Documentation for Algorithms in Supervised Machine Learning Problems

---

Table of content

What is Classification & Regression, When & where to use what ?

Classification & Regression algorithms are both Supervised Learning algorithms (means they both require a target variable to train the model). Both the algorithms are used for prediction in Machine learning and work with the labeled datasets(datasets having labels or target variables). But the difference between both is how they are used for different kinds of machine learning problems.

The main difference between Regression and Classification algorithms that Regression algorithms are used to predict the continuous values such as price, salary, age, number of calories, etc. and Classification algorithms are used to predict/Classify the discrete values such as Male or Female, True or False, Spam or Not Spam, 1 or 0, etc.

Classification

Classification predictive modeling is the task of approximating a mapping function (f) from input variables (X) to **discrete** output variables (y). The output variables are often called labels or categories. The mapping function predicts the class or category for a given observation.

For example, an email of text can be classified as belonging to one of two classes: “spam“ and “not spam“.

- ➢ A classification problem requires that examples be classified into one of two or more classes.
- ➢ A classification can have real-valued or discrete input variables.
- ➢ A problem with two classes is often called a two-class or binary classification problem.
- ➢ A problem with more than two classes is often called a multi-class classification problem.
- ➢ A problem where an example is assigned multiple classes is called a multi-label classification problem.

It is common for classification models to predict a continuous value as the probability of a given example belonging to each output class. The probabilities can be interpreted as the likelihood or confidence of a given example belonging to each class.
A predicted probability can be converted into a class value by selecting the class label that has the highest probability.
For example, a specific email or text may be assigned the probabilities of 0.1 as being “spam” and 0.9 as being “not spam”. We can convert these probabilities to a class label by selecting the “not spam” label as it has the highest predicted likelihood.
There are many ways to estimate the skill of a classification predictive model, but perhaps the most common is to calculate the classification accuracy. The classification accuracy is the percentage of correctly classified examples out of all predictions made. An algorithm that is capable of learning a classification predictive model is called a classification algorithm.

Regression

Regression predictive modeling is the task of approximating a mapping function (f) from input variables (X) to a **continuous** output variable (y).
A continuous output variable is a real-value, such as an integer or floating point value. These are often quantities, such as amounts and sizes.

For example, a house may be predicted to sell for a specific dollar value, perhaps in the range of $100,000 to $200,000.

- ➢ A regression problem requires the prediction of a quantity.
- ➢ A regression can have real valued or discrete input variables.
- ➢ A problem with multiple input variables is often called a multivariate regression problem.
- ➢ A regression problem where input variables are ordered by time is called a time series forecasting problem.

Because a regression predictive model predicts a quantity, the skill of the model must be reported as an error in those predictions.
There are many ways to estimate the skill of a regression predictive model, but perhaps the most common is to calculate the root mean squared error, abbreviated by the acronym RMSE.
An algorithm that is capable of learning a regression predictive model is called a regression algorithm.

To summarize, fundamentally, classification is about predicting a label and regression is about predicting a quantity.

NOTE: Some algorithms have the word “regression” in their name, such as linear regression and logistic regression, which can make things confusing because linear regression is a regression algorithm whereas logistic regression is a classification algorithm.

Scikit Learn library in python

Now, to apply the classification & regression methods to a model, a preferred tool (or library) is “Sci-kit learn” available in python.

Scikit-learn is probably the most useful library for machine learning in Python. The sklearn library contains a lot of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction.

Please note that sklearn is used to build machine learning models. It should not be used for reading the data, manipulating and summarizing it. There are better libraries for that (e.g. NumPy, Pandas etc.)

The various algorithms (listed below) to be used for classification & regression are to be selected from this library itself.

Classification Algorithms (summary, parameters, pros, cons)

1. Logistic Regression

Logistic regression is a fundamental classification technique. It belongs to the group of linear classifiers and is somewhat similar to polynomial and linear regression. Logistic regression is fast and relatively uncomplicated, and it's convenient for you to interpret the results. Although it's essentially a method for binary classification, it can also be applied to multiclass problems.

Don't confuse this classification algorithm with regression methods for using regression in its title. Logistic regression mostly performs binary classification, so the label outputs are binary. We can also think of logistic regression as a special case of linear regression when the output variable is categorical, where we are using a log of odds as the dependent variable. What is awesome about a logistic regression? It takes a linear combination of features and applies a nonlinear function (sigmoid) to it, so it's a tiny instance of the neural network!

Parameters: (link)
Logistic regression does not demand much tuning of parameters(this is in fact a plus point of this algorithm). But there are few important parameters that we need to know in order to improve results. They are listed below:

- penalty: *{'l1', 'l2', 'elasticnet', 'none'}, default='l2'*
  l2 regularization will make the coefficients of irrelevant features close to zero(not exactly zero) so that they do not affect much on the output variable.
  l1 regularization will make the coefficients of irrelevant variables as zero, so apparently providing feature selection properties.
  elasticnet regularization is a combination of l1 and l2 regularization.

- C: *float, default=1.0*
  Inverse of regularization strength; must be a positive float.Smaller values specify stronger regularization i.e., simpler models i.e., probable underfitting. Larger values specify weak regularization i.e., complex models i.e., probable overfitting.

- solver: *{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'*
  Algorithm to use in the optimization problem.

	Solvers				
Penalties	**'liblinear'**	**'lbfgs'**	**'newton-cg'**	**'sag'**	**'saga'**
Multinomial + L2 penalty	no	yes	yes	yes	yes
OVR + L2 penalty	yes	yes	yes	yes	yes
Multinomial + L1 penalty	no	no	no	no	yes
OVR + L1 penalty	yes	no	no	no	yes
Elastic-Net	no	no	no	no	yes
No penalty ('none')	no	yes	yes	yes	yes
Behaviors					
Penalize the intercept (bad)	yes	no	no	no	no
Faster for large datasets	no	no	no	yes	yes
Robust to unscaled datasets	yes	yes	yes	no	no

Tip:
This algorithm can be used for any classification problem that is preferably binary (it can also perform multi class classification, but binary is preferred). For example you can use it if your output class has 2 outcomes; cancer detection problems, whether a student will pass/fail, default/no default in case of customer taking loan, whether a customer will churn or not, email is spam or not etc.

You can start fitting the model with default parameters, then depending upon the results, you can change certain parameters and view the changed results and decide the range for those parameters.

Pros:
- Outputs a probabilistic interpretation, which can be investigated with P-R curves, AUC-ROC curves etc.
- Has a small number of hyperparameters(usually called as parameters)
- Overfitting can be addressed though regularization.
- Using L1 regularization will provide feature importances.

Cons:
- May overfit when provided with large numbers of features (i.e., when features>records)
- Can only learn linear hypothesis functions so are less suitable to complex relationships between features and target (i.e., preferred when classes are linearly separable)
- Input data might need scaling
- May not handle irrelevant features well, especially if the features are strongly correlated
- Not a very powerful algorithm, and can be easily outperformed by other algorithms.

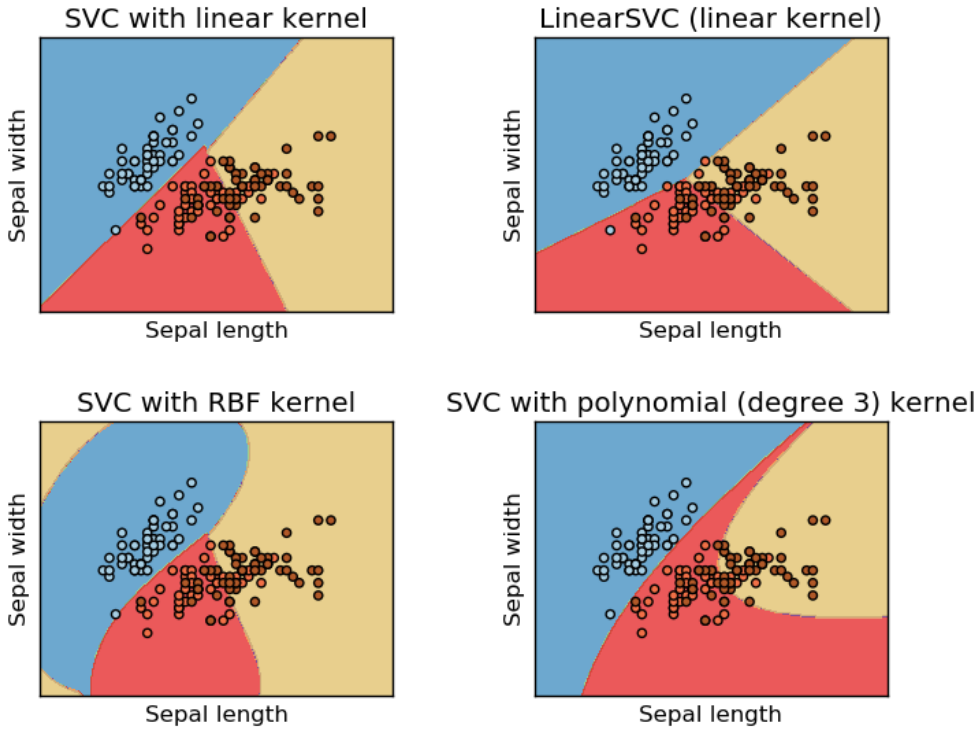
2. SVM (Support Vector Machine)

Ultimately the goal of SVM is to find optimal hyperplanes(or differentiating planes) that best separate the dataset. Typically a hyperplane is chosen such that the ‘margin’ between the plane and data points of different classes on either side is maximized.

Sometimes the data points can’t be separated by straight plane, so SVM needs to map them to a higher dimensional space (using kernels parameter) where they can be split by straight plane (hyperplanes). This looks like a curvy line on the original space, even though it is really a straight thing in a much higher dimensional space. The kernel trick can be used to adapt SVMs so that they become capable of learning non-linear decision boundaries i.e., convert non-separable problems into separable problems.

Parameters: **(link)**
Now, parameters for this algorithm play a crucial part for the results. Few important parameters that have higher impact on model performance are listed below:

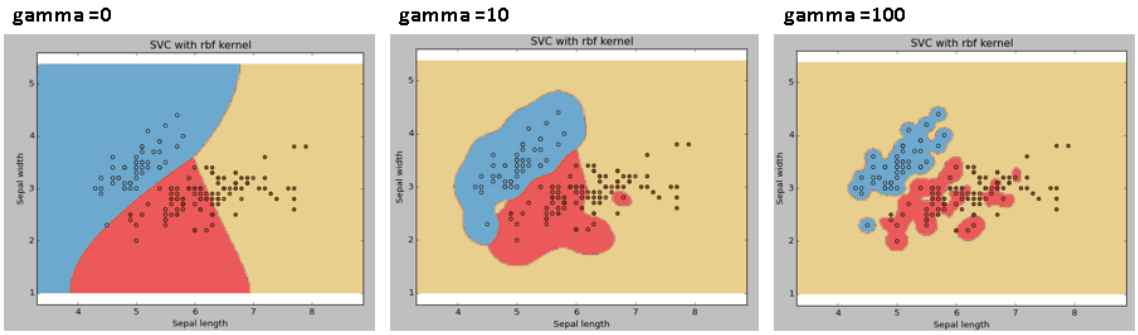
- kernel: *{‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’}, default=’rbf’*
  rbf, poly, sigmoid are preferred when the classes are non-linearly separable, i.e., when the hyperplane is non-linear.
  linear is used when the classes are linearly separable. Also when the number of features is large(say, >1000)



- degree: *int, default=3*
  Use this parameter only if you have set the kernel as “poly”. This is nothing but the degree of the separating curves(boundaries)

- C: *float, default=1.0*
  Regularization parameter. Inverse of regularization strength. C is the penalty parameter of the error term. It controls the trade off between smooth decision boundaries and classifying the training points correctly. Increasing C values may lead to overfitting the training data.

- gamma: *{‘scale’, ‘auto’} or float, default=’scale’*
  Gamma is used when we use the Gaussian RBF kernel. If you use a linear or poly kernel then you do not need gamma, you only need C. Gamma decides how much curvature we want in a decision boundary.
  Gamma high means more curvature. Gamma low means less curvature.
  Intuitively, the gamma parameter defines how far the influence of a single training example reaches, with low values meaning ‘far’ and high values meaning ‘close’.
  The gamma parameters can be seen as the inverse of the radius of influence of samples selected by the model as support vectors. Very high values of gamma may lead to overfitting.



Pros:

- Fairly robust against overfitting, especially in higher dimensional space
- There are many kernels to choose from, and some may perform very well on the dataset in question
- Outliers have less impact. I.e., robust to outliers
- Effective in high-dimensional spaces(i.e., when there are high features. Also when features>records)
- Can be used for both cases when classes are linearly separable & non-linearly separable.

Cons:

- Feature scaling is required
- Hyperparameter tuning is crucial and their meanings are often not intuitive

- Do not scale well to large datasets (difficult to parallelize)
- Takes high training time for large datasets.
- Does not perform well in case of overlapped classes.

3. KNN Classifier

K-Nearest Neighbor (KNN) algorithm is a distance based supervised learning algorithm that is used for solving supervised ML problems. In this algorithm, the class of a new test point is decided based upon the class of its neighbours. I.e., the algorithm observes the nature of the nearest neighbours and decides the class for the new test data point.
The “K” in “KNN” stands for the number of cases that are considered to be "nearest" when you convert the cases as points in an euclidean space.
That means, if we consider k=3, and when a new point has to be assigned a class, it will be on the basis of the classes of the 3 nearest points that surround the new point as per the euclidean distance.
To identify the nearest neighbors we use various techniques of measuring distance, the most common of them being the ‘Euclidean Distance’. This is nothing but the shortest distance.
KNN can be used for both classification and regression predictive problems. For classification, the neighbors of the datapoint all vote for its class. The class with the most votes wins. For regression, the value of the datapoint is determined as the average of its neighbors . However, KNN is more widely used in classification problems in the industry.

Parameters: (link)
Now again, the KNN algorithm does not demand much hyperparameter tuning except for one very important and major parameter (given below). This is because the default value of other parameters is suitable for most of the cases and need not be changed.

- n_neighbors: *int, default=5*
  This is the number of nearest neighbours that are allowed to cast a vote in deciding the class of the new point. Inshort, this is the “K” in “KNN” which is explained above.
  This parameter is very & most important in deciding the model. Therefore it should be selected properly. Neither too low, nor too high. For eg. as we decrease the value of K to 1, our predictions become less stable because the class of new points is judged only from the influence of 1 nearest neighbour. This might cause underfitting.
  Inversely, as we increase the value of K, our predictions become more stable due to majority voting / averaging, and thus, more likely to make more accurate predictions (up to a certain point).
  Eventually, we begin to witness an increasing number of errors. It is at this point we know we have pushed the value of K too far.

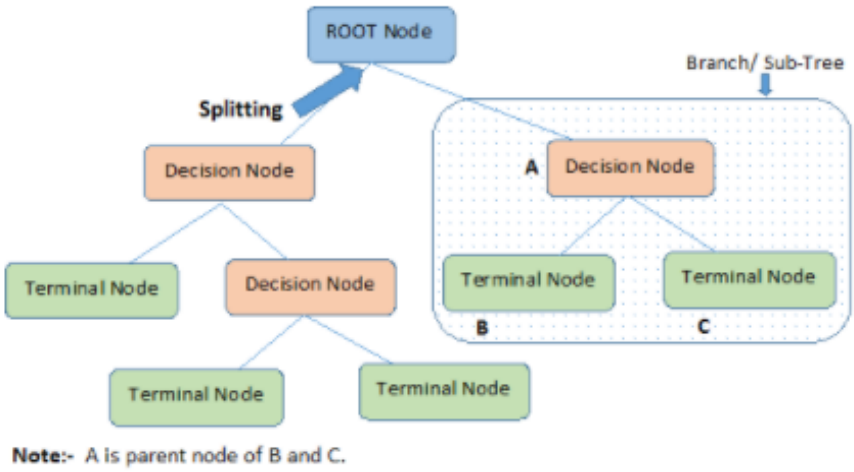
Tip:
You can use it for any classification problem when the dataset is smaller, and has a lesser number of features so that computation time taken by KNN is less. If you do not know the shape of the data and the way output and inputs are related (whether classes can be separated by a line or ellipse or parabola etc.), then you can use KNN without any hesitation.

Pros:
- The algorithm is simple and easy to implement
- Naturally handles multiclass classification and can learn complex decision boundaries
- There's no need to build a heavy model, tune several parameters, or make any additional assumptions
- Only one major parameter. KNN might take some time while selecting the first hyper parameter but after that, rest of the parameters are aligned to it
- The algorithm is versatile. It can be used for both classification (import KNeighboursClassifier) and regression (import KNeighboursRegressor)

Cons:
- Slow for large datasets
- The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase
- Feature scaling is an absolute must (Normalization preferred)
- Sensitive to outliers
- Does not work well on Imbalanced data (when there is over-majority of only one kind of class)

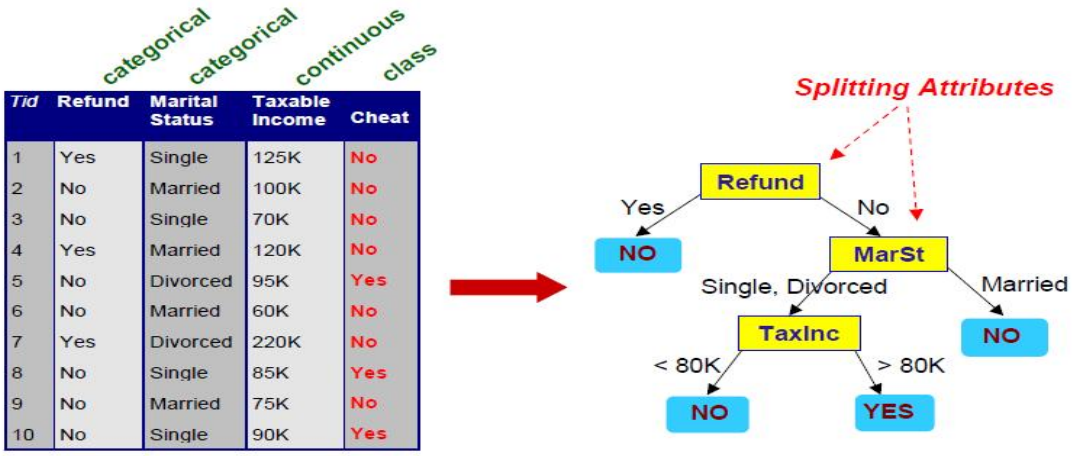
4. Decision Tree Classifier

A Decision Tree is a flowchart like structure, where each node represents a decision, each branch represents an outcome of the decision, and each terminal node provides a label.



Decision Tree Classifier is a simple, common and most-widely used classification technique. It applies a straightforward idea to solve the classification problem. Decision Tree Classifier poses a series of carefully crafted questions about the attributes of the test record. Each time it receives an answer, a follow-up question is asked until a conclusion about the class label of the record is reached.

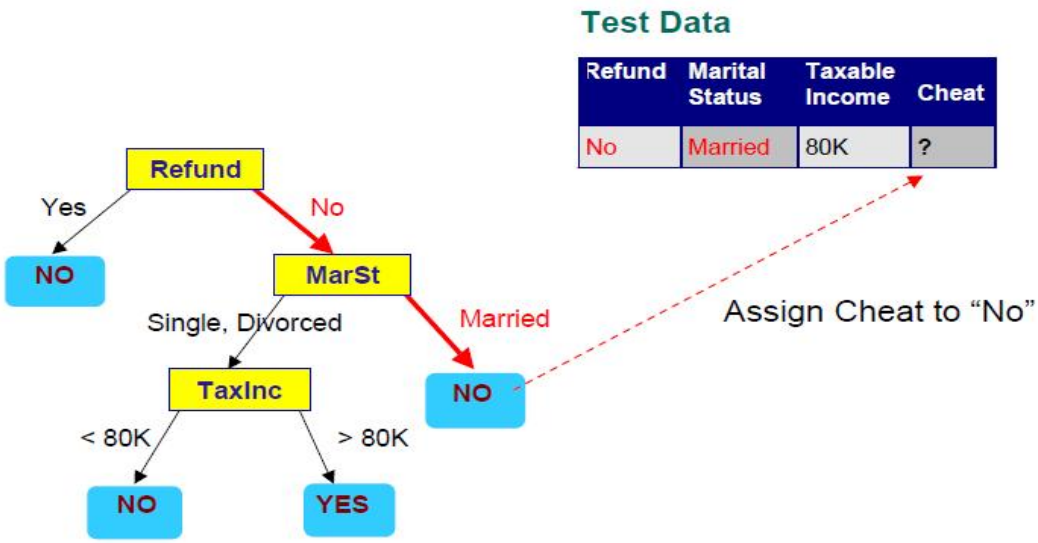
The decision tree classifiers organize a series of test questions and conditions in a tree structure. The following figure shows an example decision tree for predicting whether a person cheats based on certain given features. In the decision tree, the root and internal nodes contain attribute test conditions to separate records that have different characteristics. All the terminal nodes are assigned a class label “Yes” or “No”



Training Data Model: Decision Tree

Once the decision tree has been constructed, classifying a test record is straightforward. Starting from the root node, we apply the test condition to the record and follow the appropriate branch based on the outcome of the test. It then leads us either to another internal node, for which a new test condition is applied, or to a leaf node. When we reach

the leaf node, the class label associated with the leaf node is then assigned to the record, As shown in the following figure, it traces the path in the decision tree to predict the class label of the test record, and the path terminates at a leaf node.



Parameters: [(link)]

This algorithm has some serious and important set of parameters which the user should be well aware of. Listed below are the important parameters that could define the performance of model:

- min_samples_split: *int, default=2*
  The minimum number of samples required to split an internal node. A node will split into a sub-node only if it has a certain (minimum) number of samples. If a particular node has samples less than this parameter's value, then that node would not split further (will act as terminal node or leaf node). Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- min_samples_leaf: *int, default=1*
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least these many (the specified value) training samples in each of the left and right branches. If a sub-node has samples less than the specified value, then its parent node will never split. Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- max_depth: *int, default=None*

The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note that very high depth can lead to overfitting. And very low depth can lead to underfitting.

- max_leaf_nodes: *int, default=None*
  Grow a tree with this maximum number of leaves in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Note that very high values can lead to overfitting. And very low values can lead to underfitting.

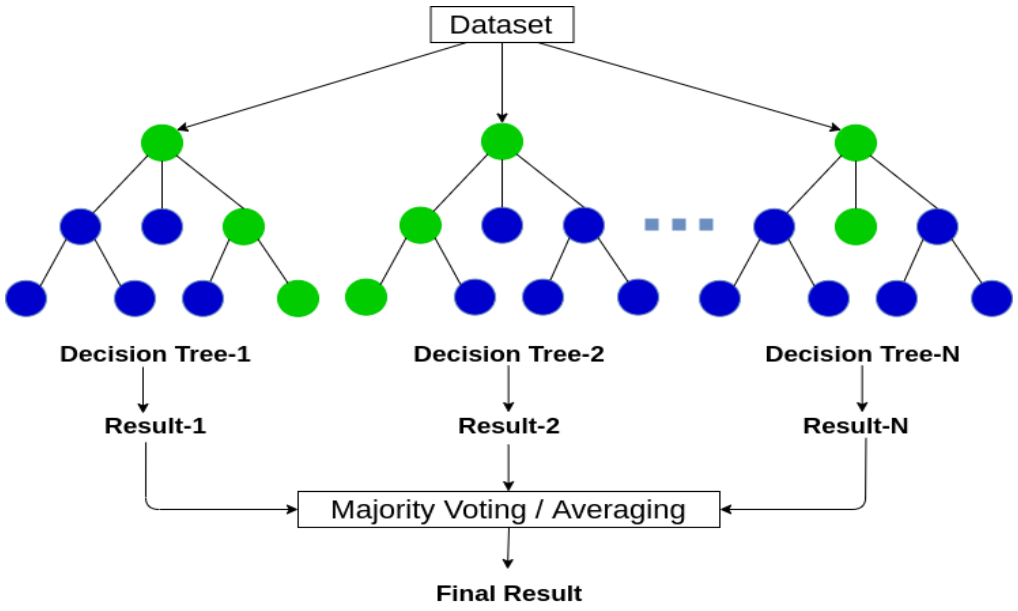
Pros:
- Easy to interpret and explain
- Compared to other algorithms, decision trees requires less effort for data preparation during pre-processing
- Can work with both, numerical and categorical features
- Feature scaling NOT required
- Missing values in the data also do NOT affect the process of building a decision tree to any considerable extent
- Feature importance is already built in due to the way in which decision nodes are built
- Performs well on large datasets
- Can adapt to learn arbitrarily complex hypothesis functions

Cons:
- Prone to overfitting (can be handled with proper tuning)
- A small change in the data can cause a large change in the structure of the decision tree causing instability
- Takes time to train decision trees

5. Random Forest Classifier

Random Forest is a popular supervised machine learning algorithm. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model. It also undertakes dimensional reduction methods, treats missing values, outlier values and other essential steps of data exploration, and does a fairly good job. As we know that a forest is made up of trees and more trees means more robust forest. Similarly, a random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble

method which is better than a single decision tree because it reduces the over-fitting by averaging the result.
In Random Forest, we grow multiple trees as opposed to a single tree in the Decision Tree model. 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) *and in case of regression, it takes the average of outputs by different trees*.



Parameters: (link)
Since this is an algorithm built on a decision tree algorithm, it has the same parameters as decision trees and few new parameters. Listed below are the overall important parameters for random forest algorithm that a user should be well aware about:

- n_estimators: *int, default=100*
  This is the number of decision tree models (Trees) that you want to build in the forest. More the number of trees, the better the accuracy of the model. But, too high values can cause the accuracy to drop. Hence, an appropriate value should be selected by the user.

- min_samples_split: *int or float, default=2*
  The minimum number of samples required to split an internal node. A node will split into a sub-node only if it has a certain (minimum) number of samples. If a particular node has samples less than this parameter’s value, then that node would not split further (will act as terminal node or leaf node). Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- min_samples_leaf: *int or float, default=1*
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least these many (the specified value) training samples in each of the left and right branches. If a sub-node has samples less than the specified value, then its parent node will never split. Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- max_depth: *int, default=None*
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note that very high depth can lead to overfitting. And very low depth can lead to underfitting.

- max_leaf_nodes: *int, default=None*
  Grow a tree with this maximum number of leaves in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Note that very high values can lead to overfitting. And very low values can lead to underfitting.

Pros:
- Highly flexible and generally very accurate
- Naturally assigns feature importance scores, so can handle redundant feature columns
- Overcomes the problem of overfitting by averaging or combining the results of different decision trees
- Scales to large datasets
- Feature scaling not needed
- Can learn non-linear hypothesis functions
- Can handle missing values and outliers very well

Cons:
- Less intuitive
- High complexity
- Takes time to train

Regression Algorithms (summary, parameters, pros, cons)

1. Linear Regression

Linear regression is a basic and commonly used type of predictive analysis. The overall idea of regression is to examine two things: (1) does a set of predictor variables do a good job in predicting an outcome (dependent) variable? (2) Which variables in particular are significant predictors of the outcome variable, and in what way do they–indicated by the magnitude and sign of the beta estimates–impact the outcome variable?
These regression estimates are used to explain the relationship between one dependent variable(y) and one or more independent variables(X).
The simplest form of the regression equation with one dependent and one independent variable is defined by the formula y = c + b*x, where y = estimated dependent variable score, c = constant, b = regression coefficient, and x = score on the independent variable.
To summarize, linear regression is a great tool (when the dependent & independent variables have linear relation), but it is not mostly recommended for most practical use as it over-simplifies real world problems by assuming a linear relationship among variables.

Parameters: (link)
Linear regression algorithm has only 5 hyperparameters. And the default values of these parameters are so setted that users need not bother to look or change those. Hence only providing the link.

Tip:
You can select polynomial regression algorithm(or other regression algorithms) if you know that the relation between variables is non-linear

Pros:
- Simple to implement and easier to interpret
- Very perfect if you know that dependent & independent variables have linear relationship
- Susceptible to overfitting but it can be avoided using dimensionality reduction methods.
- Hyperparameter tuning not required in most cases

Cons:
- Assumes a linear relation between dependent and independent variables *(important)*
- Not good when outliers are present

- The variables should be independent of each other (no multicollinearity)
- Feature scaling is required

2. KNN Regressor

The basic theory of this algorithm is similar to that of the KNN classifier as given above. K nearest neighbors is a simple algorithm that stores all available cases and predicts the numerical target based on a similarity measure (e.g., distance functions). KNN has been used in statistical estimation and pattern recognition already in the beginning of 1970's as a non-parametric technique. Algorithm A simple implementation of KNN regression is to calculate the average of the numerical target of the K nearest neighbors. Another approach uses an inverse distance weighted average of the K nearest neighbors. KNN regression uses the same distance functions as KNN classification.

Parameters: (link)
Now again, the KNN algorithm does not demand much hyperparameter tuning except for one very important and major parameter (given below). This is because the default value of other parameters is suitable for most of the cases and need not be changed.

- n_neighbors: *int, default=5*
  This is the number of nearest neighbours that are allowed to cast a vote in deciding the class of the new point.

Pros:
- The algorithm is simple and easy to implement
- There's no need to build a heavy model, tune several parameters, or make any additional assumptions
- Only one major parameter. KNN might take some time while selecting the first hyper parameter but after that, rest of the parameters are aligned to it
- The algorithm is versatile. It can be used for both classification (import KNeighborsClassifier) and regression (import KNeighborsRegressor)

Cons:
- Slow for large datasets
- The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase
- Feature scaling is an absolute must (Normalization preferred)
- Sensitive to outliers
- Does not work well on Imbalanced data (when there is over-majority of only one kind of class)

3. SVR

Support Vector Regression (SVR) is a supervised learning algorithm that is used to predict discrete values. Support Vector Regression uses the same principle as the SVM (Support Vector Machine as discussed above in the classification section). The basic idea behind SVR is to find the best fit line. In SVR, the best fit line is the hyperplane that has the maximum number of points.
Unlike other Regression models that try to minimize the error between the real and predicted value, the SVR tries to fit the best line within a threshold value. The threshold value is the distance between the hyperplane and boundary line.
In-short, terminologies in SVR that relate with that in SVM are as below:

1) Hyperplane: In SVM this is basically the separation line between the data classes. Although in SVR we define it as the line that will will help us predict the continuous value or target value

2) Kernel: The function used to map a lower dimensional data into a higher dimensional data. (discussed in more detail in parameters)

3) Boundary line: In SVM there are two lines other than Hyperplane which creates a margin. Think of it as two lines which are at a distance of ‘e’ (formally called epsilon) from the hyperplane on either side. The support vectors can be on the Boundary lines or outside it. This boundary line separates the two classes. In SVR the concept is the same. Thus the decision boundary is a margin of tolerance that is We take only those points who are within this boundary. And we set this boundary range using a parameter called epsilon. Or in simple terms, using epsilon we take only those points which have the least suitable error rate. Thus giving us a better fitting model.

4) Support vectors: These are the data points which are closest to the boundary. The distance of the points is minimum or least from the boundary.

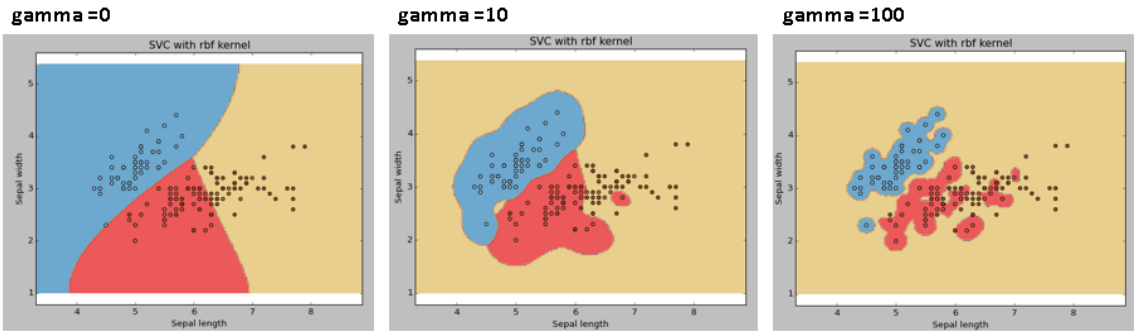
Parameters: (link)
Now just like in SVM, SVR also has a set of few important parameters that require attention from the user. They are as below:

- kernel: *{‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’}, default=’rbf’*
  rbf, poly, sigmoid are preferred when the hyperplane is non-linear.
  linear is used when the hyperplane is linear. Also when the number of features is large(say, >1000)

- epsilon: *float, default=0.1*
  Intuitively, this is the distance between the hyperplane and either of the boundary lines. I.e., the allowable error margin. epsilon=0 forces the regression to penalize every point that is not exactly on the regression line (hyperplane). Whereas

epsilon>0 allows an indifference margin around the regression in which a deviation will not be counted as an error.

- gamma: *{'scale', 'auto'} or float, default='scale'*
  gamma decides how much curvature we want in a decision boundary.
  gamma high means more curvature.
  gamma low means less curvature.



- C: *float, default=1.0*
  Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty. Increasing C values may lead to overfitting the training data.

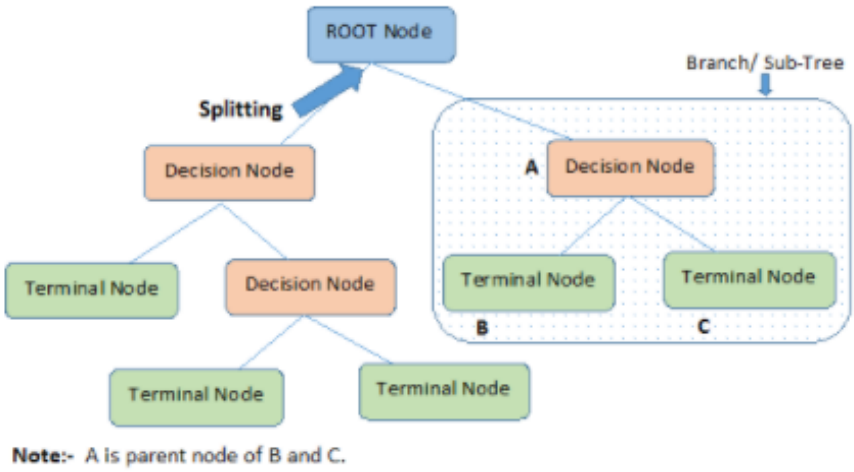
- degree: *int, default=3*
  Use this parameter only if you have set the kernel as "poly". This is nothing but the degree of hyperplane in SVR

Pros:
- It is robust to outliers
- Decision model can be easily updated
- It has excellent generalization capability, with high prediction accuracy
- Works very well on non-linear problems as well

Cons:
- They are not suitable for large datasets
- Feature scaling is a must (Standardization preferred)
- Less learning resources available compared to other algorithms

4. Decision Tree Regressor

A Decision Tree is a flowchart like structure, where each node represents a decision, each branch represents an outcome of the decision, and each terminal node provides a prediction.



This algorithm is exactly the same as the Decision Tree Classifier algorithm given in the classification section. The only difference is that, this one is used when the target variables are continuous (regression problem)

Parameters: (link)
This algorithm has some serious and important set of parameters which the user should be well aware of. Listed below are the important parameters that could define the performance of model:

- min_samples_split: *int or float, default=2*
  The minimum number of samples required to split an internal node. A node will split into a sub-node only if it has a certain (minimum) number of samples. If a particular node has samples less than this parameter's value, then that node would not split further (will act as terminal node or leaf node). Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- min_samples_leaf: *int or float, default=1*
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least these many (the specified value) training samples in each of the left and right branches. If a sub-node has samples less than the specified value, then its parent node will never split. Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- max_depth: *int, default=None*
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples. Note that very high depth can lead to overfitting. And very low depth can lead to underfitting.

- max_leaf_nodes: *int, default=None*
  Grow a tree with this maximum number of leaves in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Note that very high values can lead to overfitting. And very low values can lead to underfitting.

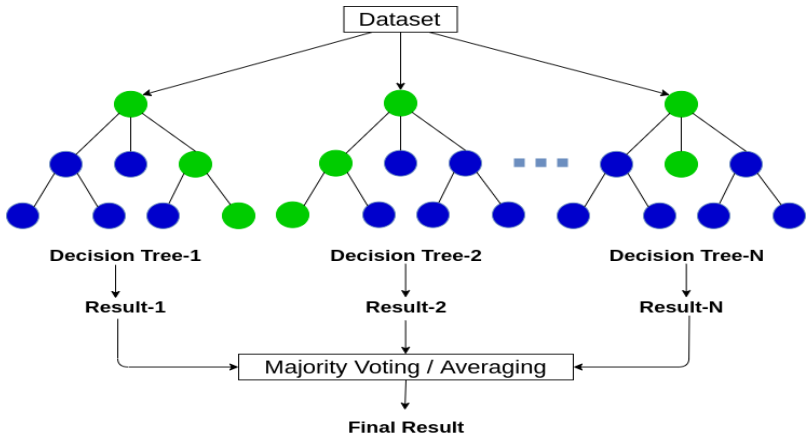
Pros:
- Easy to interpret and explain
- Compared to other algorithms, decision trees requires less effort for data preparation during pre-processing
- Can work with both, numerical and categorical features
- Feature scaling NOT required
- Missing values in the data also do NOT affect the process of building a decision tree to any considerable extent
- Feature importance is already built in due to the way in which decision nodes are built
- Performs well on large datasets
- Can adapt to learn arbitrarily complex hypothesis functions

Cons:
- Prone to overfitting (can be handled with proper tuning)
- A small change in the data can cause a large change in the structure of the decision tree causing instability
- Takes time to train decision trees

5. Random Forest Regressor

Random Forest is a popular supervised machine learning algorithm. It can be used for both Classification and Regression problems in ML. It is based on the concept of ensemble learning, which is a process of combining multiple classifiers to solve a complex problem and to improve the performance of the model. It also undertakes dimensional reduction methods, treats missing values, outlier values and other essential steps of data exploration, and does a fairly good job. As we know that a forest is made up of trees and more trees means more robust forest. Similarly, a random forest algorithm creates decision trees on data samples and then gets the prediction from each of them and finally selects the best solution by means of voting. It is an ensemble method which is better than a single decision tree because it reduces the over-fitting by averaging the result.
In Random Forest, we grow multiple trees as opposed to a single tree in the Decision Tree model. 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) *and in case of regression, it takes the average of outputs by different trees*.



Parameters: (link)

Since this is an algorithm built on a decision tree algorithm, it has the same parameters as decision trees and few new parameters. Listed below are the overall important parameters for random forest algorithm that a user should be well aware about:

- n_estimators: *int, default=100*
  This is the number of decision tree models (Trees) that you want to build in the forest. More the number of trees, the better the accuracy of the model. But, too high values can cause the accuracy to drop. Hence, an appropriate value should be selected by the user.

- min_samples_split: *int or float, default=2*
  The minimum number of samples required to split an internal node. A node will split into a sub-node only if it has a certain (minimum) number of samples. If a particular node has samples less than this parameter's value, then that node would not split further (will act as terminal node or leaf node). Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- min_samples_leaf: *int or float, default=1*
  The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least these many (the specified value) training samples in each of the left and right branches. If a sub-node has samples less than the specified value, then its parent node will never split. Note that, higher value of this parameter controls overfitting. But, too high values can lead to underfitting.

- max_depth: *int, default=None*
  The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

Note that very high depth can lead to overfitting. And very low depth can lead to underfitting.

- max_leaf_nodes: *int, default=None*
  Grow a tree with this maximum number of leaves in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes. Note that very high values can lead to overfitting. And very low values can lead to underfitting.

Pros:
- Highly flexible and generally very accurate
- Naturally assigns feature importance scores, so can handle redundant feature columns
- Overcomes the problem of overfitting by averaging or combining the results of different decision trees
- Scales to large datasets
- Feature scaling not needed
- Can learn non-linear hypothesis functions
- Can handle missing values and outliers very well

Cons:
- Less intuitive
- High complexity
- Takes time to train

Selection summary (which algorithm to select ?)

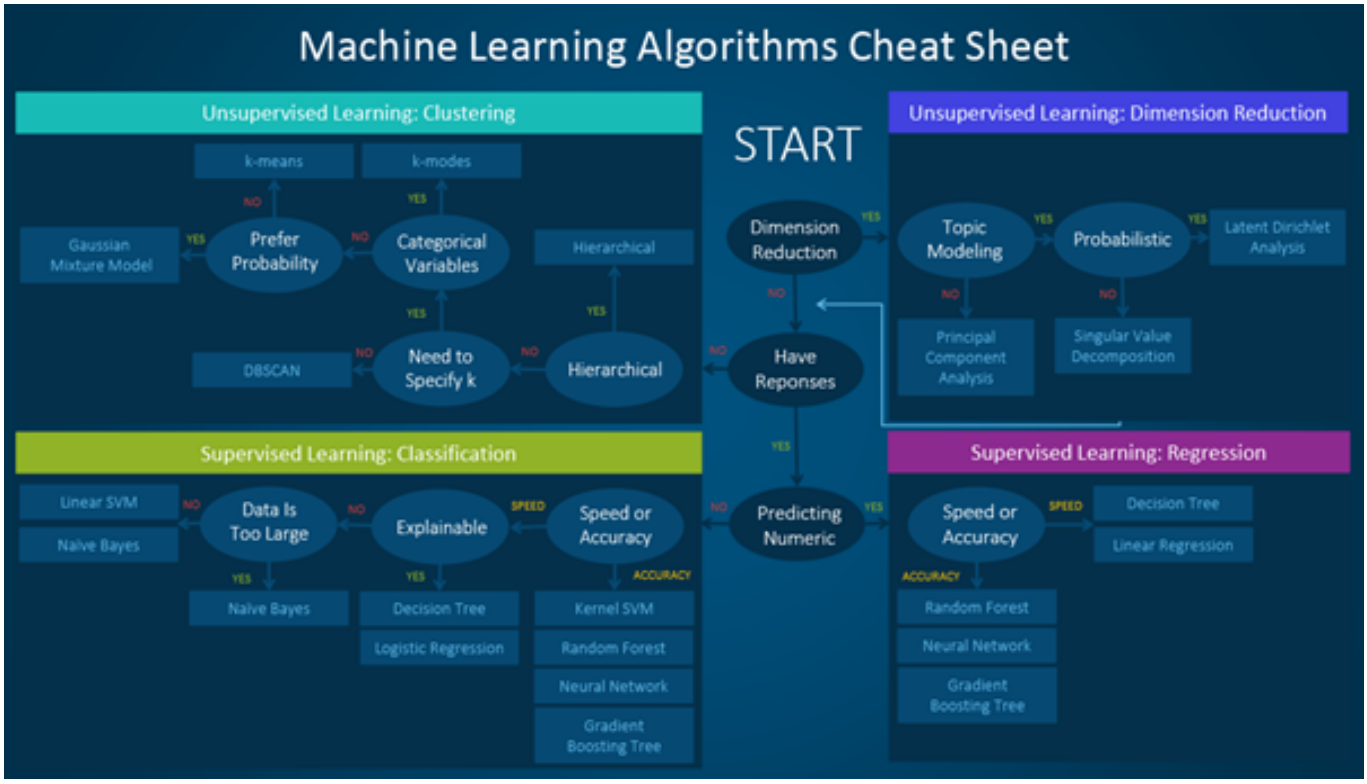
By now, we have seen a total of 10 primary algorithms that can be used for classification and regression problems (5 each) in machine learning. But, which algorithm out of these 5 to actually select when it comes to a particular type of problem ?

Now, we need to understand that this is the question which is faced and suffered by every single data scientist dealing with classification or regression problems. Because, the answer to this question does depend on a lot of factors. As a matter of fact, there is no straightforward and clear way to determine this (in machine learning terminology, this condition is often termed as, "*No free lunch*").
But after thoroughly understanding and analysing the dataset, one can refer to some of the factors affecting the choice of a model, which are:

- The accuracy of the model.
- The interpretability of the model.
- The complexity of the model.
- The scalability of the model.

- How long does it take to build, train, and test the model?
- How long does it take to make predictions using the model?
- Does the model meet the business goal?

Apart from this, the figure below is a quick cheat-sheet to accompany the selection of algorithm based on few criteria



While these are some factors that can be referred to, based on the algorithms we have seen above, the tables below again give an idea for a quick algorithm selection guide based on a few more factors.

Note: Wherever there is a tick-mark in these tables below, it means that the particular algorithm is suitable for that particular factor condition. However, wherever there are blank spaces (no tick-marks), it does not mean that the particular algorithm cannot be used for that factor condition. It can be used, but just the case that they are not made for it, or that there are better algorithms for that particular factor condition. (just like, we can still have soup with a normal spoon, but we choose to use the spoons especially made for having soup, for a better experience)

For Classification analysis	Factors →	Size of training data		Accuracy	Interpretability	Speed	Takes some training time	Linearly		n>>records
	Algorithms ↓	large	small					separable	non-separable	
	Logistic Regression	✓	✓		✓	✓		✓		
	SVM		✓	✓			✓	✓	✓	✓
	KNN	✓	✓		✓		✓	✓	✓	✓
	Decision tree classifier	✓	✓		✓	✓		✓	✓	
	Random forest classifier	✓	✓	✓			✓	✓	✓	

For Regression analysis	Factors →	Size of training data		Accuracy	Interpretability	Speed	Takes some training time	Linearity		n>>records
	Algorithms ↓	large	small					Linear	non-linear	
	Linear Regression		✓		✓	✓		✓		
	KNN	✓	✓		✓		✓	✓	✓	✓
	SVR		✓	✓			✓	✓	✓	✓
	Decision tree regressor	✓	✓		✓	✓		✓	✓	
	Random forest regressor	✓	✓	✓			✓	✓	✓	

Some other important algorithms

1. XGBoost

XGBoost is an algorithm that has recently been dominating applied machine learning (and Kaggle competitions) for structured or tabular data. It is an implementation of gradient boosted decision trees designed for speed and performance.

XGBoost stands for e**X**treme **G**radient **Boost**ing.
It is a software library (just like sci-kit learn) that you can download and install on your machine, then access from a variety of interfaces (like python or a model in sci-kit learn). The library is laser focused on computational speed and model performance. Nevertheless, it does offer a number of advanced features.

Model Features:
The implementation of the model supports the features of the scikit-learn and R implementations, with new additions like regularization. Three main forms of gradient boosting are supported:

- Gradient Boosting algorithm also called gradient boosting machine including the learning rate.
- Stochastic Gradient Boosting with sub-sampling at the row, column and column per split levels.
- Regularized Gradient Boosting with both L1 and L2 regularization.

System Features:
The library provides a system for use in a range of computing environments, not least:

- Parallelization of tree construction using all of your CPU cores during training.
- Distributed Computing for training very large models using a cluster of machines.
- Out-of-Core Computing for very large datasets that don't fit into memory.
- Cache Optimization of data structures and algorithms to make best use of hardware.

Algorithm Features:
The implementation of the algorithm was engineered for efficiency of compute time and memory resources. A design goal was to make the best use of available resources to train the model. Some key algorithm implementation features include:

- Sparse Aware implementation with automatic handling of missing data values.
- Block Structure to support the parallelization of tree construction.
- Continued Training so that you can further boost an already fitted model on new data.

The XGBoost library implements the gradient boosting decision tree algorithm.
This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.

Boosting is an ensemble technique where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the AdaBoost algorithm that weights data points that are hard to predict.

Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.

This approach supports both regression and classification predictive modeling problems.

Parameters: (link)
Now, XGBoost has a very wide range of parameters. I.e., actually too many parameters. All these parameters can be viewed in the link provided above. But here, we will be going through some main and important parameters that primarily define the performance of the model.

- n_estimator: *int, default=100*
  This is the number of models you want to include in the XGBoost. Too low values will lead to underfitting, whereas too high values will cause overfitting. Also, large values will naturally take a longer time as compared to lower values, to train the model. Typical value range is 100-1000. This parameter has a close relation with another parameter, “learning_rate” which we will see next.

- learning_rate: *float, default=0.1*
  This is the step size shrinkage used in update to prevent overfitting. After each boosting step, we can directly get the weights of new features, and ‘learning_rate’ shrinks the feature weights to make the boosting process more conservative. Now this means each tree (model) we add to the ensemble helps us less. So, we can set a higher value for ‘n_estimators’ without overfitting. In general, a small learning rate and a large number of estimators will yield more accurate XGBoost models. The parameter range is [0, 1]. But usually, 0.01-0.3 range is advised.

- early_stopping_rounds: int, default
  This is an approach to training complex machine learning models to avoid overfitting. It works by monitoring the performance of the model that is being trained on a separate test dataset and stopping the training procedure once the performance on the test dataset has not improved after a fixed number of training iterations. It avoids overfitting by attempting to automatically select the inflection point where performance on the test dataset starts to decrease while performance on the training dataset continues to improve as the model starts to overfit.
  This parameter is required to be called while fitting the model (and not while creating an instance of the algorithm as usually the case with other parameters). Also, you need to set aside some data for calculating the validation score, this is done by setting one more parameter, “eval_set”.

  Eg.: model.fit(X_train, y_train, early_stopping_rounds=5, eval_set=[(X_valid, y_valid)])

  In general, this parameter offers a way to automatically find the ideal value of number of estimators

- gamma: *float, default=0*

A node (in the base model, decision tree) is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split. The larger gamma is, the more conservative the algorithm will be.

- max_depth: *int, default=3*
  Maximum depth of a base-model tree. Increasing this value will make the model more complex and more likely to overfit.

- reg_alpha: *float, default=0*
  L1 regularization term on weights. Increasing this value will make the model more conservative.

- reg_lambda: *float, default=1*
  L2 regularization term on weights. Increasing this value will make the model more conservative.

- n_jobs: *int, default=1*
  This parameter is helpful when the datasets are large and runtime is a crucial factor. This parameter uses parallelism to build the models, hence reducing in the training time. It is common to set this value equal to the number of cores on your machine. Setting the value to -1 will utilize all cores. Note that this parameter is only responsible for the runtime factor and not for the performance of the model in terms of accuracy.

Pros:
- Robust to missing data, highly correlated features and irrelevant features in much the same way as random forest
- Naturally assigns feature importance scores
- In Kaggle competitions at least, XGBoost appears to be the method of choice, with slightly better performance than random forest
- Feature scaling not required
- Can learn non-linear hypothesis function
- Handles large sized datasets well
- Good Execution speed

Cons:
- Difficult interpretation
- Has many hyperparameters to tune, so model development may not be as fast
- Overfitting is possible if parameters are not tuned properly

*****