs when the features (or independent variables) are not independent from each other.
* **Little or no auto-correlation**: Another assumption is that there is little or no autocorrelation in the data. Autocorrelation occurs when the residual errors are not independent from each other. You can refer [here](https://en.wikipedia.org/wiki/Autocorrelation#Regression_analysis) for more insight into this topic.
* **Homoscedasticity**: Homoscedasticity describes a situation in which the error term (that is, the “noise” or random disturbance in the relationship between the independent variables and the dependent variable) is the same across all values of the independent variables. As shown below, figure 1 has homoscedasticity while figure 2 has heteroscedasticity.  
  

As we reach to the end of this article, we discuss some applications of linear regression below.

**Applications:**

**1. Trend lines:** A trend line represents the variation in some quantitative data with passage of time (like GDP, oil prices, etc.). These trends usually follow a linear relationship. Hence, linear regression can be applied to predict future values. However, this method suffers from a lack of scientific validity in cases where other potential changes can affect the data.

**2. Economics:** Linear regression is the predominant empirical tool in economics. For example, it is used to predict consumption spending, fixed investment spending, inventory investment, purchases of a country’s exports, spending on imports, the demand to hold liquid assets, labor demand, and labor supply.

**3. Finance:** Capital price asset model uses linear regression to analyze and quantify the systematic risks of an investment.

Linear Regression Using Tensorflow

**Tensorflow**

Tensorflow is an open-source computation library made by Google. It is a popular choice for creating applications that require high end numerical computations and/or need to utilize Graphics Processing Units for the computation purposes. These are the main reasons due to which Tensorflow is one of the most popular choices for Machine Learning applicaations, especially Deep Learning. It also has APIs like Estimator which provide high level of abstraction which building Machine Learning Applications. In this article, we will not be using any high level APIs, rather we will be building the Linear Regression model using low level Tensorflow in the Lazy Exceution Mode during which Tensorflow creates a **Directed Acyclic Graph** or DAG which keeps track of all the computations, and then executes all the computations done inside a **Tensorflow Session**.

**Implementation**

We will start by importing the necessary libraries. We will use **Numpy** along with Tensorflow for computations and **Matplotlib** for plotting.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| import numpy as np  import tensorflow as tf  import matplotlib.pyplot as plt |

***chevron\_right***

*filter\_none*

In order to make the random numbers predictable, we will define fixed seeds for both Numpy and Tensorflow.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| np.random.seed(101)  tf.set\_random\_seed(101) |

***chevron\_right***

*filter\_none*

Now, let us generate some random data for training the Linear Regression Model.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Genrating random linear data  # There will be 50 data points ranging from 0 to 50  x = np.linspace(0, 50, 50)  y = np.linspace(0, 50, 50)    # Adding noise to the random linear data  x += np.random.uniform(-4, 4, 50)  y += np.random.uniform(-4, 4, 50)    n = len(x) # Number of data points |

***chevron\_right***

*filter\_none*

Let us visualize the training data.

*filter\_none*

*edit*  
*close*

*play\_arrow*

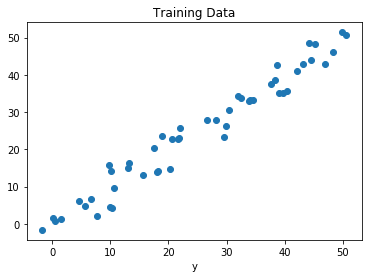
*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Plot of Training Data  plt.scatter(x, y)  plt.xlabel('x')  plt.xlabel('y')  plt.title("Training Data")  plt.show() |

***chevron\_right***

*filter\_none*

**Output:**



Now we will start creating our model by defining the **placeholders** X and Y, so that we can feed our training examples X and Y into the **optimizer** during the training process.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| X = tf.placeholder("float")  Y = tf.placeholder("float") |

***chevron\_right***

*filter\_none*

Now we will declare two trainable Tensorflow **Variables** for the Weights and Bias and initializing them randomly using np.random.randn().

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| W = tf.Variable(np.random.randn(), name = "W")  b = tf.Variable(np.random.randn(), name = "b") |

***chevron\_right***

*filter\_none*

Now we will define the hyperparameters of the model, the Learning Rate and the number of Epochs.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| learning\_rate = 0.01  training\_epochs = 1000 |

***chevron\_right***

*filter\_none*

Now, we will be building the Hypothesis, the Cost Function and the Optimizer. We won’t be implementing the Gradient Descent Optimizer manually since it is built inside Tensorflow. After that, we will be initializing the Variables.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Hypothesis  y\_pred = tf.add(tf.multiply(X, W), b)    # Mean Squared Error Cost Function  cost = tf.reduce\_sum(tf.pow(y\_pred-Y, 2)) / (2 \* n)    # Gradient Descent Optimizer  optimizer = tf.train.GradientDescentOptimizer(learning\_rate).minimize(cost)    # Global Variables Initializer  init = tf.global\_variables\_initializer() |

***chevron\_right***

*filter\_none*

Now we will begin the training process inside a Tensorflow Session.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Starting the Tensorflow Session  with tf.Session() as sess:        # Initializing the Variables      sess.run(init)        # Iterating through all the epochs      for epoch in range(training\_epochs):            # Feeding each data point into the optimizer using Feed Dictionary          for (\_x, \_y) in zip(x, y):              sess.run(optimizer, feed\_dict = {X : \_x, Y : \_y})            # Displaying the result after every 50 epochs          if (epoch + 1) % 50 == 0:              # Calculating the cost a every epoch              c = sess.run(cost, feed\_dict = {X : x, Y : y})              print("Epoch", (epoch + 1), ": cost =", c, "W =", sess.run(W), "b =", sess.run(b))        # Storing necessary values to be used outside the Session      training\_cost = sess.run(cost, feed\_dict ={X: x, Y: y})      weight = sess.run(W)      bias = sess.run(b) |

***chevron\_right***

*filter\_none*

**Output:**

Epoch: 50 cost = 5.8868036 W = 0.9951241 b = 1.2381054

Epoch: 100 cost = 5.7912707 W = 0.99812365 b = 1.0914398

Epoch: 150 cost = 5.7119675 W = 1.0008028 b = 0.96044314

Epoch: 200 cost = 5.6459413 W = 1.0031956 b = 0.8434396

Epoch: 250 cost = 5.590799 W = 1.0053328 b = 0.7389357

Epoch: 300 cost = 5.544608 W = 1.007242 b = 0.6455922

Epoch: 350 cost = 5.5057883 W = 1.008947 b = 0.56222

Epoch: 400 cost = 5.473066 W = 1.01047 b = 0.48775345

Epoch: 450 cost = 5.4453845 W = 1.0118302 b = 0.42124167

Epoch: 500 cost = 5.421903 W = 1.0130452 b = 0.36183488

Epoch: 550 cost = 5.4019217 W = 1.0141305 b = 0.30877414

Epoch: 600 cost = 5.3848577 W = 1.0150996 b = 0.26138115

Epoch: 650 cost = 5.370246 W = 1.0159653 b = 0.21905091

Epoch: 700 cost = 5.3576994 W = 1.0167387 b = 0.18124212

Epoch: 750 cost = 5.3468933 W = 1.0174294 b = 0.14747244

Epoch: 800 cost = 5.3375573 W = 1.0180461 b = 0.11730931

Epoch: 850 cost = 5.3294764 W = 1.0185971 b = 0.090368524

Epoch: 900 cost = 5.322459 W = 1.0190892 b = 0.0663058

Epoch: 950 cost = 5.3163586 W = 1.0195289 b = 0.044813324

Epoch: 1000 cost = 5.3110332 W = 1.0199214 b = 0.02561663

Now let us look at the result.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Calculating the predictions  predictions = weight \* x + bias  print("Training cost =", training\_cost, "Weight =", weight, "bias =", bias, '\n') |

***chevron\_right***

*filter\_none*

**Output:**

Training cost = 5.3110332 Weight = 1.0199214 bias = 0.02561663

Note that in this case both the Weight and bias are scalars. This is because, we have considered only one dependent variable in out training data. If we have m dependent variables in our training dataset, the Weight will be an m-dimensional vector while bias will be a scalar.

Finally, we will plot our result.

*filter\_none*

*edit*  
*close*

*play\_arrow*

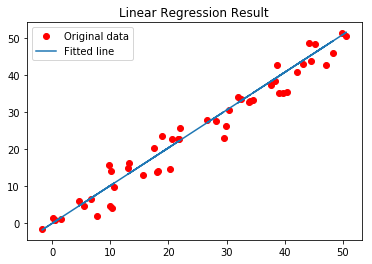
*link*  
*brightness\_4*  
*code*

|  |
| --- |
| # Plotting the Results  plt.plot(x, y, 'ro', label ='Original data')  plt.plot(x, predictions, label ='Fitted line')  plt.title('Linear Regression Result')  plt.legend()  plt.show() |

***chevron\_right***

*filter\_none*

**Output:**



**K means Clustering – Introduction**

We are given a data set of items, with certain features, and values for these features (like a vector). The task is to categorize those items into groups. To achieve this, we will use the kMeans algorithm; an unsupervised learning algorithm.

The algorithm works as follows:

1. First we initialize k points, called means, randomly.
2. We categorize each item to its closest mean and we update the mean’s coordinates, which are the averages of the items categorized in that mean so far.
3. We repeat the process for a given number of iterations and at the end, we have our clusters.

The “points” mentioned above are called means, because they hold the mean values of the items categorized in it. To initialize these means, we have a lot of options. An intuitive method is to initialize the means at random items in the data set. Another method is to initialize the means at random values between the boundaries of the data set (if for a feature *x* the items have values in [0,3], we will initialize the means with values for *x* at [0,3]).

The above algorithm in pseudocode:

Initialize k means with random values

For a given number of iterations:

Iterate through items:

Find the mean closest to the item

Assign item to mean

Update mean

We receive input as a text file (‘data.txt’). Each line represents an item, and it contains numerical values (one for each feature) split by commas. You can find a sample data set [here.](https://github.com/MrDupin/Machine-Learning/blob/master/Clustering/kMeans%20-%20Standard/data.txt)

We will read the data from the file, saving it into a list. Each element of the list is another list containing the item values for the features. We do this with the following function:

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def ReadData(fileName):        # Read the file, splitting by lines      f = open(fileName, 'r');      lines = f.read().splitlines();      f.close();        items = [];        for i in range(1, len(lines)):          line = lines[i].split(',');          itemFeatures = [];            for j in range(len(line)-1):              v = float(line[j]); # Convert feature value to float              itemFeatures.append(v); # Add feature value to dict            items.append(itemFeatures);        shuffle(items);        return items; |

***chevron\_right***

*filter\_none*

**Initialize Means**

We want to initialize each mean’s values in the range of the feature values of the items. For that, we need to find the min and max for each feature. We accomplish that with the following function:

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def FindColMinMax(items):      n = len(items[0]);      minima = [sys.maxint for i in range(n)];      maxima = [-sys.maxint -1 for i in range(n)];        for item in items:          for f in range(len(item)):              if (item[f] < minima[f]):                  minima[f] = item[f];                if (item[f] > maxima[f]):                  maxima[f] = item[f];    return minima,maxima; |

***chevron\_right***

*filter\_none*

The variables *minima, maxima* are lists containing the min and max values of the items respectively. We initialize each mean’s feature values randomly between the corresponding minimum and maximum in those above two lists:

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def InitializeMeans(items, k, cMin, cMax):        # Initialize means to random numbers between      # the min and max of each column/feature      f = len(items[0]); # number of features      means = [[0 for i in range(f)] for j in range(k)];        for mean in means:          for i in range(len(mean)):                # Set value to a random float              # (adding +-1 to avoid a wide placement of a mean)              mean[i] = uniform(cMin[i]+1, cMax[i]-1);        return means; |

***chevron\_right***

*filter\_none*

**Euclidean Distance**

We will be using the euclidean distance as a metric of similarity for our data set (note: depending on your items, you can use another similarity metric).

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def EuclideanDistance(x, y):      S = 0; #  The sum of the squared differences of the elements      for i in range(len(x)):          S += math.pow(x[i]-y[i], 2);        return math.sqrt(S); #The square root of the sum |

***chevron\_right***

*filter\_none*

**Update Means**

To update a mean, we need to find the average value for its feature, for all the items in the mean/cluster. We can do this by adding all the values and then dividing by the number of items, or we can use a more elegant solution. We will calculate the new average without having to re-add all the values, by doing the following:

m = (m\*(n-1)+x)/n

where *m* is the mean value for a feature, *n* is the number of items in the cluster and *x* is the feature value for the added item. We do the above for each feature to get the new mean.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def UpdateMean(n,mean,item):      for i in range(len(mean)):          m = mean[i];          m = (m\*(n-1)+item[i])/float(n);          mean[i] = round(m, 3);        return mean; |

***chevron\_right***

*filter\_none*

**Classify Items**

Now we need to write a function to classify an item to a group/cluster. For the given item, we will find its similarity to each mean, and we will classify the item to the closest one.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def Classify(means,item):        # Classify item to the mean with minimum distance      minimum = sys.maxint;      index = -1;        for i in range(len(means)):            # Find distance from item to mean          dis = EuclideanDistance(item, means[i]);            if (dis < minimum):              minimum = dis;              index = i;        return index; |

***chevron\_right***

*filter\_none*

**Find Means**

To actually find the means, we will loop through all the items, classify them to their nearest cluster and update the cluster’s mean. We will repeat the process for some fixed number of iterations. If between two iterations no item changes classification, we stop the process as the algorithm has found the optimal solution.

The below function takes as input *k* (the number of desired clusters), the items and the number of maximum iterations, and returns the means and the clusters. The classification of an item is stored in the array *belongsTo* and the number of items in a cluster is stored in *clusterSizes*.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

|  |
| --- |
| def CalculateMeans(k,items,maxIterations=100000):        # Find the minima and maxima for columns      cMin, cMax = FindColMinMax(items);        # Initialize means at random points      means = InitializeMeans(items,k,cMin,cMax);        # Initialize clusters, the array to hold      # the number of items in a class      clusterSizes= [0 for i in range(len(means))];        # An array to hold the cluster an item is in      belongsTo = [0 for i in range(len(items))];        # Calculate means      for e in range(maxIterations):            # If no change of cluster occurs, halt          noChange = True;          for i in range(len(items)):                item = items[i];                # Classify item into a cluster and update the              # corresponding means.              index = Classify(means,item);                clusterSizes[index] += 1;              cSize = clusterSizes[index];              means[index] = UpdateMean(cSize,means[index],item);                # Item changed cluster              if(index != belongsTo[i]):                  noChange = False;                belongsTo[i] = index;            # Nothing changed, return          if (noChange):              break;        return means; |

***chevron\_right***

*filter\_none*

**Find Clusters**

Finally we want to find the clusters, given the means. We will iterate through all the items and we will classify each item to its closest cluster.

*filter\_none*

*edit*  
*close*

*play\_arrow*

*link*  
*brightness\_4*  
*code*

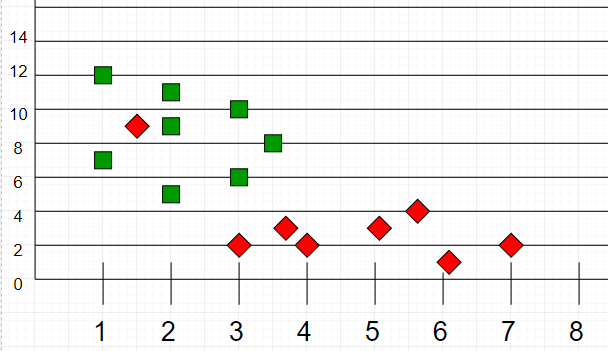
|  |
| --- |
| def FindClusters(means,items):      clusters = [[] for i in range(len(means))]; # Init clusters        for item in items:            # Classify item into a cluster          index = Classify(means,item);            # Add item to cluster          clusters[index].append(item);        return clusters; |

**K-Nearest Neighbours**

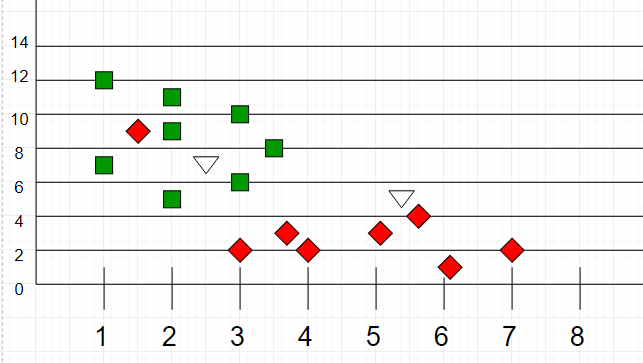
K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as [GMM](https://en.wikipedia.org/wiki/Mixture_model), which assume a Gaussian distribution of the given data).

We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:  


Now, given another set of data points (also called testing data), allocate these points a group by analyzing the training set. Note that the unclassified points are marked as ‘White’.



**Intuition**  
If we plot these points on a graph, we may be able to locate some clusters, or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbours belong to. This means, a point close to a cluster of points classified as ‘Red’ has a higher probability of getting classified as ‘Red’.

Intuitively, we can see that the first point (2.5, 7) should be classified as ‘Green’ and the second point (5.5, 4.5) should be classified as ‘Red’.

**Algorithm**  
Let m be the number of training data samples. Let p be an unknown point.

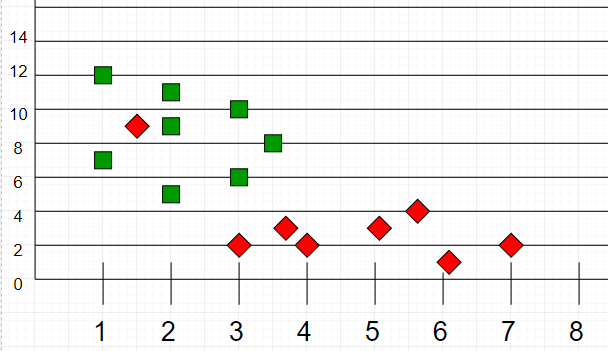
1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m:
3. Calculate Euclidean distance d(arr[i], p).
4. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
5. Return the majority label among S.

**K-Nearest Neighbours**

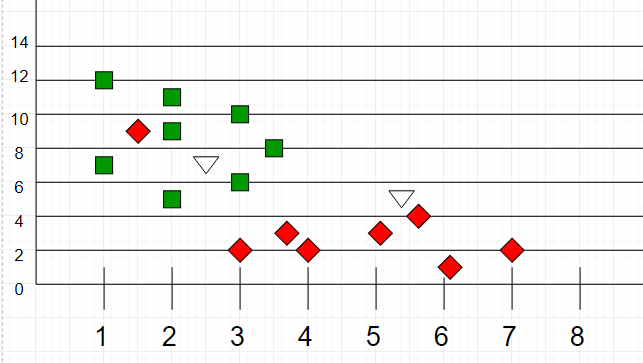
K-Nearest Neighbors is one of the most basic yet essential classification algorithms in Machine Learning. It belongs to the supervised learning domain and finds intense application in pattern recognition, data mining and intrusion detection.

It is widely disposable in real-life scenarios since it is non-parametric, meaning, it does not make any underlying assumptions about the distribution of data (as opposed to other algorithms such as [GMM](https://en.wikipedia.org/wiki/Mixture_model), which assume a Gaussian distribution of the given data).

We are given some prior data (also called training data), which classifies coordinates into groups identified by an attribute.

As an example, consider the following table of data points containing two features:  


Now, given another set of data points (also called testing data), allocate these points a group by analyzing the training set. Note that the unclassified points are marked as ‘White’.



**Intuition**  
If we plot these points on a graph, we may be able to locate some clusters, or groups. Now, given an unclassified point, we can assign it to a group by observing what group its nearest neighbours belong to. This means, a point close to a cluster of points classified as ‘Red’ has a higher probability of getting classified as ‘Red’.

Intuitively, we can see that the first point (2.5, 7) should be classified as ‘Green’ and the second point (5.5, 4.5) should be classified as ‘Red’.

**Algorithm**  
Let m be the number of training data samples. Let p be an unknown point.

1. Store the training samples in an array of data points arr[]. This means each element of this array represents a tuple (x, y).
2. for i=0 to m:
3. Calculate Euclidean distance d(arr[i], p).
4. Make set S of K smallest distances obtained. Each of these distances correspond to an already classified data point.
5. Return the majority label among S.