**Linear Regression**

Linear regression is a prediction method that is more than 200 years old.

[Simple linear regression](http://machinelearningmastery.com/simple-linear-regression-tutorial-for-machine-learning/) is a great first machine learning algorithm to implement as it requires you to estimate properties from your training dataset, but is simple enough for beginners to understand.

In this tutorial, you will discover how to implement the simple linear regression algorithm from scratch in Python.

After completing this tutorial you will know:

How to estimate statistical quantities from training data.

How to estimate linear regression coefficients from data.

How to make predictions using linear regression for new data.

Let’s get started.

===================================================================

How To Implement Simple Linear Regression From Scratch With Python  
Photo by [Kamyar Adl](https://www.flickr.com/photos/kamshots/456696484/), some rights reserved.

Description

This section is divided into two parts, a description of the simple linear regression technique and a description of the dataset to which we will later apply it.

Simple Linear Regression

Linear regression assumes a linear or straight line relationship between the input variables (X) and the single output variable (y).

More specifically, that output (y) can be calculated from a linear combination of the input variables (X). When there is a single input variable, the method is referred to as a simple linear regression.

In simple linear regression we can use statistics on the training data to estimate the coefficients required by the model to make predictions on new data.

The line for a simple linear regression model can be written as:



|  |  |
| --- | --- |
| 1 | y = b0 + b1 \* x |

where b0 and b1 are the coefficients we must estimate from the training data.

Once the coefficients are known, we can use this equation to estimate output values for y given new input examples of x.

It requires that you calculate statistical properties from the data such as mean, variance and covariance.

All the algebra has been taken care of and we are left with some arithmetic to implement to estimate the simple linear regression coefficients.

Briefly, we can estimate the coefficients as follows:



|  |  |
| --- | --- |
| 1  2 | B1 = sum((x(i) - mean(x)) \* (y(i) - mean(y))) / sum( (x(i) - mean(x))^2 )  B0 = mean(y) - B1 \* mean(x) |

where the i refers to the value of the ith value of the input x or output y.

Don’t worry if this is not clear right now, these are the functions will implement in the tutorial.

Swedish Insurance Dataset

We will use a real dataset to demonstrate simple linear regression.

The dataset is called the “Auto Insurance in Sweden” dataset and involves predicting the total payment for all the claims in thousands of Swedish Kronor (y) given the total number of claims (x).

This means that for a new number of claims (x) we will be able to predict the total payment of claims (y).

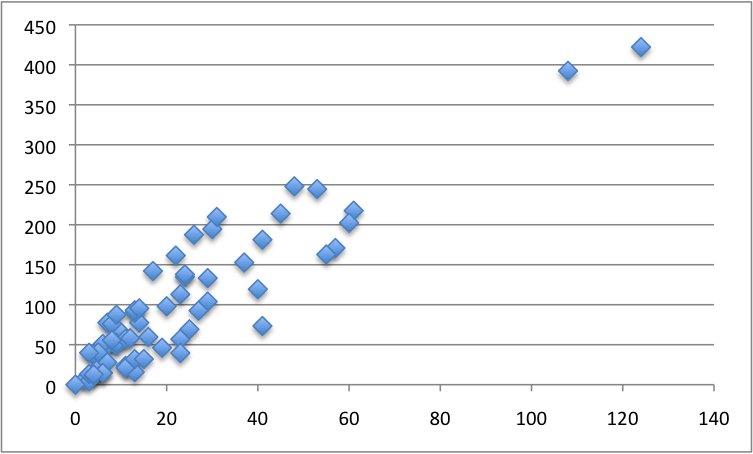
Here is a small sample of the first 5 records of the dataset.



|  |  |
| --- | --- |
| 1  2  3  4  5 | 108,392.5  19,46.2  13,15.7  124,422.2  40,119.4 |

Using the Zero Rule algorithm (that predicts the mean value) a Root Mean Squared Error or RMSE of about 72.251 (thousands of Kronor) is expected.

Below is a scatter plot of the entire dataset.



Swedish Insurance Dataset

You can download the raw dataset from [here](https://www.math.muni.cz/~kolacek/docs/frvs/M7222/data/AutoInsurSweden.txt) or [here](http://college.cengage.com/mathematics/brase/understandable_statistics/7e/students/datasets/slr/frames/slr06.html).

Save it to a CSV file in your local working directory with the name “insurance.csv“.

Note, you may need to convert the European “,” to the decimal “.”. You will also need change the file from white-space-separated variables to CSV format.

Tutorial

This tutorial is broken down into five parts:

Calculate Mean and Variance.

Calculate Covariance.

Estimate Coefficients.

Make Predictions.

Predict Insurance.

These steps will give you the foundation you need to implement and train simple linear regression models for your own prediction problems.

1. Calculate Mean and Variance

The first step is to estimate the mean and the variance of both the input and output variables from the training data.

The mean of a list of numbers can be calculated as:



|  |  |
| --- | --- |
| 1 | mean(x) = sum(x) / count(x) |

Below is a function named mean() that implements this behavior for a list of numbers.



|  |  |
| --- | --- |
| 1  2  3 | # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values)) |

The variance is the sum squared difference for each value from the mean value.

Variance for a list of numbers can be calculated as:



|  |  |
| --- | --- |
| 1 | variance = sum( (x - mean(x))^2 ) |

Below is a function named variance() that calculates the variance of a list of numbers. It requires the mean of the list to be provided as an argument, just so we don’t have to calculate it more than once.



|  |  |
| --- | --- |
| 1  2  3 | # Calculate the variance of a list of numbers  def variance(values, mean):  return sum([(x-mean)\*\*2 for x in values]) |

We can put these two functions together and test them on a small contrived dataset.

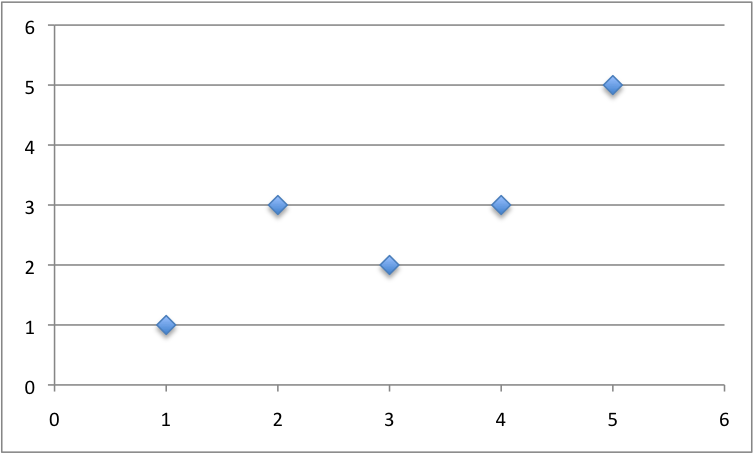
Below is a small dataset of x and y values.

NOTE: delete the column headers from this data if you save it to a .CSV file for use with the final code example.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | x, y  1, 1  2, 3  4, 3  3, 2  5, 5 |

We can plot this dataset on a scatter plot graph as follows:



Small Contrived Dataset For Simple Linear Regression

We can calculate the mean and variance for both the x and y values in the example below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | # Estimate Mean and Variance    # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values))    # Calculate the variance of a list of numbers  def variance(values, mean):  return sum([(x-mean)\*\*2 for x in values])    # calculate mean and variance  dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  mean\_x, mean\_y = mean(x), mean(y)  var\_x, var\_y = variance(x, mean\_x), variance(y, mean\_y)  print('x stats: mean=%.3f variance=%.3f' % (mean\_x, var\_x))  print('y stats: mean=%.3f variance=%.3f' % (mean\_y, var\_y)) |

Running this example prints out the mean and variance for both columns.



|  |  |
| --- | --- |
| 1  2 | x stats: mean=3.000 variance=10.000  y stats: mean=2.800 variance=8.800 |

This is our first step, next we need to put these values to use in calculating the covariance.

2. Calculate Covariance

The covariance of two groups of numbers describes how those numbers change together.

Covariance is a generalization of correlation. Correlation describes the relationship between two groups of numbers, whereas covariance can describe the relationship between two or more groups of numbers.

Additionally, covariance can be normalized to produce a correlation value.

Nevertheless, we can calculate the covariance between two variables as follows:



|  |  |
| --- | --- |
| 1 | covariance = sum((x(i) - mean(x)) \* (y(i) - mean(y))) |

Below is a function named covariance() that implements this statistic. It builds upon the previous step and takes the lists of x and y values as well as the mean of these values as arguments.



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | # Calculate covariance between x and y  def covariance(x, mean\_x, y, mean\_y):  covar = 0.0  for i in range(len(x)):  covar += (x[i] - mean\_x) \* (y[i] - mean\_y)  return covar |

We can test the calculation of the covariance on the same small contrived dataset as in the previous section.

Putting it all together we get the example below.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | # Calculate Covariance    # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values))    # Calculate covariance between x and y  def covariance(x, mean\_x, y, mean\_y):  covar = 0.0  for i in range(len(x)):  covar += (x[i] - mean\_x) \* (y[i] - mean\_y)  return covar    # calculate covariance  dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  mean\_x, mean\_y = mean(x), mean(y)  covar = covariance(x, mean\_x, y, mean\_y)  print('Covariance: %.3f' % (covar)) |

Running this example prints the covariance for the x and y variables.



|  |  |
| --- | --- |
| 1 | Covariance: 8.000 |

We now have all the pieces in place to calculate the coefficients for our model.

3. Estimate Coefficients

We must estimate the values for two coefficients in simple linear regression.

The first is B1 which can be estimated as:



|  |  |
| --- | --- |
| 1 | B1 = sum((x(i) - mean(x)) \* (y(i) - mean(y))) / sum( (x(i) - mean(x))^2 ) |

We have learned some things above and can simplify this arithmetic to:



|  |  |
| --- | --- |
| 1 | B1 = covariance(x, y) / variance(x) |

We already have functions to calculate covariance() and variance().

Next, we need to estimate a value for B0, also called the intercept as it controls the starting point of the line where it intersects the y-axis.



|  |  |
| --- | --- |
| 1 | B0 = mean(y) - B1 \* mean(x) |

Again, we know how to estimate B1 and we have a function to estimate mean().

We can put all of this together into a function named coefficients() that takes the dataset as an argument and returns the coefficients.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | # Calculate coefficients  def coefficients(dataset):  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  x\_mean, y\_mean = mean(x), mean(y)  b1 = covariance(x, x\_mean, y, y\_mean) / variance(x, x\_mean)  b0 = y\_mean - b1 \* x\_mean  return [b0, b1] |

We can put this together with all of the functions from the previous two steps and test out the calculation of coefficients.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | # Calculate Coefficients    # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values))    # Calculate covariance between x and y  def covariance(x, mean\_x, y, mean\_y):  covar = 0.0  for i in range(len(x)):  covar += (x[i] - mean\_x) \* (y[i] - mean\_y)  return covar    # Calculate the variance of a list of numbers  def variance(values, mean):  return sum([(x-mean)\*\*2 for x in values])    # Calculate coefficients  def coefficients(dataset):  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  x\_mean, y\_mean = mean(x), mean(y)  b1 = covariance(x, x\_mean, y, y\_mean) / variance(x, x\_mean)  b0 = y\_mean - b1 \* x\_mean  return [b0, b1]    # calculate coefficients  dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]  b0, b1 = coefficients(dataset)  print('Coefficients: B0=%.3f, B1=%.3f' % (b0, b1)) |

Running this example calculates and prints the coefficients.



|  |  |
| --- | --- |
| 1 | Coefficients: B0=0.400, B1=0.800 |

Now that we know how to estimate the coefficients, the next step is to use them.

4. Make Predictions

The simple linear regression model is a line defined by coefficients estimated from training data.

Once the coefficients are estimated, we can use them to make predictions.

The equation to make predictions with a simple linear regression model is as follows:



|  |  |
| --- | --- |
| 1 | y = b0 + b1 \* x |

Below is a function named simple\_linear\_regression() that implements the prediction equation to make predictions on a test dataset. It also ties together the estimation of the coefficients on training data from the steps above.

The coefficients prepared from the training data are used to make predictions on the test data, which are then returned.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def simple\_linear\_regression(train, test):  predictions = list()  b0, b1 = coefficients(train)  for row in test:  yhat = b0 + b1 \* row[0]  predictions.append(yhat)  return predictions |

Let’s pull together everything we have learned and make predictions for our simple contrived dataset.

As part of this example, we will also add in a function to manage the evaluation of the predictions called evaluate\_algorithm() and another function to estimate the Root Mean Squared Error of the predictions called rmse\_metric().

The full example is listed below.



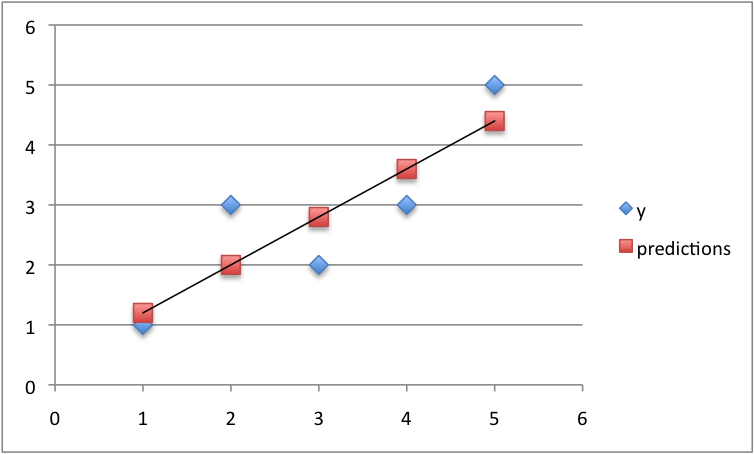
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62 | # Standalone simple linear regression example  from math import sqrt    # Calculate root mean squared error  def rmse\_metric(actual, predicted):  sum\_error = 0.0  for i in range(len(actual)):  prediction\_error = predicted[i] - actual[i]  sum\_error += (prediction\_error \*\* 2)  mean\_error = sum\_error / float(len(actual))  return sqrt(mean\_error)    # Evaluate regression algorithm on training dataset  def evaluate\_algorithm(dataset, algorithm):  test\_set = list()  for row in dataset:  row\_copy = list(row)  row\_copy[-1] = None  test\_set.append(row\_copy)  predicted = algorithm(dataset, test\_set)  print(predicted)  actual = [row[-1] for row in dataset]  rmse = rmse\_metric(actual, predicted)  return rmse    # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values))    # Calculate covariance between x and y  def covariance(x, mean\_x, y, mean\_y):  covar = 0.0  for i in range(len(x)):  covar += (x[i] - mean\_x) \* (y[i] - mean\_y)  return covar    # Calculate the variance of a list of numbers  def variance(values, mean):  return sum([(x-mean)\*\*2 for x in values])    # Calculate coefficients  def coefficients(dataset):  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  x\_mean, y\_mean = mean(x), mean(y)  b1 = covariance(x, x\_mean, y, y\_mean) / variance(x, x\_mean)  b0 = y\_mean - b1 \* x\_mean  return [b0, b1]    # Simple linear regression algorithm  def simple\_linear\_regression(train, test):  predictions = list()  b0, b1 = coefficients(train)  for row in test:  yhat = b0 + b1 \* row[0]  predictions.append(yhat)  return predictions    # Test simple linear regression  dataset = [[1, 1], [2, 3], [4, 3], [3, 2], [5, 5]]  rmse = evaluate\_algorithm(dataset, simple\_linear\_regression)  print('RMSE: %.3f' % (rmse)) |

Running this example displays the following output that first lists the predictions and the RMSE of these predictions.



|  |  |
| --- | --- |
| 1  2 | [1.1999999999999995, 1.9999999999999996, 3.5999999999999996, 2.8, 4.3999999999999995]  RMSE: 0.693 |

Finally, we can plot the predictions as a line and compare it to the original dataset.



Predictions For Small Contrived Dataset For Simple Linear Regression

5. Predict Insurance

We now know how to implement a simple linear regression model.

Let’s apply it to the Swedish insurance dataset.

This section assumes that you have downloaded the dataset to the file insurance.csv and it is available in the current working directory.

We will add some convenience functions to the simple linear regression from the previous steps.

Specifically a function to load the CSV file called load\_csv(), a function to convert a loaded dataset to numbers called str\_column\_to\_float(), a function to evaluate an algorithm using a train and test set called train\_test\_split() a function to calculate RMSE called rmse\_metric() and a function to evaluate an algorithm called evaluate\_algorithm().

The complete example is listed below.

A training dataset of 60% of the data is used to prepare the model and predictions are made on the remaining 40%.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98 | # Simple Linear Regression on the Swedish Insurance Dataset  from random import seed  from random import randrange  from csv import reader  from math import sqrt    # Load a CSV file  def load\_csv(filename):  dataset = list()  with open(filename, 'r') as file:  csv\_reader = reader(file)  for row in csv\_reader:  if not row:  continue  dataset.append(row)  return dataset    # Convert string column to float  def str\_column\_to\_float(dataset, column):  for row in dataset:  row[column] = float(row[column].strip())    # Split a dataset into a train and test set  def train\_test\_split(dataset, split):  train = list()  train\_size = split \* len(dataset)  dataset\_copy = list(dataset)  while len(train) < train\_size:  index = randrange(len(dataset\_copy))  train.append(dataset\_copy.pop(index))  return train, dataset\_copy    # Calculate root mean squared error  def rmse\_metric(actual, predicted):  sum\_error = 0.0  for i in range(len(actual)):  prediction\_error = predicted[i] - actual[i]  sum\_error += (prediction\_error \*\* 2)  mean\_error = sum\_error / float(len(actual))  return sqrt(mean\_error)    # Evaluate an algorithm using a train/test split  def evaluate\_algorithm(dataset, algorithm, split, \*args):  train, test = train\_test\_split(dataset, split)  test\_set = list()  for row in test:  row\_copy = list(row)  row\_copy[-1] = None  test\_set.append(row\_copy)  predicted = algorithm(train, test\_set, \*args)  actual = [row[-1] for row in test]  rmse = rmse\_metric(actual, predicted)  return rmse    # Calculate the mean value of a list of numbers  def mean(values):  return sum(values) / float(len(values))    # Calculate covariance between x and y  def covariance(x, mean\_x, y, mean\_y):  covar = 0.0  for i in range(len(x)):  covar += (x[i] - mean\_x) \* (y[i] - mean\_y)  return covar    # Calculate the variance of a list of numbers  def variance(values, mean):  return sum([(x-mean)\*\*2 for x in values])    # Calculate coefficients  def coefficients(dataset):  x = [row[0] for row in dataset]  y = [row[1] for row in dataset]  x\_mean, y\_mean = mean(x), mean(y)  b1 = covariance(x, x\_mean, y, y\_mean) / variance(x, x\_mean)  b0 = y\_mean - b1 \* x\_mean  return [b0, b1]    # Simple linear regression algorithm  def simple\_linear\_regression(train, test):  predictions = list()  b0, b1 = coefficients(train)  for row in test:  yhat = b0 + b1 \* row[0]  predictions.append(yhat)  return predictions    # Simple linear regression on insurance dataset  seed(1)  # load and prepare data  filename = 'insurance.csv'  dataset = load\_csv(filename)  for i in range(len(dataset[0])):  str\_column\_to\_float(dataset, i)  # evaluate algorithm  split = 0.6  rmse = evaluate\_algorithm(dataset, simple\_linear\_regression, split)  print('RMSE: %.3f' % (rmse)) |

Running the algorithm prints the RMSE for the trained model on the training dataset.

A score of about 38 (thousands of Kronor) was achieved, which is much better than the Zero Rule algorithm that achieves approximately 72 (thousands of Kronor) on the same problem.



|  |  |
| --- | --- |
| 1 | RMSE: 38.339 |

Extensions

The best extension to this tutorial is to try out the algorithm on more problems.

Small datasets with just an input (x) and output (y) columns are popular for demonstration in statistical books and courses. Many of these datasets are available online.

Seek out some more small datasets and make predictions using simple linear regression.

Did you apply simple linear regression to another dataset?  
Share your experiences in the comments below.

Review

In this tutorial, you discovered how to implement the simple linear regression algorithm from scratch in Python.

Specifically, you learned:

How to estimate statistics from a training dataset like mean, variance and covariance.

How to estimate model coefficients and use them to make predictions.

How to use simple linear regression to make predictions on a real dataset.

Do you have any questions?  
Ask your question in the comments below and I will do my best to answer.

================================================================================================

We are probably living in the most defining period of human history. The period when computing moved from large mainframes to PCs to cloud. But what makes it defining is not what has happened, but what is coming our way in years to come.

What makes this period exciting for some one like me is the democratization of the tools and techniques, which followed the boost in computing. Today, as a data scientist, I can build data crunching machines with complex algorithms for a few dollors per hour. But, reaching here wasn’t easy! I had my dark days and nights.

I have deliberately skipped the statistics behind these techniques, as you don’t need to understand them at the start. So, if you are looking for statistical understanding of these algorithms, you should look elsewhere. But, if you are looking to equip yourself to start building machine learning project, you are in for a treat.

Broadly, there are 3 types of Machine Learning Algorithms..

1. Supervised Learning

How it works: This algorithm consist of a target / outcome variable (or dependent variable) which is to be predicted from a given set of predictors (independent variables). Using these set of variables, we generate a function that map inputs to desired outputs. The training process continues until the model achieves a desired level of accuracy on the training data. Examples of Supervised Learning: Regression, [Decision Tree](https://www.analyticsvidhya.com/blog/2015/01/decision-tree-simplified/), [Random Forest](https://www.analyticsvidhya.com/blog/2014/06/introduction-random-forest-simplified/), KNN, Logistic Regression etc.

2. Unsupervised Learning

How it works: In this algorithm, we do not have any target or outcome variable to predict / estimate.  It is used for clustering population in different groups, which is widely used for segmenting customers in different groups for specific intervention. Examples of Unsupervised Learning: Apriori algorithm, K-means.

3. Reinforcement Learning:

How it works:  Using this algorithm, the machine is trained to make specific decisions. It works this way: the machine is exposed to an environment where it trains itself continually using trial and error. This machine learns from past experience and tries to capture the best possible knowledge to make accurate business decisions. Example of Reinforcement Learning: Markov Decision Process

List of Common Machine Learning Algorithms

Here is the list of commonly used machine learning algorithms. These algorithms can be applied to almost any data problem:

Linear Regression

Logistic Regression

Decision Tree

SVM

Naive Bayes

KNN

K-Means

Random Forest

Dimensionality Reduction Algorithms

Gradient Boosting algorithms

GBM

XGBoost

LightGBM

CatBoost

1. Linear Regression

It is used to estimate real values (cost of houses, number of calls, total sales etc.) based on continuous variable(s). Here, we establish relationship between independent and dependent variables by fitting a best line. This best fit line is known as regression line and represented by a linear equation Y= a \*X + b.

The best way to understand linear regression is to relive this experience of childhood. Let us say, you ask a child in fifth grade to arrange people in his class by increasing order of weight, without asking them their weights! What do you think the child will do? He / she would likely look (visually analyze) at the height and build of people and arrange them using a combination of these visible parameters. This is linear regression in real life! The child has actually figured out that height and build would be correlated to the weight by a relationship, which looks like the equation above.

In this equation:

Y – Dependent Variable

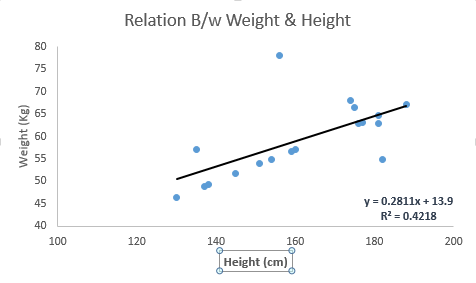
a – Slope

X – Independent variable

b – Intercept

These coefficients a and b are derived based on minimizing the sum of squared difference of distance between data points and regression line.

Look at the below example. Here we have identified the best fit line having linear equation y=0.2811x+13.9. Now using this equation, we can find the weight, knowing the height of a person.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Linear_Regression.png)

Linear Regression is of mainly two types: Simple Linear Regression and Multiple Linear Regression. Simple Linear Regression is characterized by one independent variable. And, Multiple Linear Regression(as the name suggests) is characterized by multiple (more than 1) independent variables. While finding best fit line, you can fit a polynomial or curvilinear regression. And these are known as polynomial or curvilinear regression.

Python Code

#Import Library

#Import other necessary libraries like pandas, numpy...

from sklearn import linear\_model

#Load Train and Test datasets

#Identify feature and response variable(s) and values must be numeric and numpy arrays

x\_train=input\_variables\_values\_training\_datasets

y\_train=target\_variables\_values\_training\_datasets

x\_test=input\_variables\_values\_test\_datasets

# Create linear regression object

linear = linear\_model.LinearRegression()

# Train the model using the training sets and check score

linear.fit(x\_train, y\_train)

linear.score(x\_train, y\_train)

#Equation coefficient and Intercept

print('Coefficient: \n', linear.coef\_)

print('Intercept: \n', linear.intercept\_)

#Predict Output

predicted= linear.predict(x\_test)

R Code

#Load Train and Test datasets

#Identify feature and response variable(s) and values must be numeric and numpy arrays

x\_train <- input\_variables\_values\_training\_datasets

y\_train <- target\_variables\_values\_training\_datasets

x\_test <- input\_variables\_values\_test\_datasets

x <- cbind(x\_train,y\_train)

# Train the model using the training sets and check score

linear <- lm(y\_train ~ ., data = x)

summary(linear)

#Predict Output

predicted= predict(linear,x\_test)

2. Logistic Regression

Don’t get confused by its name! It is a classification not a regression algorithm. It is used to estimate discrete values ( Binary values like 0/1, yes/no, true/false ) based on given set of independent variable(s). In simple words, it predicts the probability of occurrence of an event by fitting data to a [logit function](https://en.wikipedia.org/wiki/Logistic_function). Hence, it is also known as logit regression. Since, it predicts the probability, its output values lies between 0 and 1 (as expected).

Again, let us try and understand this through a simple example.

Let’s say your friend gives you a puzzle to solve. There are only 2 outcome scenarios – either you solve it or you don’t. Now imagine, that you are being given wide range of puzzles / quizzes in an attempt to understand which subjects you are good at. The outcome to this study would be something like this – if you are given a trignometry based tenth grade problem, you are 70% likely to solve it. On the other hand, if it is grade fifth history question, the probability of getting an answer is only 30%. This is what Logistic Regression provides you.

Coming to the math, the log odds of the outcome is modeled as a linear combination of the predictor variables.

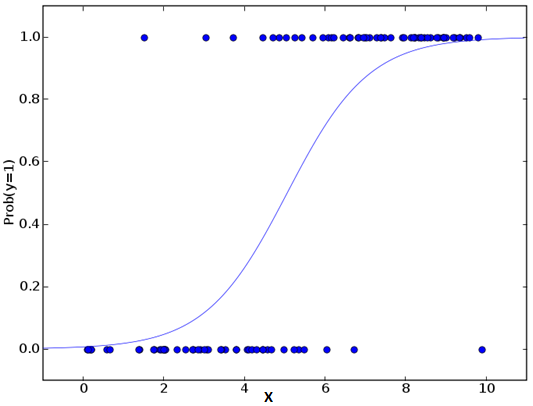
odds= p/ (1-p) = probability of event occurrence / probability of not event occurrence

ln(odds) = ln(p/(1-p))

logit(p) = ln(p/(1-p)) = b0+b1X1+b2X2+b3X3....+bkXk

Above, p is the probability of presence of the characteristic of interest. It chooses parameters that maximize the likelihood of observing the sample values rather than that minimize the sum of squared errors (like in ordinary regression).

Now, you may ask, why take a log? For the sake of simplicity, let’s just say that this is one of the best mathematical way to replicate a step function. I can go in more details, but that will beat the purpose of this article.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Logistic_Regression.png)Python Code

#Import Library

from sklearn.linear\_model import LogisticRegression

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create logistic regression object

model = LogisticRegression()

# Train the model using the training sets and check score

model.fit(X, y)

model.score(X, y)

#Equation coefficient and Intercept

print('Coefficient: \n', model.coef\_)

print('Intercept: \n', model.intercept\_)

#Predict Output

predicted= model.predict(x\_test)

R Code

x <- cbind(x\_train,y\_train)

# Train the model using the training sets and check score

logistic <- glm(y\_train ~ ., data = x,family='binomial')

summary(logistic)

#Predict Output

predicted= predict(logistic,x\_test)

Furthermore..

There are many different steps that could be tried in order to improve the model:

including interaction terms

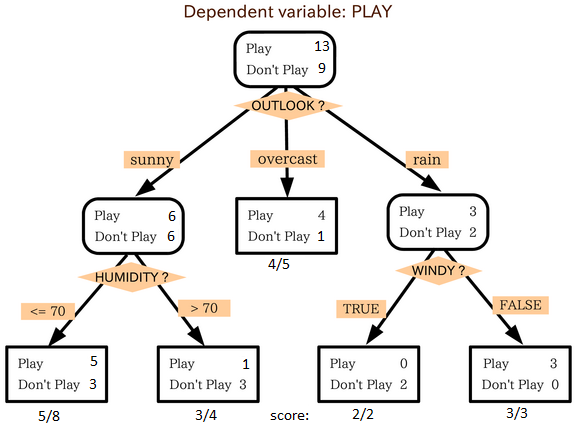
removing features

[regularization techniques](https://www.analyticsvidhya.com/blog/2015/02/avoid-over-fitting-regularization/)

using a non-linear model

3. Decision Tree

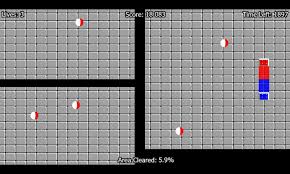
This is one of my favorite algorithm and I use it quite frequently. It is a type of supervised learning algorithm that is mostly used for classification problems. Surprisingly, it works for both categorical and continuous dependent variables. In this algorithm, we split the population into two or more homogeneous sets. This is done based on most significant attributes/ independent variables to make as distinct groups as possible. For more details, you can read: [Decision Tree Simplified](https://www.analyticsvidhya.com/blog/2015/01/decision-tree-simplified/).

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/IkBzK.png)

source: [statsexchange](http://stats.stackexchange.com/)

In the image above, you can see that population is classified into four different groups based on multiple attributes to identify ‘if they will play or not’. To split the population into different heterogeneous groups, it uses various techniques like Gini, Information Gain, Chi-square, entropy.

The best way to understand how decision tree works, is to play Jezzball – a classic game from Microsoft (image below). Essentially, you have a room with moving walls and you need to create walls such that maximum area gets cleared off with out the balls.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/download.jpg)

So, every time you split the room with a wall, you are trying to create 2 different populations with in the same room. Decision trees work in very similar fashion by dividing a population in as different groups as possible.

More: [Simplified Version of Decision Tree Algorithms](https://www.analyticsvidhya.com/blog/2015/01/decision-tree-simplified/)

Python Code

#Import Library

#Import other necessary libraries like pandas, numpy...

from sklearn import tree

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create tree object

model = tree.DecisionTreeClassifier(criterion='gini') # for classification, here you can change the algorithm as gini or entropy (information gain) by default it is gini

# model = tree.DecisionTreeRegressor() for regression

# Train the model using the training sets and check score

model.fit(X, y)

model.score(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(rpart)

x <- cbind(x\_train,y\_train)

# grow tree

fit <- rpart(y\_train ~ ., data = x,method="class")

summary(fit)

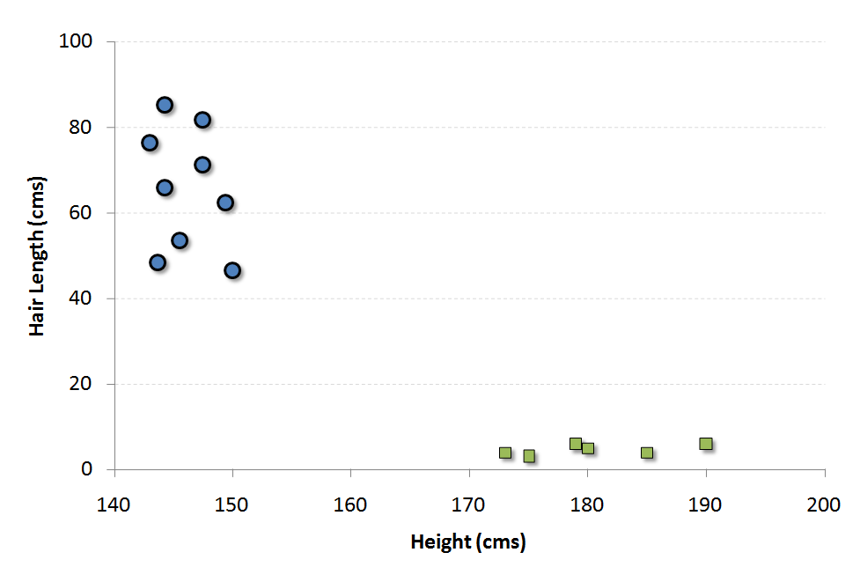
#Predict Output

predicted= predict(fit,x\_test)

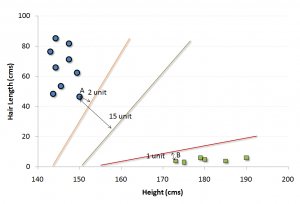
4. SVM (Support Vector Machine)

It is a classification method. In this algorithm, we plot each data item as a point in n-dimensional space (where n is number of features you have) with the value of each feature being the value of a particular coordinate.

For example, if we only had two features like Height and Hair length of an individual, we’d first plot these two variables in two dimensional space where each point has two co-ordinates (these co-ordinates are known as Support Vectors)

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/SVM1.png)

Now, we will find some line that splits the data between the two differently classified groups of data. This will be the line such that the distances from the closest point in each of the two groups will be farthest away.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/SVM2.png)

In the example shown above, the line which splits the data into two differently classified groups is the black line, since the two closest points are the farthest apart from the line. This line is our classifier. Then, depending on where the testing data lands on either side of the line, that’s what class we can classify the new data as.

More: [Simplified Version of Support Vector Machine](https://www.analyticsvidhya.com/blog/2014/10/support-vector-machine-simplified/)

Think of this algorithm as playing JezzBall in n-dimensional space. The tweaks in the game are:

You can draw lines / planes at any angles (rather than just horizontal or vertical as in classic game)

The objective of the game is to segregate balls of different colors in different rooms.

And the balls are not moving.

Python Code

#Import Library

from sklearn import svm

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create SVM classification object

model = svm.svc() # there is various option associated with it, this is simple for classification. You can refer [link](http://scikit-learn.org/stable/modules/svm.html), for mo# re detail.

# Train the model using the training sets and check score

model.fit(X, y)

model.score(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(e1071)

x <- cbind(x\_train,y\_train)

# Fitting model

fit <-svm(y\_train ~ ., data = x)

summary(fit)

#Predict Output

predicted= predict(fit,x\_test)

5. Naive Bayes

It is a classification technique based on [Bayes’ theorem](https://en.wikipedia.org/wiki/Bayes%27_theorem) with an assumption of independence between predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. For example, a fruit may be considered to be an apple if it is red, round, and about 3 inches in diameter. Even if these features depend on each other or upon the existence of the other features, a naive Bayes classifier would consider all of these properties to independently contribute to the probability that this fruit is an apple.

Naive Bayesian model is easy to build and particularly useful for very large data sets. Along with simplicity, Naive Bayes is known to outperform even highly sophisticated classification methods.

Bayes theorem provides a way of calculating posterior probability P(c|x) from P(c), P(x) and P(x|c). Look at the equation below:  
[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Bayes_rule.png)

Here,

P(c|x) is the posterior probability of class (target) given predictor (attribute).

P(c) is the prior probability of class.

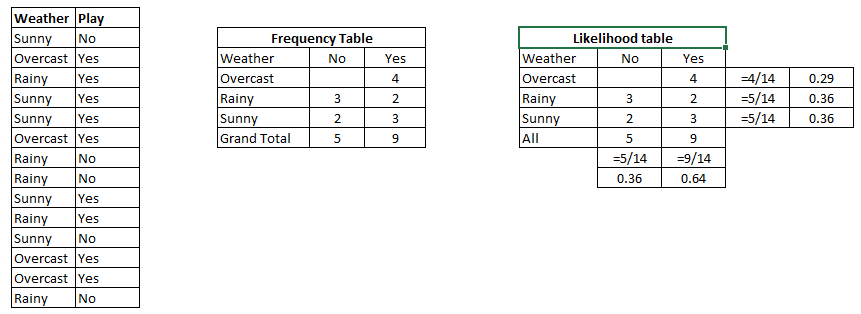
P(x|c) is the likelihood which is the probability of predictor given class.

P(x) is the prior probability of predictor.

Example: Let’s understand it using an example. Below I have a training data set of weather and corresponding target variable ‘Play’. Now, we need to classify whether players will play or not based on weather condition. Let’s follow the below steps to perform it.

Step 1: Convert the data set to frequency table

Step 2: Create Likelihood table by finding the probabilities like Overcast probability = 0.29 and probability of playing is 0.64.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Bayes_41.png)

Step 3: Now, use Naive Bayesian equation to calculate the posterior probability for each class. The class with the highest posterior probability is the outcome of prediction.

Problem: Players will pay if weather is sunny, is this statement is correct?

We can solve it using above discussed method, so P(Yes | Sunny) = P( Sunny | Yes) \* P(Yes) / P (Sunny)

Here we have P (Sunny |Yes) = 3/9 = 0.33, P(Sunny) = 5/14 = 0.36, P( Yes)= 9/14 = 0.64

Now, P (Yes | Sunny) = 0.33 \* 0.64 / 0.36 = 0.60, which has higher probability.

Naive Bayes uses a similar method to predict the probability of different class based on various attributes. This algorithm is mostly used in text classification and with problems having multiple classes.

Python Code

#Import Library

from sklearn.naive\_bayes import GaussianNB

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create SVM classification object model = GaussianNB() # there is other distribution for multinomial classes like Bernoulli Naive Bayes, [Refer link](http://scikit-learn.org/stable/modules/naive_bayes.html" \t "_blank)

# Train the model using the training sets and check score

model.fit(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(e1071)

x <- cbind(x\_train,y\_train)

# Fitting model

fit <-naiveBayes(y\_train ~ ., data = x)

summary(fit)

#Predict Output

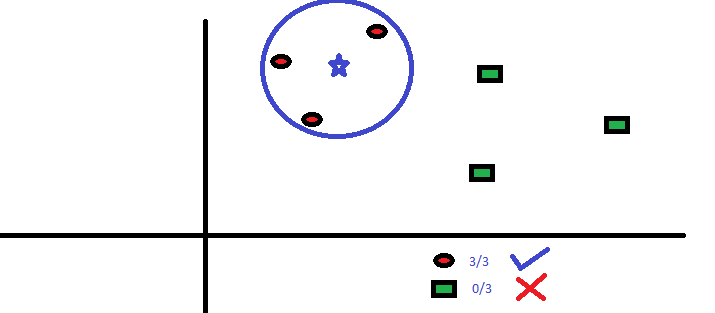
predicted= predict(fit,x\_test)

6. KNN (K- Nearest Neighbors)

It can be used for both classification and regression problems. However, it is more widely used in classification problems in the industry. K nearest neighbors is a simple algorithm that stores all available cases and classifies new cases by a majority vote of its k neighbors. The case being assigned to the class is most common amongst its K nearest neighbors measured by a distance function.

These distance functions can be Euclidean, Manhattan, Minkowski and Hamming distance. First three functions are used for continuous function and fourth one (Hamming) for categorical variables. If K = 1, then the case is simply assigned to the class of its nearest neighbor. At times, choosing K turns out to be a challenge while performing KNN modeling.

More: Introduction to k-nearest neighbors : Simplified.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/KNN.png)

KNN can easily be mapped to our real lives. If you want to learn about a person, of whom you have no information, you might like to find out about his close friends and the circles he moves in and gain access to his/her information!

Things to consider before selecting KNN:

KNN is computationally expensive

Variables should be normalized else higher range variables can bias it

Works on pre-processing stage more before going for KNN like outlier, noise removal

Python Code

#Import Library

from sklearn.neighbors import KNeighborsClassifier

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create KNeighbors classifier object model

KNeighborsClassifier(n\_neighbors=6) # default value for n\_neighbors is 5

# Train the model using the training sets and check score

model.fit(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(knn)

x <- cbind(x\_train,y\_train)

# Fitting model

fit <-knn(y\_train ~ ., data = x,k=5)

summary(fit)

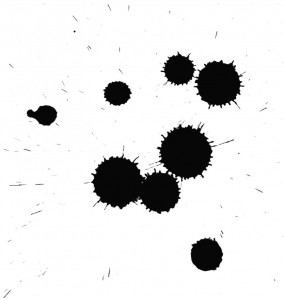
#Predict Output

predicted= predict(fit,x\_test)

7. K-Means

It is a type of unsupervised algorithm which  solves the clustering problem. Its procedure follows a simple and easy  way to classify a given data set through a certain number of  clusters (assume k clusters). Data points inside a cluster are homogeneous and heterogeneous to peer groups.

Remember figuring out shapes from ink blots? k means is somewhat similar this activity. You look at the shape and spread to decipher how many different clusters / population are present!

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/splatter_ink_blot_texture_by_maki_tak-d5p6zph.jpg)

How K-means forms cluster:

K-means picks k number of points for each cluster known as centroids.

Each data point forms a cluster with the closest centroids i.e. k clusters.

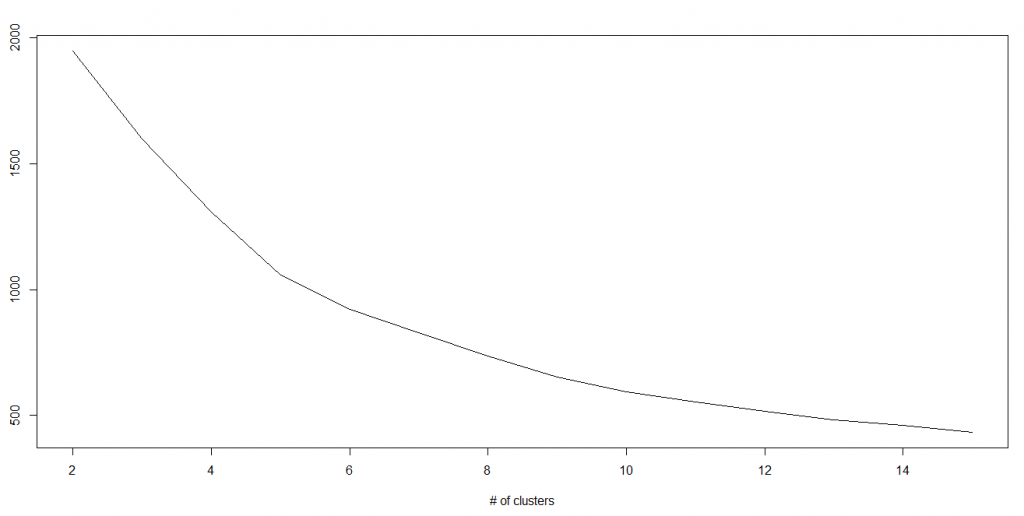
Finds the centroid of each cluster based on existing cluster members. Here we have new centroids.

As we have new centroids, repeat step 2 and 3. Find the closest distance for each data point from new centroids and get associated with new k-clusters. Repeat this process until convergence occurs i.e. centroids does not change.

How to determine value of K:

In K-means, we have clusters and each cluster has its own centroid. Sum of square of difference between centroid and the data points within a cluster constitutes within sum of square value for that cluster. Also, when the sum of square values for all the clusters are added, it becomes total within sum of square value for the cluster solution.

We know that as the number of cluster increases, this value keeps on decreasing but if you plot the result you may see that the sum of squared distance decreases sharply up to some value of k, and then much more slowly after that. Here, we can find the optimum number of cluster.

[](https://www.analyticsvidhya.com/wp-content/uploads/2015/08/Kmenas.png)

Python Code

#Import Library

from sklearn.cluster import KMeans

#Assumed you have, X (attributes) for training data set and x\_test(attributes) of test\_dataset

# Create KNeighbors classifier object model

k\_means = KMeans(n\_clusters=3, random\_state=0)

# Train the model using the training sets and check score

model.fit(X)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(cluster)

fit <- kmeans(X, 3) # 5 cluster solution

8. Random Forest

Random Forest is a trademark term for an ensemble of decision trees. In Random Forest, we’ve collection of decision trees (so known as “Forest”). To classify a new object based on attributes, each tree gives a classification and we say the tree “votes” for that class. The forest chooses the classification having the most votes (over all the trees in the forest).

Each tree is planted & grown as follows:

If the number of cases in the training set is N, then sample of N cases is taken at random but with replacement. This sample will be the training set for growing the tree.

If there are M input variables, a number m<<M is specified such that at each node, m variables are selected at random out of the M and the best split on these m is used to split the node. The value of m is held constant during the forest growing.

Each tree is grown to the largest extent possible. There is no pruning.

For more details on this algorithm, comparing with decision tree and tuning model parameters, I would suggest you to read these articles:

[Introduction to Random forest – Simplified](https://www.analyticsvidhya.com/blog/2014/06/introduction-random-forest-simplified/)

[Comparing a CART model to Random Forest (Part 1)](https://www.analyticsvidhya.com/blog/2014/06/comparing-cart-random-forest-1/)

[Comparing a Random Forest to a CART model (Part 2)](https://www.analyticsvidhya.com/blog/2014/06/comparing-random-forest-simple-cart-model/)

[Tuning the parameters of your Random Forest model](https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/)

Python

#Import Library

from sklearn.ensemble import RandomForestClassifier

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create Random Forest object

model= RandomForestClassifier()

# Train the model using the training sets and check score

model.fit(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(randomForest)

x <- cbind(x\_train,y\_train)

# Fitting model

fit <- randomForest(Species ~ ., x,ntree=500)

summary(fit)

#Predict Output

predicted= predict(fit,x\_test)

9. Dimensionality Reduction Algorithms

In the last 4-5 years, there has been an exponential increase in data capturing at every possible stages. Corporates/ Government Agencies/ Research organisations are not only coming with new sources but also they are capturing data in great detail.

For example: E-commerce companies are capturing more details about customer like their demographics, web crawling history, what they like or dislike, purchase history, feedback and many others to give them personalized attention more than your nearest grocery shopkeeper.

As a data scientist, the data we are offered also consist of many features, this sounds good for building good robust model but there is a challenge. How’d you identify highly significant variable(s) out 1000 or 2000? In such cases, dimensionality reduction algorithm helps us along with various other algorithms like Decision Tree, Random Forest, PCA, Factor Analysis, Identify based on correlation matrix, missing value ratio and others.

To know more about this algorithms, you can read “[Beginners Guide To Learn Dimension Reduction Techniques](https://www.analyticsvidhya.com/blog/2015/07/dimension-reduction-methods/)“.

Python  Code

#Import Library

from sklearn import decomposition

#Assumed you have training and test data set as train and test

# Create PCA obeject pca= decomposition.PCA(n\_components=k) #default value of k =min(n\_sample, n\_features)

# For Factor analysis

#fa= decomposition.FactorAnalysis()

# Reduced the dimension of training dataset using PCA

train\_reduced = pca.fit\_transform(train)

#Reduced the dimension of test dataset

test\_reduced = pca.transform(test)

#For more detail on this, please refer  [this link](http://scikit-learn.org/stable/modules/decomposition.html" \l "decompositions" \t "_blank).

R Code

library(stats)

pca <- princomp(train, cor = TRUE)

train\_reduced <- predict(pca,train)

test\_reduced <- predict(pca,test)

10. Gradient Boosting Algorithms

10.1. GBM

GBM is a boosting algorithm used when we deal with plenty of data to make a prediction with high prediction power. Boosting is actually an ensemble of learning algorithms which combines the prediction of several base estimators in order to improve robustness over a single estimator. It combines multiple weak or average predictors to a build strong predictor. These boosting algorithms always work well in data science competitions like Kaggle, AV Hackathon, CrowdAnalytix.

More: [Know about Boosting algorithms in detail](https://www.analyticsvidhya.com/blog/2015/05/boosting-algorithms-simplified/)

Python Code

#Import Library

from sklearn.ensemble import GradientBoostingClassifier

#Assumed you have, X (predictor) and Y (target) for training data set and x\_test(predictor) of test\_dataset

# Create Gradient Boosting Classifier object

model= GradientBoostingClassifier(n\_estimators=100, learning\_rate=1.0, max\_depth=1, random\_state=0)

# Train the model using the training sets and check score

model.fit(X, y)

#Predict Output

predicted= model.predict(x\_test)

R Code

library(caret)

x <- cbind(x\_train,y\_train)

# Fitting model

fitControl <- trainControl( method = "repeatedcv", number = 4, repeats = 4)

fit <- train(y ~ ., data = x, method = "gbm", trControl = fitControl,verbose = FALSE)

predicted= predict(fit,x\_test,type= "prob")[,2]

GradientBoostingClassifier and Random Forest are two different boosting tree classifier and often people ask about the [difference between these two algorithms](http://discuss.analyticsvidhya.com/t/what-is-the-fundamental-difference-between-randomforest-and-gradient-boosting-algorithms/2341).

10.2. XGBoost

Another classic gradient boosting algorithm that’s known to be the decisive choice between winning and losing in some Kaggle competitions.

The XGBoost has an immensely high predictive power which makes it the best choice for accuracy in events as it possesses both linear model and the tree learning algorithm, making the algorithm almost 10x faster than existing gradient booster techniques.

The support includes various objective functions, including regression, classification and ranking.

One of the most interesting things about the XGBoost is that it is also called a regularized boosting technique. This helps to reduce overfit modelling and has a massive support for a range of languages such as Scala, Java, R, Python, Julia and C++.

Supports distributed and widespread training on many machines that encompass GCE, AWS, Azure and Yarn clusters. XGBoost can also be integrated with Spark, Flink and other cloud dataflow systems with a built in cross validation at each iteration of the boosting process.

To learn more about XGBoost and parameter tuning, visit <https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/>.

Python Code:

from xgboost import XGBClassifier

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

from sklearn.metrics import accuracy\_score

X = dataset[:,0:10]

Y = dataset[:,10:]

seed = 1

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, Y, test\_size=0.33, random\_state=seed)

model = XGBClassifier()

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

#Make predictions for test data

y\_pred = model.predict(X\_test)

R Code:

require(caret)

x <- cbind(x\_train,y\_train)

# Fitting model

TrainControl <- trainControl( method = "repeatedcv", number = 10, repeats = 4)

model<- train(y ~ ., data = x, method = "xgbLinear", trControl = TrainControl,verbose = FALSE)

OR

model<- train(y ~ ., data = x, method = "xgbTree", trControl = TrainControl,verbose = FALSE)

predicted <- predict(model, x\_test)

10.3. LightGBM

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

Faster training speed and higher efficiency

Lower memory usage

Better accuracy

Parallel and GPU learning supported

Capable of handling large-scale data

The framework is a fast and high-performance gradient boosting one based on decision tree algorithms, used for ranking, classification and many other machine learning tasks. It was developed under the Distributed Machine Learning Toolkit Project of Microsoft.

Since the LightGBM is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms.

Also, it is surprisingly very fast, hence the word ‘Light’.

Refer to the article to know more about LightGBM: <https://www.analyticsvidhya.com/blog/2017/06/which-algorithm-takes-the-crown-light-gbm-vs-xgboost/>

Python Code:

data = np.random.rand(500, 10) # 500 entities, each contains 10 features

label = np.random.randint(2, size=500) # binary target

train\_data = lgb.Dataset(data, label=label)

test\_data = train\_data.create\_valid('test.svm')

param = {'num\_leaves':31, 'num\_trees':100, 'objective':'binary'}

param['metric'] = 'auc'

num\_round = 10

bst = lgb.train(param, train\_data, num\_round, valid\_sets=[test\_data])

bst.save\_model('model.txt')

# 7 entities, each contains 10 features

data = np.random.rand(7, 10)

ypred = bst.predict(data)

R Code:

library(RLightGBM)

data(example.binary)

#Parameters

num\_iterations <- 100

config <- list(objective = "binary",  metric="binary\_logloss,auc", learning\_rate = 0.1, num\_leaves = 63, tree\_learner = "serial", feature\_fraction = 0.8, bagging\_freq = 5, bagging\_fraction = 0.8, min\_data\_in\_leaf = 50, min\_sum\_hessian\_in\_leaf = 5.0)

#Create data handle and booster

handle.data <- lgbm.data.create(x)

lgbm.data.setField(handle.data, "label", y)

handle.booster <- lgbm.booster.create(handle.data, lapply(config, as.character))

#Train for num\_iterations iterations and eval every 5 steps

lgbm.booster.train(handle.booster, num\_iterations, 5)

#Predict

pred <- lgbm.booster.predict(handle.booster, x.test)

#Test accuracy

sum(y.test == (y.pred > 0.5)) / length(y.test)

#Save model (can be loaded again via lgbm.booster.load(filename))

lgbm.booster.save(handle.booster, filename = "/tmp/model.txt")

If you’re familiar with the Caret package in R, this is another way of implementing the LightGBM.

require(caret)

require(RLightGBM)

data(iris)

model <-caretModel.LGBM()

fit <- train(Species ~ ., data = iris, method=model, verbosity = 0)

print(fit)

y.pred <- predict(fit, iris[,1:4])

library(Matrix)

model.sparse <- caretModel.LGBM.sparse()

#Generate a sparse matrix

mat <- Matrix(as.matrix(iris[,1:4]), sparse = T)

fit <- train(data.frame(idx = 1:nrow(iris)), iris$Species, method = model.sparse, matrix = mat, verbosity = 0)

print(fit)

10.4. Catboost

CatBoost is a recently open-sourced machine learning algorithm from Yandex. It can easily integrate with deep learning frameworks like Google’s TensorFlow and Apple’s Core ML.

The best part about CatBoost is that it does not require extensive data training like other ML models, and can work on a variety of data formats; not undermining how robust it can be.

Make sure you handle missing data well before you proceed with the implementation.

Catboost can automatically deal with categorical variables without showing the type conversion error, which helps you to focus on tuning your model better rather than sorting out trivial errors.

Learn more about Catboost from this article: <https://www.analyticsvidhya.com/blog/2017/08/catboost-automated-categorical-data/>

Python Code:

import pandas as pd

import numpy as np

from catboost import CatBoostRegressor

#Read training and testing files

train = pd.read\_csv("train.csv")

test = pd.read\_csv("test.csv")

#Imputing missing values for both train and test

train.fillna(-999, inplace=True)

test.fillna(-999,inplace=True)

#Creating a training set for modeling and validation set to check model performance

X = train.drop(['Item\_Outlet\_Sales'], axis=1)

y = train.Item\_Outlet\_Sales

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

X\_train, X\_validation, y\_train, y\_validation = train\_test\_split(X, y, train\_size=0.7, random\_state=1234)

categorical\_features\_indices = np.where(X.dtypes != np.float)[0]

#importing library and building model

from catboost import CatBoostRegressormodel=CatBoostRegressor(iterations=50, depth=3, learning\_rate=0.1, loss\_function='RMSE')

model.fit(X\_train, y\_train,cat\_features=categorical\_features\_indices,eval\_set=(X\_validation, y\_validation),plot=True)

submission = pd.DataFrame()

submission['Item\_Identifier'] = test['Item\_Identifier']

submission['Outlet\_Identifier'] = test['Outlet\_Identifier']

submission['Item\_Outlet\_Sales'] = model.predict(test)

R Code:

set.seed(1)

require(titanic)

require(caret)

require(catboost)

tt <- titanic::titanic\_train[complete.cases(titanic::titanic\_train),]

data <- as.data.frame(as.matrix(tt), stringsAsFactors = TRUE)

drop\_columns = c("PassengerId", "Survived", "Name", "Ticket", "Cabin")

x <- data[,!(names(data) %in% drop\_columns)]y <- data[,c("Survived")]

fit\_control <- trainControl(method = "cv", number = 4,classProbs = TRUE)

grid <- expand.grid(depth = c(4, 6, 8),learning\_rate = 0.1,iterations = 100, l2\_leaf\_reg = 1e-3,            rsm = 0.95, border\_count = 64)

report <- train(x, as.factor(make.names(y)),method = catboost.caret,verbose = TRUE, preProc = NULL,tuneGrid = grid, trControl = fit\_control)

print(report)

importance <- varImp(report, scale = FALSE)

print(importance)

End Notes

By now, I am sure, you would have an idea of commonly used machine learning algorithms. My sole intention behind writing this article and providing the codes in R and Python is to get you started right away. If you are keen to master machine learning, start right away. Take up problems, develop a physical understanding of the process, apply these codes and see the fun!