MACHINE LEARNING WORKSHEET SET_5

1) R-squared or Residual Sum of Squares (RSS) which one of these two is a better measure of goodness of fit model in regression and why?

Ans: The residual sum of squares (RSS) is a statistical technique used to measure the amount of variance in a data set that is not explained by a regression model itself. Instead, it estimates the variance in the residuals, or error term.

Linear regression is a measurement that helps determine the strength of the relationship between a dependent variable and one or more other factors, known as independent or explanatory variables.

- The residual sum of squares (RSS) measures the level of variance in the error term, or residuals, of a regression model.
- The smaller the residual sum of squares, the better your model fits your data; the greater the residual sum of squares, the poorer your model fits your data.
- A value of zero means your model is a perfect fit.
- Statistical models are used by investors and portfolio managers to track an investment's price and use that data to predict future movements.
- The RSS is used by financial analysts in order to estimate the validity of their econometric models.

How to Calculate the Residual Sum of Squares

$$RSS = \sum^{n}_{i=1} (y^{i} - f(x_i))^2$$

Where:

y_i = *the* i^{th} *value of the variable to be predicted*

$f(x_i)$ = *predicted value of* y_i

n = *upper limit of summation*

Residual Sum of Squares (RSS) vs. Residual Standard Error (RSE)

The residual standard error (RSE) is another statistical term used to describe the difference in standard deviations of observed values versus predicted values as shown by points in a regression analysis. It is a goodness-of-fit

measure that can be used to analyze how well a set of data points fit with the actual model.

RSE is computed by dividing the RSS by the number of observations in the sample less 2, and then taking the square root: $RSE = [RSS/(n-2)]^{1/2}$

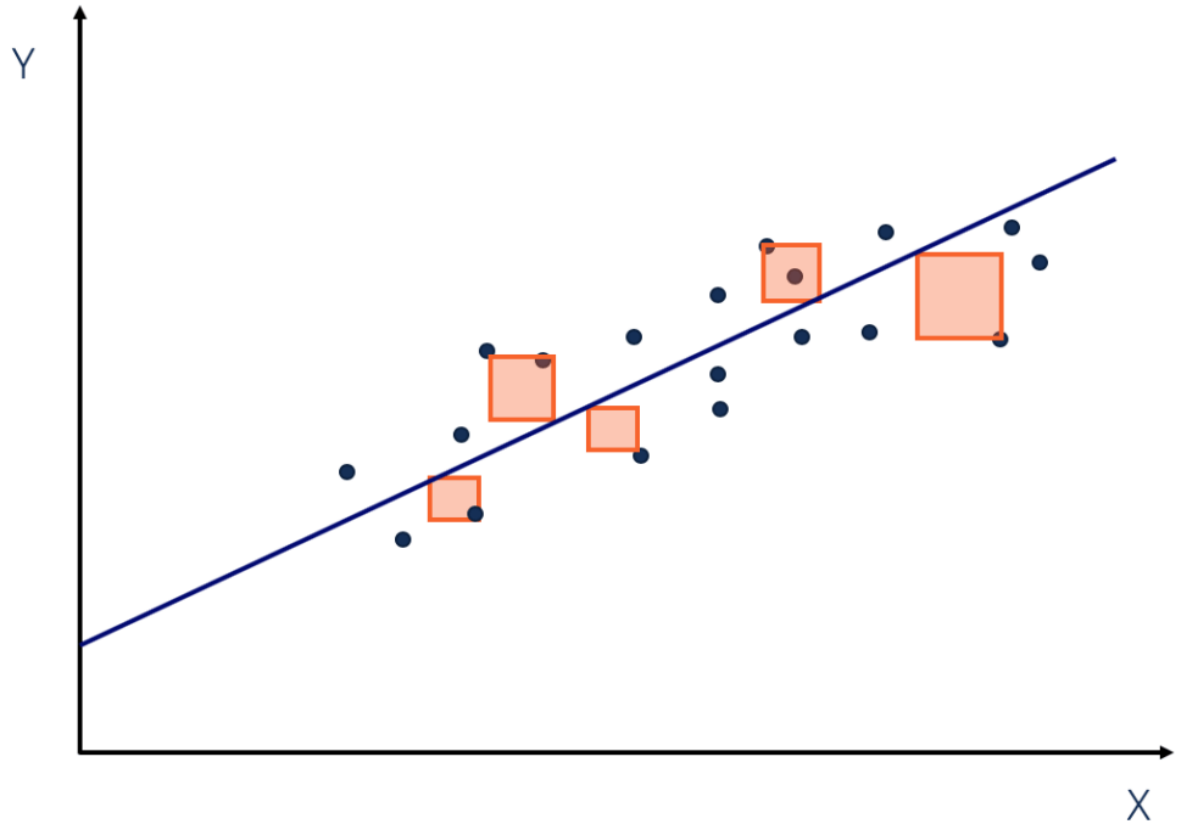
Special Considerations

Financial markets have increasingly become more quantitatively driven; as such, in search of an edge, many investors are using advanced statistical techniques to aid in their decisions. Big data, machine learning, and artificial intelligence applications further necessitate the use of statistical properties to guide contemporary investment strategies. The residual sum of squares—or RSS statistics—is one of many statistical properties enjoying a renaissance.

Statistical models are used by investors and portfolio managers to track an investment's price and use that data to predict future movements. The study—called regression analysis—might involve analyzing the relationship in price movements between a commodity and the stocks of companies engaged in producing the commodity.

2) What are TSS (Total Sum of Squares), ESS (Explained Sum of Squares) and RSS (Residual Sum of Squares) in regression. Also mention the equation relating these three metrics with each other.

Ans: Sum of squares (SS) is a statistical tool that is used to identify the dispersion of data as well as how well the data can fit the model in regression analysis. The sum of squares got its name because it is calculated by finding the sum of the squared differences.



This image is only for illustrative purposes.

The sum of squares is one of the most important outputs in regression analysis. The general rule is that a smaller sum of squares indicates a better model, as there is less variation in the data.

In finance, understanding the sum of squares is important because linear regression models are widely used in both theoretical and practical finance.

Types of Sum of Squares

In regression analysis, the three main types of sum of squares are the total sum of squares, regression sum of squares, and residual sum of squares.

1. Total sum of squares

The total sum of squares is a variation of the values of a dependent variable from the sample mean of the dependent variable. Essentially, the total sum of squares quantifies the total variation in a sample. It can be determined using the following formula:

$$TSS = \sum_{i=1}^{n} (y_i - \bar{y})^2$$

Where:

- y_i – the value in a sample
- $\bar{y}$ – the mean value of a sample

2. Regression sum of squares (also known as the sum of squares due to regression or explained sum of squares)

The regression sum of squares describes how well a regression model represents the modeled data. A higher regression sum of squares indicates that the model does not fit the data well.

The formula for calculating the regression sum of squares is:

$$SSR = \sum_{i=1}^{n} (\hat{y}_i - \bar{y})^2$$

Where:

- $\hat{y}_i$ – the value estimated by the regression line
- $\bar{y}$ – the mean value of a sample

3. Residual sum of squares (also known as the sum of squared errors of prediction)

The residual sum of squares essentially measures the variation of modeling errors. In other words, it depicts how the variation in the dependent variable in a regression model cannot be explained by the model. Generally, a lower residual sum of squares indicates that the regression model can better explain the data, while a higher residual sum of squares indicates that the model poorly explains the data.

The residual sum of squares can be found using the formula below:

$$SSE = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Where:

- y_i – the observed value
- $\hat{y}_i$ – the value estimated by the regression line

The relationship between the three types of sum of squares can be summarized by the following equation:

$$TSS = SSR + SSE$$

Additional Resources

Thank you for reading CFI's guide to Sum of Squares. To keep learning and advancing your career, the following CFI resources will be helpful:

3) What is the need of regularization in machine learning?

Ans: TWhat is Regularization in Machine Learning?

Regularization refers to techniques that are used to calibrate machine learning models in order to minimize the adjusted loss function and prevent overfitting or underfitting.

Figure 5: Regularization on an over-fitted model

Using Regularization, we can fit our machine learning model appropriately on a given test set and hence reduce the errors in it.

Regularization Techniques

There are two main types of regularization techniques: Ridge Regularization and Lasso Regularization.

Figure 6: Regularization techniques

Ridge Regularization :

Also known as Ridge Regression, it modifies the over-fitted or under fitted models by adding the penalty equivalent to the sum of the squares of the magnitude of coefficients.

This means that the mathematical function representing our machine learning model is minimized and coefficients are calculated. The magnitude of coefficients is squared and added. Ridge Regression performs regularization by shrinking the coefficients present. The function depicted below shows the cost function of ridge regression :

Figure 7: Cost Function of Ridge Regression

In the cost function, the penalty term is represented by Lambda λ. By changing the values of the penalty function, we are controlling the penalty term. The higher the penalty, it reduces the magnitude of coefficients. It shrinks the parameters. Therefore, it is used to prevent multicollinearity, and it reduces the model complexity by coefficient shrinkage.

Consider the graph illustrated below which represents Linear regression :

Figure 8: Linear regression model

Cost function = Loss + λ x∑‖w‖^2

For Linear Regression line, let's consider two points that are on the line,

Loss = 0 (considering the two points on the line)

λ= 1

w = 1.4

Then, Cost function = 0 + 1 x 1.42

= 1.96

For Ridge Regression, let's assume,

Loss = 0.32 + 0.22 = 0.13

λ = 1

w = 0.7

Then, Cost function = 0.13 + 1 x 0.72

= 0.62

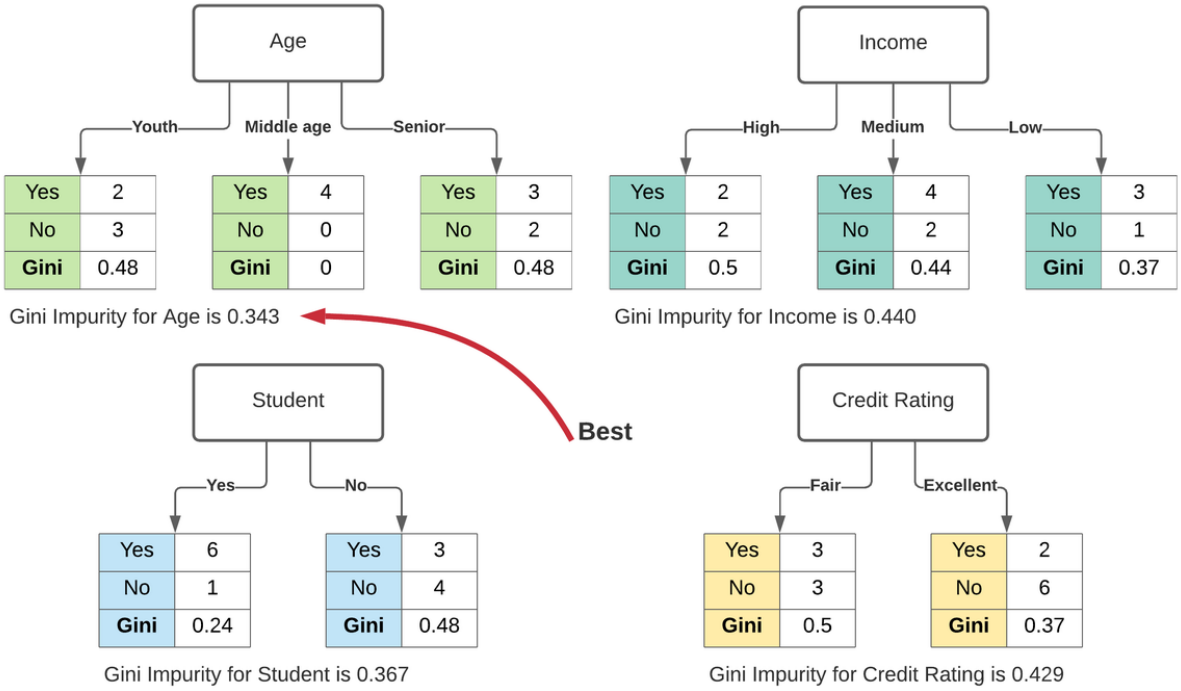
Figure 9: Ridge regression model

Comparing the two models, with all data points, we can see that the Ridge regression line fits the model more accurately than the linear regression line.

Figure 10: Optimization of model fit using Ridge Regression

4) What is Gini–impurity index?

Ans: Gini Impurity is a measurement used to build Decision Trees to determine how the features of a dataset should split nodes to form the tree. More precisely, the Gini Impurity of a dataset is a number between 0-0.5, which indicates the likelihood of new, random data being misclassified if it were given a random class label according to the class distribution in the dataset.



For example, say you want to build a classifier that determines if someone will default on their credit card. You have some labeled data with features, such as bins for age, income, credit rating, and whether or not each person is a student. To find the best feature for the first split of the tree – the root node – you could calculate how poorly each feature divided the data into the correct class, default ("yes") or didn't default ("no"). This calculation would measure the **impurity** of the split, and the feature with the lowest impurity would

determine the best feature for splitting the current node. This process would continue for each subsequent node using the remaining features.

In the image above, ??? has minumum gini impurity, so ??? is selected as the root in the decision tree.

Mathematical definition

Consider a dataset ? that contains samples from ? classes. The probability of samples belonging to class ? at a given node can be denoted as ??. Then the Gini Impurity of ? is defined as: $????(?)=1-\sum ?=1???2$

The node with uniform class distribution has the highest impurity. The minimum impurity is obtained when all records belong to the same class. Several examples are given in the following table to demonstrate the Gini Impurity computation.

	Count		Probability		Gini Impurity
	$?1$	$?2$	$?1$	$?2$	$1-?12-?22$
Node A	0	10	0	1	$1-02-12=0$
Node B	3	7	0.3	0.7	$1-0.32-0.72=0.42$
Node C	5	5	0.5	0.5	$1-0.52-0.52=0.5$

5) Are unregularized decision-trees prone to overfitting? If yes, why?

Ans: Decision trees are a type of model used for both classification and regression. Trees answer sequential questions which send us down a certain route of the tree given the answer. The model behaves with “if this than that” conditions ultimately yielding a specific result. This is easy to see with the image below which maps out whether or not to play golf.

The flow of this tree works downward beginning at the top with the outlook. The outlook has one of three options: sunny, overcast, or rainy. If sunny, we travel down to the next level. Will it be windy? True or false? If true, we choose not to play golf that day. If false we choose to play. If the outlook was changed to overcast, we would end there

and decide to play. If the outlook was rainy, we would then look at the humidity. If the humidity was high we would not play, if the humidity is normal we would play.

Tree depth is an important concept. This represents how many questions are asked before we reach our predicted classification. We can see that the deepest the tree gets in the example above is two. The sunny and rainy routes both have a depth of two. The overcast route only has a depth of one, although the overall tree depth is denoted by its longest route. Thus, this tree has a depth of two.

Advantages to using decision trees:

1. Easy to interpret and make for straightforward visualizations.

2. The internal workings are capable of being observed and thus make it possible to reproduce work.

3. Can handle both numerical and categorical data.

4. Perform well on large datasets

5. Are extremely fast

Disadvantages of decision trees:

1. Building decision trees require algorithms capable of determining an optimal choice at each node. One popular algorithm is the Hunt's algorithm. This is a greedy model, meaning it makes the most optimal decision at each step, but does not take into account the global optimum. What does this mean? At each step the algorithm chooses the best result. However, choosing the best result at a given step does not ensure you will be headed down the route that will lead to the

optimal decision when you make it to the final node of the tree, called the leaf node.

2. Decision trees are prone to overfitting, especially when a tree is particularly deep. This is due to the amount of specificity we look at leading to smaller sample of events that meet the previous assumptions. This small sample could lead to unsound conclusions. An example of this could be predicting if the Boston Celtics will beat the Miami Heat in tonight's basketball game. The first level of the tree could ask if the Celtics are playing home or away. The second level might ask if the Celtics have a higher win percentage than their opponent, in this case the Heat. The third level asks if the Celtic's leading scorer is playing? The fourth level asks if the Celtic's second leading scorer is playing. The fifth level asks if the Celtics are traveling back to the east coast from 3 or more consecutive road games on the west coast. While all of these questions may be relevant, there may only be two previous games where the conditions of tonights game were met. Using only two games as the basis for our classification would not be adequate for an informed decision. One way to combat this issue is by setting a max depth. This will limit our risk of overfitting; but as always, this will be at the expense of error due to bias. Thus if we set a max depth of three, we would only ask if the game is home or away, do the Celtics have a higher winning percentage than their opponent, and is their leading scorer playing. This is a simpler model with less variance sample to sample but ultimately will not be a strong predictive model.

Ideally, we would like to minimize both error due to bias and error due to variance. Enter random forests. Random forests mitigate this problem well. A random forest is simply a collection of decision trees whose results are aggregated into one final result. Their ability to limit overfitting without substantially increasing error due to bias is why they are such powerful models.

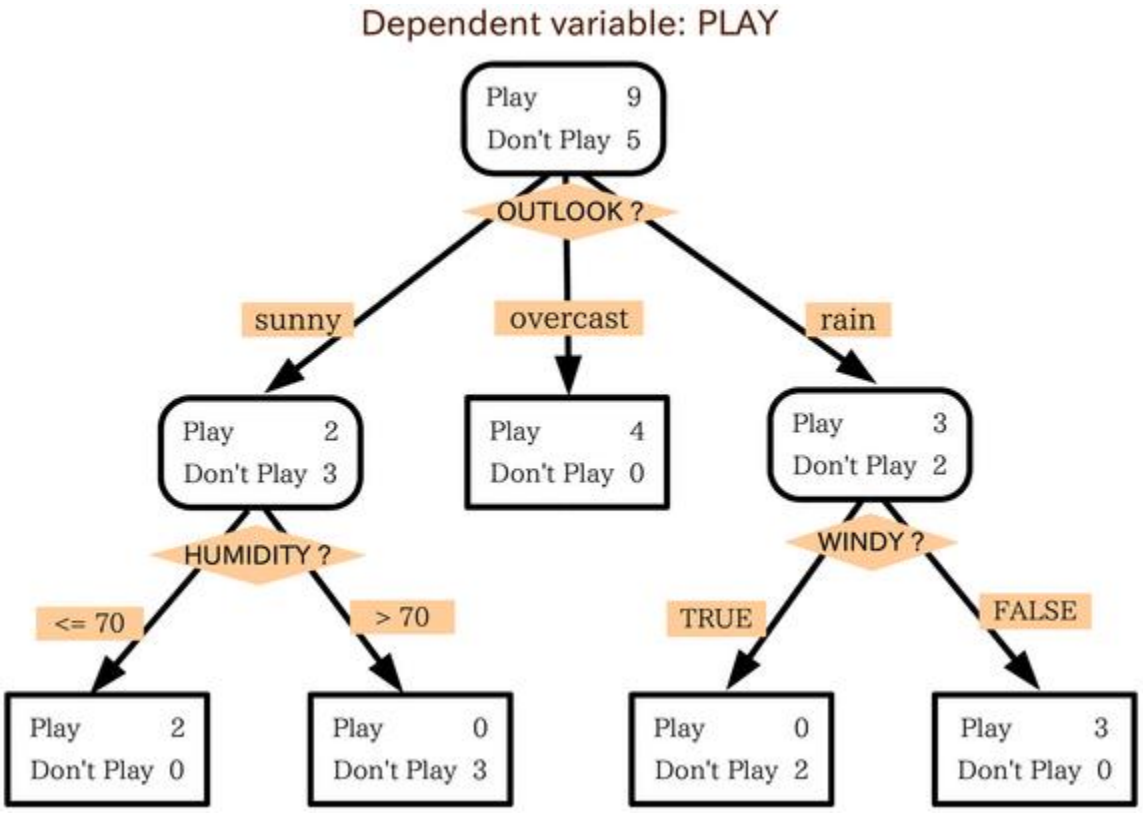
One way Random Forests reduce variance is by training on different samples of the data. A second way is by using a random subset of features. This means if we have 30 features, random forests will only use a certain number of those features in each model, say five. Unfortunately, we have omitted 25 features that could be useful. But as stated, a random forest is a collection of decision trees. Thus, in each tree we can utilize five random features. If we use many trees in our forest, eventually many or all of our features will have been included. This inclusion of many features will help limit our error due to bias and error due to variance. If features weren't chosen randomly, base trees in our forest could become highly correlated. This is because a few features could be particularly predictive and thus, the same features would be chosen in many of the base trees. If many of these trees included the same features we would not be combating error due to variance.

With that said, random forests are a strong modeling technique and much more robust than a single decision tree. They aggregate many decision trees to limit overfitting as well as error due to bias and therefore yield useful results.

6) What is an ensemble technique in machine learning?

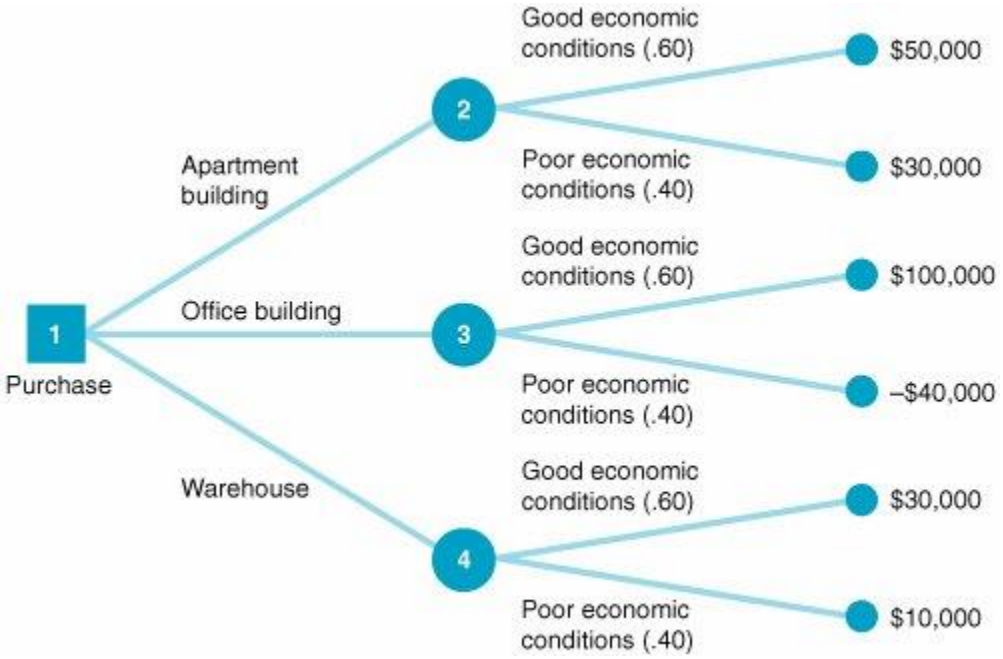
Ans: Ensemble Methods, what are they? **Ensemble methods** is a machine learning technique that combines several base models in order to produce one optimal predictive model. To better understand this definition lets take a step back into ultimate goal of machine learning and model building. This is going to make more sense as I dive into specific examples and why Ensemble methods are used.

I will largely utilize Decision Trees to outline the definition and practicality of Ensemble Methods (however it is important to note that Ensemble Methods do not only pertain to Decision Trees).



A Decision Tree determines the predictive value based on series of questions and conditions. For instance, this simple Decision Tree determining on whether an individual should play outside or not. The

tree takes several weather factors into account, and given each factor either makes a decision or asks another question. In this example, every time it is overcast, we will play outside. However, if it is raining, we must ask if it is windy or not? If windy, we will not play. But given no wind, tie those shoelaces tight because were going outside to play.



Decision Trees can also solve quantitative problems as well with the same format. In the Tree to the left, we want to know wether or not to invest in a commercial real estate property. Is it an office building? A Warehouse? An Apartment building? Good economic conditions? Poor Economic Conditions? How much will an investment return? These questions are answered and solved using this decision tree.

When making Decision Trees, there are several factors we must take into consideration: On what features do we make our decisions on? What is the threshold for classifying each question into a yes or no answer? In the first Decision Tree, what if we wanted to ask ourselves

if we had friends to play with or not. If we have friends, we will play every time. If not, we might continue to ask ourselves questions about the weather. By adding an additional question, we hope to greater define the Yes and No classes.

This is where Ensemble Methods come in handy! Rather than just relying on one Decision Tree and hoping we made the right decision at each split, Ensemble Methods allow us to take a sample of Decision Trees into account, calculate which features to use or questions to ask at each split, and make a final predictor based on the aggregated results of the sampled Decision Trees.

Types of Ensemble Methods

1. ***BAGG***ing, or ***B***ootstrap ***AGG***regating. **BAGG**ing gets its name because it combines ***B***ootstrapping and ***Agg***regation to form one ensemble model. Given a sample of data, multiple bootstrapped subsamples are pulled. A Decision Tree is formed on each of the bootstrapped subsamples. After each subsample Decision Tree has been formed, an algorithm is used to aggregate over the Decision Trees to form the most efficient predictor. The image below will help explain:

Given a Dataset, bootstrapped subsamples are pulled. A Decision Tree is formed on each bootstrapped sample. The results of each tree are aggregated to yield the strongest, most accurate predictor.

2. **Random Forest** Models. Random Forest Models can be thought of as **BAGG**ing, with a slight tweak. When deciding where to split and how to make decisions, BAGGed Decision Trees have the full disposal of features to choose from. Therefore, although the

bootstrapped samples may be slightly different, the data is largely going to break off at the same features throughout each model. In contrary, Random Forest models decide where to split based on a random selection of features. Rather than splitting at similar features at each node throughout, Random Forest models implement a level of differentiation because each tree will split based on different features. This level of differentiation provides a greater ensemble to aggregate over, ergo producing a more accurate predictor. Refer to the image for a better understanding.

Similar to BAGGing, bootstrapped subsamples are pulled from a larger dataset. A decision tree is formed on each subsample. HOWEVER, the decision tree is split on different features (in this diagram the features are represented by shapes).

In Summary

The goal of any machine learning problem is to find a single model that will best predict our wanted outcome. Rather than making one model and hoping this model is the best/most accurate predictor we can make, ensemble methods take a myriad of models into account, and average those models to produce one final model. It is important to note that Decision Trees are not the only form of ensemble methods, just the most popular and relevant in DataScience today.

7) What is the difference between Bagging and Boosting techniques?

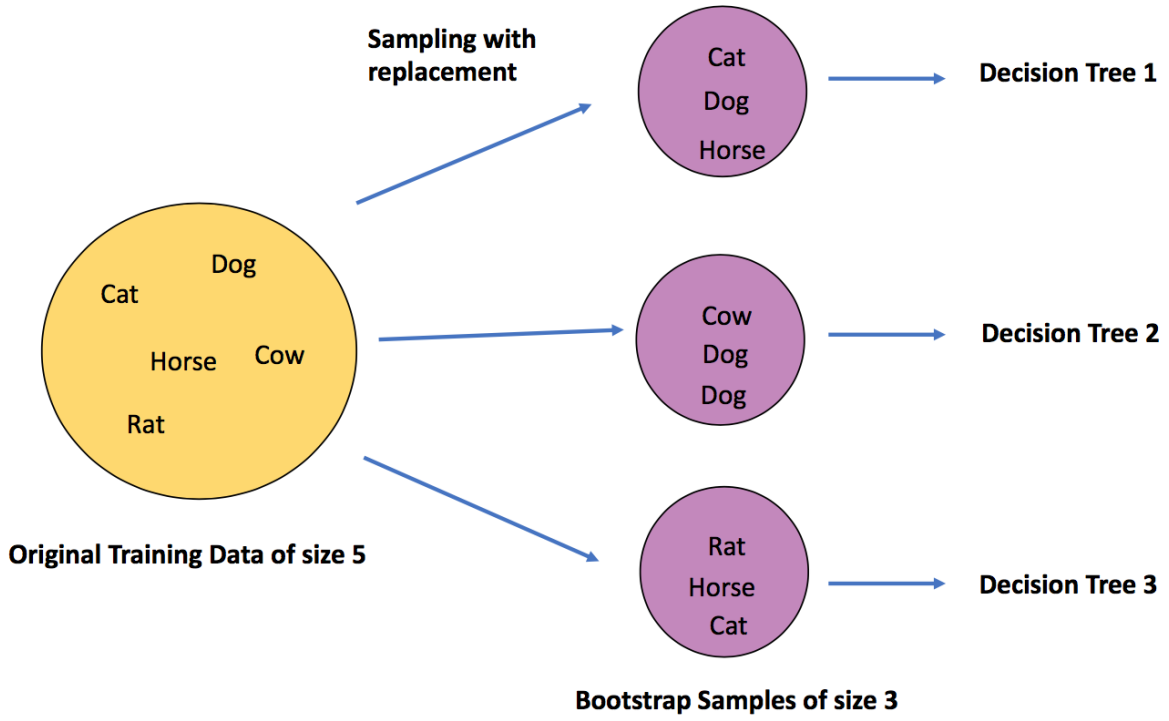
Ans:

S.NO	Bagging	Boosting
1.	The simplest way of combining predictions that belong to the same type.	A way of combining predictions that belong to the different types.
2.	Aim to decrease variance, not bias.	Aim to decrease bias, not variance.
3.	Each model receives equal weight.	Models are weighted according to their performance.
4.	Each model is built independently.	New models are influenced by the performance of previously built models.
5.	Different training data subsets are selected using row sampling with replacement and random sampling methods from the entire training dataset.	Every new subset contains the elements that were misclassified by previous models.
6.	Bagging tries to solve the over-fitting problem.	Boosting tries to reduce bias.
7.	If the classifier is unstable (high variance), then apply bagging.	If the classifier is stable and simple (high bias) the apply boosting.
8.	In this base classifiers are trained parallelly.	In this base classifiers are trained sequentially.
9	Example: The Random forest model uses Bagging.	Example: The AdaBoost uses Boosting techniques

8) What is out-of-bag error in random forests?

Ans: *his blog attempts to explain the internal functioning of oob_score when it is set as true in the "RandomForestClassifier" in "Scikit learn" framework. This blog describes the intuition behind the Out of Bag (OOB) score in Random forest, how it is calculated and where it is useful.*

In the applications that require good interpretability of the model, DTs work very well especially if they are of small depth. However, DTs with real-world datasets can have large depths. Higher depth DTs are more prone to overfitting and thus lead to higher variance in the model. This shortcoming of DT is explored by the Random Forest model. In the Random Forest model, the original training data is **randomly** sampled-with-replacement generating small subsets of data (see the image below). These subsets are also known as bootstrap samples. These bootstrap samples are then fed as training data to many DTs of large depths. Each of these DTs is trained separately on these bootstrap samples. This aggregation of DTs is called the Random Forest ensemble. The concluding result of the ensemble model is determined by counting a majority vote from all the DTs. This concept is known as Bagging or Bootstrap Aggregation. Since each DT takes a different set of training data as input, the deviations in the original training dataset do not impact the final result obtained from the aggregation of DTs. Therefore, bagging as a concept reduces variance without changing the bias of the complete ensemble.



Generation of bootstrap samples with replacement. “Sampling-with-replacement” here means that if a data point is chosen in the first random draw it still remains in the original sample for choosing in another random draw that may follow with an equal probability. This can be seen in the image above as “Dog” is chosen twice in the second bootstrap sample.

What is the Out of Bag score in Random Forests?

Out of bag (OOB) score is a way of validating the Random forest model. Below is a simple intuition of how is it calculated followed by a description of how it is different from validation score and where it is advantageous.

For the description of OOB score calculation, let’s assume there are five DTs in the random forest ensemble labeled from 1 to 5. For simplicity, suppose we have a simple original training data set as below.

Outlook	Temperature	Humidity	Wind	Play Tennis
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Hot	High	Weak	Yes
Windy	Cold	Low	Weak	Yes

Let the first bootstrap sample is made of the first three rows of this data set as shown in the green box below. This bootstrap sample will be used as the training data for the DT “1”.

Outlook	Temperature	Humidity	Wind	Play Tennis
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Hot	High	Weak	Yes
Windy	Cold	Low	Weak	Yes

Bootstrap sample

Then the last row that is “left out” in the original data (see the red box in the image below) is known as Out of Bag sample. This row will not be used as the training data for DT 1. Please note that in reality there will be several such rows which are left out as Out of Bag, here for simplicity only one is shown.

Outlook	Temperature	Humidity	Wind	Play Tennis
Sunny	Hot	High	Weak	No
Sunny	Hot	High	Strong	No
Sunny	Hot	High	Weak	Yes
Windy	Cold	Low	Weak	Yes

Out of Bag sample

After the DTs models have been trained, this leftover row or the OOB sample will be given as unseen data to the DT 1. The DT 1 will predict the outcome of this row. Let DT 1 predicts this row correctly as “YES”. Similarly, this row will be passed through all the DTs that did not contain this row in their bootstrap training data. Let’s assume that apart from DT 1, DT 3 and DT 5 also did not have this row in their bootstrap training data. The predictions of this row by DT 1, 3, 5 are summarized in the table below.

Decision Tree	Prediction
1	YES
3	NO
5	YES
Majority vote : YES	

We see that by a majority vote of 2 “YES” vs 1 “NO” the prediction of this row is “YES”. It is noted that the final prediction of this row by majority vote is a **correct prediction** since originally in the “Play Tennis” column of this row is also a “YES”.

Similarly, each of the OOB sample rows is passed through every DT that did not contain the OOB sample row in its bootstrap training data and a majority prediction is noted for each row.

And lastly, the OOB score is computed as **the number of correctly predicted rows from the out of bag sample.**

What is the difference between OOB score and validation score?

Since we have understood how OOB score is estimated let’s try to comprehend how it differs from the validation score.

As compared to the validation score OOB score is computed on data that was not necessarily used in the analysis of the model. Whereas for calculation validation score, a part of the original training dataset is actually set aside before training the models. Additionally, the OOB score is calculated using only a subset of DTs not containing the OOB sample in their bootstrap training dataset. While the validation score is calculated using all the DTs of the ensemble.

Where can OOB score be useful?

As noted above, only a subset of DTs is used for determining the OOB score. This leads to reducing the overall aggregation effect in bagging. Thus in general, validation on a full ensemble of DTs is better than a subset of DT for estimating the score. However, occasionally the

dataset is not big enough and hence set aside a part of it for validation is unaffordable. Consequently, in cases where we do not have a large dataset and want to consume it all as the training dataset, the OOB score provides a good trade-off. Nonetheless, it should be noted that validation score and OOB score are unalike, computed in a different manner and should not be thus compared.

In an ideal case, about 36.8 % of the total training data forms the OOB sample. This can be shown as follows.

If there are N rows in the training data set. Then, the probability of not picking a row in a random draw is

9) What is K-fold cross-validation?

Ans: Let's say that you have trained a machine learning model. Now, you need to find out how well this model performs. Is it accurate enough to be used? How does it compare to another model? There are several evaluation methods to determine this. One such method is called K-fold cross validation.

Cross validation is an evaluation method used in machine learning to find out how well your machine learning model can predict the outcome of unseen data. It is a method that is easy to comprehend, works well for a limited data sample and also offers an evaluation that is less biased, making it a popular choice.

The data sample is split into 'k' number of smaller samples, hence the name: K-fold Cross Validation. You may also hear terms like four fold cross validation, or ten fold cross validation, which essentially means

that the sample data is being split into four or ten smaller samples respectively.

How is k-fold cross validation performed?

The general stratergy is quite straight forward and the following steps can be used:

2. First, shuffle the dataset and split into k number of subsamples. (It is important to try to make the subsamples equal in size and ensure k is less than or equal to the number of elements in the dataset).
3. In the first iteration, the first subset is used as the test data while all the other subsets are considered as the training data.
4. Train the model with the training data and evaluate it using the test subset. Keep the evaluation score or error rate, and get rid of the model.
5. Now, in the next iteration, select a different subset as the test data set, and make everything else (including the test set we used in the previous iteration) part of the training data.
6. Re-train the model with the training data and test it using the new test data set, keep the evaluation score and discard the model.
7. Continue iterating the above k times. Each data subsamples will be used in each iteration until all data is considered. You will end up with a k number of evaluation scores.
8. The total error rate is the average of all these individual evaluation scores.

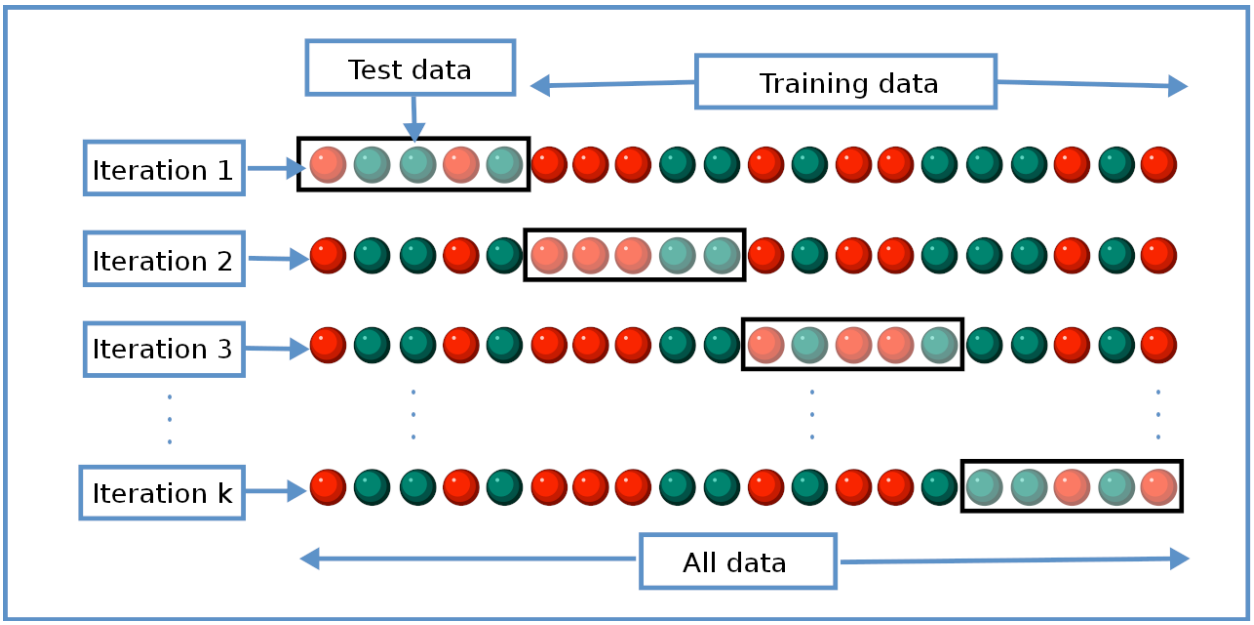


Diagram of k-fold cross validation By Gufosowa — Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=82298768

How to determine the best value for ‘k’ in K-Fold Cross Validation?

Chosing a good value for k is important. A poor value for k can result in a poor evaluation of the model’s abilities. In other words, it can cause the measured ability of the model to be overestimated (high bias) or change widely depending on the training data used (high variance).

Generally, there are three ways to select k:

- Let k = 5, or k =10. Through experimentation, it has been found that selecting k to be 5 or 10 results in sufficiently good results.
- Let k = n, where n is the size of the dataset. This ensures each sample is used in the test data set.
- Another way is to chose k so that every split data sample is sufficiently large, ensuring they are statistically represented in the larger dataset.

Types of cross validation

Cross validation can be divided into two major categories:

- Exhaustive, where the method learn and test on every single possibility of dividing the dataset into training and testing subsets.
- Non-exhaustive cross validation methods where **all** ways of splitting the sample are **not** computed.

Exhaustive cross-validation

Leave-p-out cross validation is a method of exhaustive cross validation. Here, p number of observations (or elements in the sample dataset) are left out as the training dataset, everything else is considered as part of the training data. For more clarity, if you look at the above image, p is equal to 5, as shown by the 5 circles in the ‘test data’.

Leave-one-out cross validation a special form of leave-p-out exhuastive cross validation method, where p = 1. This is also a specific case for k-fold cross validation, where k = N(number of elements in the sample dataset).

Non-exhaustive cross-validation

K-fold cross validation where k is not equal to N, Stratified cross validation and repeated random sub-sampling validation are non-exhaustive cross validation methods.

Stratified cross validation: partitions are selected such that each partition contains roughly the same amount of elements for each class label. For example, in binary classification, every split has elements of

which roughly 50% belongs to class 0 and 50% that belongs to class 1.

Repeated random sub-sampling validation (Monte Carlo cross validation): Data is split into multiple random subsets and the model is trained and evaluated for each split. The results are averaged over the splits. Unlike the k-fold cross validation, proportions of the training and test set size are not dependent on the size of the data set, which is an advantage. However, a disadvantage is that some data elements will never be selected as a part of the test set, while some may be selected multiple times. When the amount of random splits are increased and approach infinity, the results tend to be similar to that of leave-p-out cross validation.

10) What is hyper parameter tuning in machine learning and why it is done?

Ans: A Machine Learning model is defined as a mathematical model with a number of parameters that need to be learned from the data. By training a model with existing data, we are able to fit the model parameters.
However, there is another kind of parameter, known as ***Hyperparameters***, that cannot be directly learned from the regular training process. They are usually fixed before the actual training process begins. These parameters express important properties of the model such as its complexity or how fast it should learn.

Some examples of model hyperparameters include:

The penalty in Logistic Regression Classifier i.e. L1 or L2 regularization

The learning rate for training a neural network.

The C and sigma hyperparameters for support vector machines.

The k in k-nearest neighbors.

The aim of this article is to explore various strategies to tune hyperparameters for Machine learning models.

Models can have many hyperparameters and finding the best combination of parameters can be treated as a search problem. The two best strategies for Hyperparameter tuning are:

11) What issues can occur if we have a large learning rate in Gradient Descent?

Ans: n supervised learning, to enable an algorithm's predictions to be as close to the actual values/labels as possible, we employ two things: 1) A cost function and 2) A technique to minimize the cost function. There are popular forms of cost functions used for different tasks that the algorithms are expected to perform. Also, a popular technique used to minimize the cost function is the gradient descent method. We will understand these concepts to understand the role of 'learning rate' in machine learning.

Cost Function

A cost function is a measure of the error in prediction committed by an algorithm. It indicates the difference between the predicted and the actual values for a given dataset. Closer the predicted value to the actual value, the smaller the difference and lower the value of the cost function. Lower the value of the cost function, the better the predictive capability of the model. An ideal value of the cost function is zero. Some of the popular cost functions used in machine learning for applications such as regression, classification, and density approximation are shown in table-1.

To further the discussion, let's consider the cost function for regression, in which the objective is to learn a mapping function between the *predictors* (independent variables) and *target* (dependent variable). If we assume the relationship to be linear, the equation for the predicted value (yi) is shown below.

$$\hat{y}_i = \beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i} \qquad \textbf{Equation-1}$$

In equation-1, x_1 and x_2 are the two predictors and $\beta_0, \beta_1, \beta_2$ are the model parameters. The algorithm learns(estimates) the values of these parameters during training.

In regression, the typical cost function (***CF***) used is the mean squared error (MSE) cost function. The form of the function is shown below.

$$CF = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \qquad \textbf{Equation-2}$$

In the above equation, yi and yi are respectively the actual and predicted values. **n** is the number of records in the dataset. Replacing yi in the above equation, the cost function can be re-written as shown below

$$CF = \frac{1}{n}\sum_{i=1}^{n}(y_i - (\beta_0 + \beta_1 x_{1i} + \beta_2 x_{2i}))^2 \qquad \textbf{Equation-3}$$

In the above equation, it is important to note that the values for yi**,** x1i and x2i come from the dataset and cannot be manipulated to minimize the cost function. Only the model parameters $\beta_0, \beta_1, \beta_2$ can be manipulated to minimize the cost function. For the given dataset, these parameters can be estimated using the gradient descent method such that the cost function value is minimum.

Gradient Descent Method

Gradient descent is the popular optimization algorithm used in machine learning to estimate the model parameters. During training a model, the value of each parameter is guessed or assigned random values initially. The cost function is calculated based on the initial values and the parameter estimates are improved over several steps such that the cost function assumes a minimum value eventually. This is shown in figure 1 below.

The equation used to improve the estimate is shown below. It is important to note that when one of the parameter's estimates is being improved, the other parameters are held constant. In our example, while the estimate for β_0 is being improved, β_1 and β_2 are held constant.

Learning rate

In machine learning, we deal with two types of parameters; 1) machine learnable parameters and 2) hyper-parameters. The Machine learnable parameters are the one which the algorithms learn/estimate on their own

during the training for a given dataset. In equation-3, β_0, β_1 and β_2 are the machine learnable parameters. The Hyper-parameters are the one which the machine learning engineers or data scientists will assign specific values to, to control the way the algorithms learn and also to tune the performance of the model. Learning rate, generally represented by the symbol 'α', shown in equation-4, is a hyper-parameter used to control the rate at which an algorithm updates the parameter estimates or learns the values of the parameters.

Effect of different values for learning rate

Learning rate is used to scale the magnitude of parameter updates during gradient descent. The choice of the value for learning rate can impact two things: 1) how fast the algorithm learns and 2) whether the cost function is minimized or not. Figure 2 shows the variation in cost function with a number of iterations/epochs for different learning rates.

It can be seen that for an optimal value of the learning rate, the cost function value is minimized in a few iterations (smaller time). This is represented by the blue line in the figure. If the learning rate used is lower than the optimal value, the number of iterations/epochs required to minimize the cost function is high (takes longer time). This is represented by the green line in the figure. If the learning rate is high, the cost function could saturate at a value higher than the minimum value. This is represented by the red line in the figure. If the learning rate selected is very high, the cost function could continue to increase with iterations/epochs. An optimal learning rate is not easy to find for a given problem. Though getting the right learning is always a challenge, there are some well-researched methods documented to figure out optimal learning rates. Some of these techniques are discussed in the following sections. In all these techniques the fundamental idea is to vary the learning rate dynamically instead of using a constant learning rate.

Decaying Learning rate

In the decaying learning rate approach, it decreases with increase in epochs/iterations. The formula used is shown below:

$$\alpha = \frac{\alpha_o}{1+\delta \times Epoch\#}$$ **Equation-5**

In the above equation, o is the initial learning rate, is the decay rate and is the learning rate at a given Epoch number. Figure 3 shows the learning rate decay with the epoch number for different initial learning rates and decay rates.

Scheduled Drop Learning rate

Unlike the decay method, where the learning rate drops monotonously, in the drop method, the learning rate is dropped by a predetermined proportion at a predetermined frequency. The formula used to calculate for a given epoch is shown in the below equation:

$$\alpha_n = \alpha_o \times D^{Quotient\left(\frac{n}{\rho}\right)}$$ **Equation-6**

In the above equation, o is the initial learning rate, '*n*' is the epoch/iteration number, '*D*' is a hyper-parameter which specifies by how much the learning rate has to drop, and ρ is another hyper-parameter which specifies the epoch-based frequency of dropping the learning rate. Figure 4 shows the variation with epochs for different values of *'D'* and *'ρ'*.

The limitation with both the decay approach and the drop approach is that they do not evaluate if decreasing the learning rate is required or not. In both the methods, the learning rate decreases irrespective of difficulty involved in minimizing the cost function.

Adaptive Learning rate

In this approach, the learning rate increases or decreases based on the gradient value of the cost function. For higher gradient value, the learning rate will be smaller and for lower gradient value, the learning rate will be larger. Hence, the learning decelerates and accelerates respectively at steeper and shallower parts of the cost function curve. The formula used in this approach is shown in the below equation.

$$\alpha_n = \frac{\alpha_o}{\sqrt{s_n}} \qquad \textbf{Equation-7}$$

In the above equation, o is the initial learning rate and 's_n' is the momentum factor, which is calculated using below equation. 'n' is the epoch/iteration number

$$S_n = \gamma S_{n-1} + (1-\gamma)\frac{\partial CF}{\partial \beta}]_n \qquad \textbf{Equation-8}$$

In the above equation, γ is a hyperparameter whose value is typically between 0.7 and 0.9. Note that in equation-7, momentum factor S_n is an exponentially weighted average of gradients. So not only the value of the current gradient is considered, but also the values of gradients from the previous epochs are considered to calculate the momentum factor. Figure 5 shows the idea behind the gradient adapted learning rate. When the cost function curve is steep, the gradient is large, and the momentum factor 'Sn' is larger. Hence the learning rate is smaller. When the cost function curve is shallow, the gradient is small and the momentum factor 'Sn' is also small. The learning rate is larger.

The gradient adapted learning rate approach eliminates the limitation in the decay and the drop approaches by considering the gradient of the cost function to increase or decrease the learning rate. This approach is widely used in training deep neural nets with stochastic gradient descent.

Cycling Learning Rate

In this approach, the learning rate varies between a base rate and a maximum rate cyclically. Figure 6 shows the idea behind this approach. The figure shows that the learning rate varies in a triangular form between the maximum and the base rates at a fixed frequency.

It is reported [1] that other forms such as sinusoidal or parabolic too yield similar results. The frequency of variation can be adjusted by setting the value of 'step size'. The formula used in this approach is shown below.

$$\alpha_E = \alpha_{E-1} + (\alpha_{max} - \alpha_{base}) \times (-1)^{quotient(\frac{E}{7}+1)} \rightarrow for\ E > S \qquad \textbf{Equation-9}$$

In the above equation, E is the learning rates for a given epoch, *E* is the epoch number, max and base are respectively the maximum and the base learning rates. *S* is the step size. Note that the above equation valid when *E*>*S*. for E≤S below equation can be used

$$\alpha_E = \alpha_{max} - \frac{(\alpha_{max} - \alpha_{base})}{S} \times (S - E) \rightarrow for\ E\ S \qquad \textbf{Equation-10}$$

An important step in making this approach work is identifying the right base and the maximum learning rates. The process to identify these, referred to as 'LR range test' in [1] is; train the model for a few epochs allowing the learning rate to vary linearly starting from a small number. Capture the accuracy of the model for different learning rates and plot. The plot could look like the image shown in figure 7.

From the plot identify two learning rate values; 1) the value at which the accuracy starts to increase and 2) the value at which the accuracy begins to fluctuate or to decrease. The first point corresponds to the base learning rate, and the second point corresponds to the maximum learning rate. Figure 8 shows the comparison of model accuracy achieved with different learning rate approaches on CIFER-10 image dataset in a convolutional neural network (CNN). In the figure, 'Original learning rate' corresponds to a fixed learning rate, 'Exponential' corresponds to an exponential learning rate decay approach and 'CLR' corresponds to the cyclic learning rate approach. While the constant learning and the exponential approach seem to have taken close to 70000 iterations to achieve an accuracy of 81%, the cyclic learning rate seems to have taken close to 25000 iterations indicating close to 2.5 times quicker convergence.

In this article, the role of learning rate as a hyper-parameter is discussed, highlighting the role it plays in the time taken to train a model and the prediction accuracy achieved by the model. Some of the important aspects to remember are; using variable learning rate instead of a constant learning

rate could help achieve higher accuracy in smaller training time. One of these or a combination of these techniques, especially the adaptive learning rate and the cyclic learning rate can be used in practical applications to train algorithms using the gradient descent method.

12) Can we use Logistic Regression for classification of Non-Linear Data? If not, why?

Ans: *Logistic Regression has traditionally been used as a linear classifier, i.e. when the classes can be separated in the feature space by linear boundaries. That can be remedied however if we happen to have a better idea as to the shape of the decision boundary…*

Logistic regression is known and used as a linear classifier. It is used to come up with a hyper*plane* in feature space to separate observations that belong to a class from all the other observations that do *not* belong to that class. The decision boundary is thus *linear*. Robust and efficient implementations are readily available (e.g. scikit-learn) to use logistic regression as a linear classifier.

While logistic regression makes core assumptions about the observations such as IID (each observation is independent of the others and they all have an identical probability distribution), the use of a linear decision boundary is *not* one of them. The linear decision boundary is used for reasons of simplicity following the Zen mantra – when in doubt simplify. In those cases where we suspect the decision boundary to be nonlinear, it may make sense to formulate logistic regression with a nonlinear model and evaluate how much better we can do. That is what this post is about. Here is the outline. We go through some code snippets here but the full code for reproducing the results can be downloaded from github.

- Briefly review the formulation of the likelihood function and its maximization. To keep the algebra at bay we stick to 2 classes, in a 2-d feature space. A point [x, y] in the feature space can belong to

only one of the classes, and the function f (x,y; c) = 0 defines the decision boundary as shown in Figure 1 below.

 Figure 1. The decision boundary f(x,y; c) = 0 in feature space, separates the observations that belong to class A from those that do not belong to class A. c are the parameters to be estimated.

- Consider a decision boundary that can be expressed as a polynomial in feature variables but linear in the weights/coefficients. This case lends itself to be modeled within the (linear) framework using the API from scikit-learn.
- Consider a generic nonlinear decision boundary that cannot be expressed as a polynomial. Here we are on our own to find the model parameters that maximize the likelihood function. There is excellent API in the scipy.optimize module that helps us out here.

1. Logistic Regression

Logistic regression is an exercise in predicting (regressing to – one can say) discrete outcomes from a continuous and/or categorical set of observations. Each observation is independent and the probability *p* that an observation belongs to the class is some (& same!) function of the features describing that observation. Consider a set of *n* observations [*x_i, y_i; Z_i*] where *x_i, y_i* are the feature values for the *ith* observation. *Z_i* equals 1 if the *ith* observation belongs to the class, and equals 0 otherwise. The likelihood of having obtained *n* such observations is simply the product of the probability *p(x_i, y_i)* of obtaining each one of them separately.

(1) 

The ratio *p/(1-p)* (known as the odds ratio) would be unity along the decision boundary as *p* = *1-p* = 0.5. If we define a function *f(x,y; c)* as:

(2)

then *f(x,y; c) =0* would be the decision boundary. *c* are the *m* parameters in the function *f*. And log-likelihood in terms of *f* would be:

(3)

All that is left to be done now is to find a set of values for the parameters *c* that maximize *log(L)* in Equation 3. We can either apply an optimization technique or solve the coupled set of *m* nonlinear equations *d log(L)/dc = 0* for *c*.

(4)

Either way, having *c* in hand completes the definition of *f* and allows us find out the class of any point in the feature space. That is logistic regression in a nut shell.

2. The function f(x,y; c)

Would any function for *f* work in Equation 2? Most certainly not. As *p* goes from 0 to 1, *log(p/(1-p))* goes from *-inf* to *+inf*. So we need *f* to be unbounded as well in both directions. The possibilities for *f* are endless so long as we make sure that *f* has a range from *-inf* to *+inf*.

2.1 A linear form for f(x,y; c)

Choosing a linear form such as

(5)

will work for sure and that leads to traditional logistic regression as available for use in scikit-learn and the reason logistic regression is known as a *linear* classifier.

2.2 A higher-order polynomial for f(x,y; c)

An easy extension Equation 5 would be to use a higher degree polynomial. A 2nd order one would simply be:

(6) 

Note that the above formulation is identical to the linear case, *if* we treat *x^2, xy*, and *y^2* as three additional independent features of the problem. The impetus for doing so would be that we can then directly apply the API from scikit-learn to get an estimate for c. We see however in the results section that the *c* obtained this way is not as good as directly solving Equation 4 for *c*. It is a bit of a mystery as to why. But in any case solving Equation 4 is easy enough with modules from scipy. The derivatives we need for Equation 4 fall out simply as

2.3 Other generic forms for f(x,y; C)

A periodic form like *f(x,y; c) = sin(c_0x + c_2y) = 0* will not work as its range is limited. But the following will.

We can again evaluate the derivatives and solve for the roots of Equation 4 but sometimes it is simpler to just directly maximize *log(L)* in Equation 3.

13) Differentiate between Adaboost and Gradient Boosting.

Ans: Boosting algorithms are one of the most popular and used algorithms. They can be considered as one of the most powerful techniques for building predictive models.

The basic idea of Boosting just well any other ensemble algorithm is to combine several weak learners into a stronger one. Boosting models tries predictors sequentially, and the subsequent model attempts to fix the errors of its predecessor.

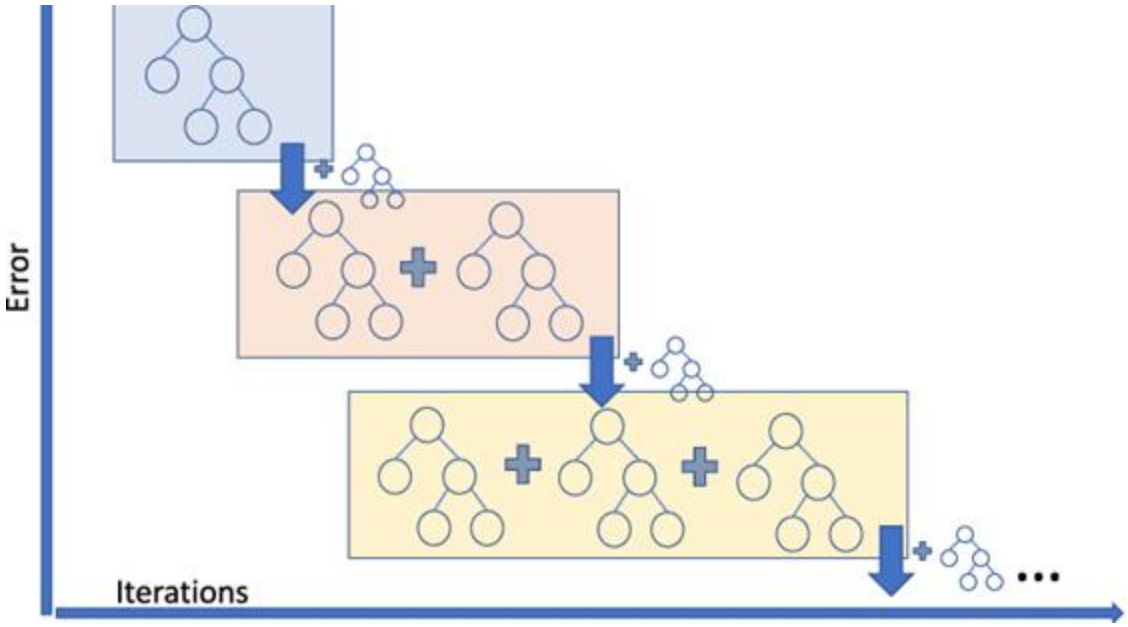
If you don’t know what ensemble models are, Ensemble methods are techniques that create multiple models and then combine them to produce improved results. Ensemble methods usually produces more accurate solutions than a single model would.

We’ll be using the cars dataset throughout the article to understand the concept of Gradient Boosting. We’ll also fit an Ada Boost model to compare both the boosting techniques.

But first let us understand what Gradient Boosting is.

Gradient Boosting

Gradient boosting is a type of boosting algorithm. It relies on the intuition that the best possible next model, when combined with previous models, minimizes the overall prediction error. The key idea is to set the target outcomes for this next model in order to minimize the error. Gradient Boosting can be used for both Classification and Regression.



The difference between Gradient Boost and Ada Boost lies in what it does with the underfitted values of its predecessor. Contrary to AdaBoost, which tweaks the instance weights at every interaction, this method tries to fit the new predictor to the *residual errors* made by the previous predictor.

Basically, Gradient Boosting involves three elements:

1. A loss function to be optimized.

2. A weak learner to make predictions.

3. An additive model to add weak learners to minimize the loss function.

Gradient Boosting is a greedy algorithm and can overfit a training dataset easily. It can benefit from regularization methods that penalize various parts of the algorithm and generally improve the performance of the algorithm by reducing overfitting.

We'll understand more about these elements in later sections. Before understanding how Gradient Boosting is different for Ada Boost, lets first learn what Ada Boost is.

Ada Boost

Adaptive Boosting, or most commonly known AdaBoost, is a Boosting algorithm. The method this algorithm uses to correct its predecessor is by paying more attention to underfitted training instances by the previous model. Hence, at every new predictor the focus will be, each time, on the harder cases.

This sequential learning technique sounds a bit like Gradient Descent, except that instead of tweaking a single predictor's parameter to minimise the cost function, AdaBoost adds predictors to the ensemble gradually making it better. The great disadvantage of this algorithm is that the model cannot be parallelized since each predictor can only be trained after the previous one has been trained and evaluated.

Now, let's understand how the two models, Gradient Boost and Ada Boost are different.

Comparing Gradient Boosting and Ada Boost

Let's say, we want to predict MPG from the cars dataset.

Ada Boost starts by building a short tree called a stump, from the training data. And the amount of say the stump has on the final output is based on how well it is compensated for the previous errors. Then Ada Boost builds a new stump based on the errors that the previous

stump made. Ada Boost continues to make stumps in this fashion until it has made the number of stumps we've asked for.

In Contrast, Gradient Boost starts by making a single leaf instead of a tree or stump. This leaf represents an initial guess for weights of all the samples. When trying to predict a continuous value like MPG, the first guess is the average value and then Gradient Boosting builds a tree. But unlike Ada Boost, this tree is larger than a stump. But Gradient Boost still restricts the size of a tree.

Both Ada Boost and Gradient Boost Scales the trees. Gradient Boost scales the trees by the same amount. Them, Gradient Boost builds another tree based on the errors made by the previous tree and then it scales the tree. Gradient Boost continues to build trees in this fashion, until it has made the number of trees we've asked for, or if the additional trees fail to improve the fit.

Let's understand the working of Gradient Boosting.

Working of Gradient Boosting

Gradient Boosting can be used for regression as well as classification. In this section, we are going to see how Gradient Boosting is used in regression with the help of an example.

We've taken a sample from the cars dataset, where we have to predict MPG (Miles per Gallon) of a car based on different variables

Note: For visualization purpose, we haven't added all the variables of the data.

Car	Origin	Cylinders	Displacement	Horsepower	Weight	Acceleration	Model	MPG
Chevrolet Chevette	US	4	98	70	2120	15.5	80	32.1
Ford Grenada	US	6	250	98	3525	19	77	18.5
Mazda GLC	Japan	4	86	65	2110	17.9	80	46.6
Toyota Corolla	Japan	4	113	95	2278	15.5	72	24
Plymouth Fury	US	6	225	95	3785	19	75	18

The First thing we do is calculate the average. This is the first attempt at predicting every car's MPG. In other words, if we stopped right now, we would predict the MPG of the cars as the average of all the MPG.

Here the average will come as (32.1+18.5+46.6+24+18/5) = 27.84

However, Gradient Boost doesn't stop here. The next thing we do, is build a tree based on errors from the first tree.

The errors that the previous tree made are the difference between observed and predicted MPGs.

Car	Origin	Cylinders	Weight	Acceleration	Model	MPG	*Average*	*Residuals*
Chevrolet Chevette	US	4	2120	15.5	80	32.1	*27.84*	*4.26*
Ford Grenada	US	6	3525	19	77	18.5	*27.84*	*-9.34*
Mazda GLC	Japan	4	2110	17.9	80	46.6	*27.84*	*18.76*
Toyota Corolla	Japan	4	2278	15.5	72	24	*27.84*	*-3.84*
Plymouth Fury	US	6	3785	19	75	18	*27.84*	*-9.84*

Note: For visualization purpose, we haven't added all the variables of the data.

Here, Residuals is the difference between the observed and predicted MPGs, i.e. (Observed MPG — Predicted MPG). These values are called pseudo residuals. The term pseudo residual is based on Linear

Regression where the difference between the observed values and Predicted values results in residuals.

Now we combine the original leaf with the new tree to make a new prediction of an individual from the training data. But this might lead to overfitting.

Gradient Boost deals with this problem by using a learning rate to scale the contributions from the new tree.

So, the predicted value for the first row: 27.84 + 0.1 * 4.26 = 28.266

Car	Cylinders	Weight	Acceleration	Model	MPG	*Average*	*Residuals*	*Predicted*
Chevrolet Chevette	4	2120	15.5	80	32.1	*27.84*	*4.26*	*28.266*
Ford Grenada	6	3525	19	77	18.5	*27.84*	*-9.34*	*26.906*
Mazda GLC	4	2110	17.9	80	46.6	*27.84*	*18.76*	*29.716*
Toyota Corolla	4	2278	15.5	72	24	*27.84*	*-3.84*	*27.456*
Plymouth Fury	6	3785	19	75	18	*27.84*	*-9.84*	*26.856*

Note: For visualization purpose, we haven't added all the variables of the data.

In the above table, we can see all the Predicted values for the training data.

We calculate the residuals again with the new predicted MPGs.

Car	Cylinders	Weight	Acceleration	Model	MPG	*Residuals*	*Predicted*	*New_residual*
Chevrolet Chevette	4	2120	15.5	80	32.1	*4.26*	*28.266*	*3.834*
Ford Grenada	6	3525	19	77	18.5	*-9.34*	*26.906*	*-8.406*
Mazda GLC	4	2110	17.9	80	46.6	*18.76*	*29.716*	*16.884*
Toyota Corolla	4	2278	15.5	72	24	*-3.84*	*27.456*	*-3.456*
Plymouth Fury	6	3785	19	75	18	*-9.84*	*26.856*	*-8.856*

Note: For visualization purpose, we haven’t added all the variables of the data.

The new Residuals are smaller than the older residuals. So, this is a step in the right direction.

Now, we combine the tree with the previous tree and the initial leaf. For example, in the first row:

27.84 + (0.1*4.26) + (0.1*3.83) = 28.649

Car	Cylinders	Weight	Acceleration	Model	MPG	*Residuals*	*Predicted*	*New_residual*	*New_Predicted*
Chevrolet Chevette	4	2120	15.5	80	32.1	*4.26*	*28.266*	*3.834*	*28.6494*
Ford Grenada	6	3525	19	77	18.5	*-9.34*	*26.906*	*-8.406*	*26.0654*
Mazda GLC	4	2110	17.9	80	46.6	*18.76*	*29.716*	*16.884*	*31.4044*
Toyota Corolla	4	2278	15.5	72	24	*-3.84*	*27.456*	*-3.456*	*27.1104*
Plymouth Fury	6	3785	19	75	18	*-9.84*	*26.856*	*-8.856*	*25.9704*

Note: For visualization purpose, we haven’t added all the variables of the data.

We get the new predicted values and then continue the same process.

In summary, we start with a leaf that is the average value if the variable we want to predict, in this case MPG. Then we add a tree based on the Residuals, and we scale the trees contribution to the final prediction with the learning rate. Then we add another tree based on new residuals, the we keep adding trees based on errors made by the previous tree.

The Loss function Of Gradient Boost

The loss function in this case, is something that evaluates how well we can predict MPG. The loss function that is most commonly used in Gradient Boosting is: ½ (Observed — Predicted) ^2

When we remove the ½, we end up with the same loss function as we use for Linear Regression.

The reason why we use this in Gradient Boost is because, when we differentiate it with respect to “Predicted”, and use the chain rule we get:

2/2 * — (observed — predicted)

The 2 and 2 cancels out and we’re left with the negative residual. This makes the math easier since Gradient Boosting uses derivative a lot.

There are other loss functions to choose from, but this is the most popular one for Gradient Boosting Regression.

Gradient Boost Model

To fit the Gradient Boost model on the data, we need to consider a few parameters. These parameters include maximum depth of the tree, number of estimators, the value of the learning rate, minimum samples split, min samples leaf etc.

To get accurate values for these parameters, we can use Hyperparameter tuning. This Hyperparameter tuning will select the best parameters with respect to a metric. Let’s say, in this case the metric is R-squared value. So, the Hyperparameter tuning will select those values for the parameters that will give us a better r squared value.

In the cars dataset, when we're predicting MPG, from Hyperparameter tuning we got the values of number of estimators as 350 and learning rate as 0.1.

From sklearn, we can use GradientBoostingRegressor and fit the Gradient Boosting for regression on the data.

After fitting the Gradient Boosting Model on the data to predict MPG, we got the following results:

R squared value	0.66
MSE value	16.63

As we can see, we got an r squared value of 0.66 and MSE of 17.35.

Now, let's compare these values with the values of Gradient Boosting and see how the models are different from each other.

Comparing the Gradient Boost and Ada Boost Models

	Gradient Boost	Ada Boost
R squared value	0.72	0.66
MSE value	15.73	16.63

We can see that there is not a massive difference between the performance of the model, but still in this case the Gradient Boosting model is a better model.

From the results, we can conclude that for this dataset and parameters the Gradient Boost model outperforms the Ada Boost model. Though this might not be the case all the time.

These results are not clear indicators which model is better, as for different variables and different datasets the models may perform

differently. Considering this dataset, the Gradient Boost does outperform the Ada boost model, but there isn't much of a difference. The Gradient certainly performs better than the Ada Boost model in this case, but the margin is not huge.

Considering no other improvisation is done on the data, both the model performed exceptionally. That is why Boosting is such an effective and popular algorithm nowadays. We even seethe use of these boosting Algorithms in so many Kaggle competitions. Boosting Algorithms are very popular, and for the right reasons too. Though Gradient Boosting and Ada boost are different, they are similar in many ways as well. In the end, they follow the same methodology of ensemble models and hence are exceptional in terms of performance and output.

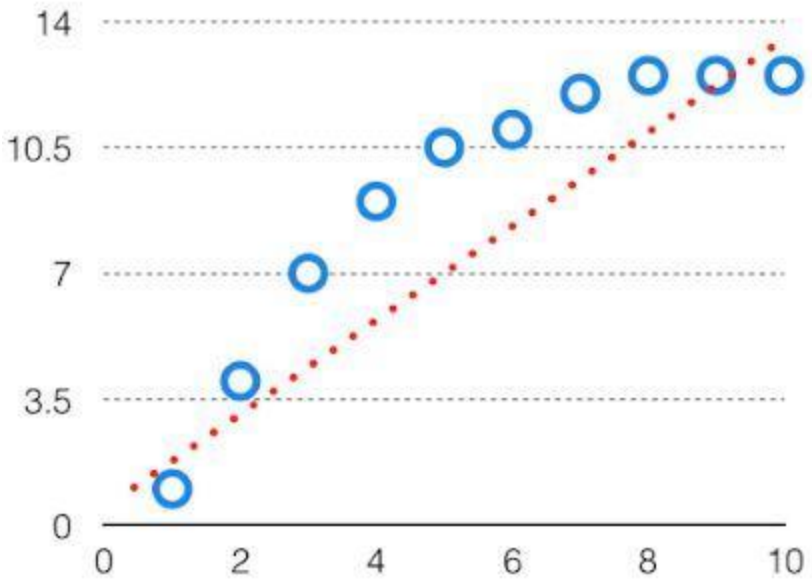
14) What is bias-variance trade off in machine learning?

Ans: It is important to understand prediction errors (bias and variance) when it comes to accuracy in any machine learning algorithm. There is a tradeoff between a model's ability to minimize bias and variance which is referred to as the best solution for selecting a value of **Regularization** constant. Proper understanding of these errors would help to avoid the overfitting and underfitting of a data set while training the algorithm.

Bias

The bias is known as the difference between the prediction of the values by the ML model and the correct value. Being high in biasing gives a large error in training as well as testing data. Its recommended that an algorithm should always be low biased to avoid the problem of underfitting.

By high bias, the data predicted is in a straight line format, thus not fitting accurately in the data in the data set. Such fitting is known as Underfitting of Data. This happens when the hypothesis is too simple or linear in nature. Refer to the graph given below for an example of such a situation.



HighBias

In such a problem, a hypothesis looks like follows.

$$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$$

Variance

The variability of model prediction for a given data point which tells us spread of our data is called the variance of the model. The model with high variance has a very complex fit to the training data and thus is not able to fit accurately on the data which it hasn't seen before. As a result, such models perform very well on training data but has high error rates on test data.

When a model is high on variance, it is then said to as **Overfitting of Data**.

Overfitting is fitting the training set accurately via complex curve and high order hypothesis but is not the solution as the error with unseen data is high.

While training a data model variance should be kept low.

In such a problem, a hypothesis looks like follows.

$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Bias Variance Tradeoff

If the algorithm is too simple (hypothesis with linear eq.) then it may be on high bias and low variance condition and thus is error-prone. If algorithms fit too complex (hypothesis with high degree eq.) then it may be on high variance and low bias. In the latter condition, the new entries will not perform well. Well, there is something between both of these conditions, known as Trade-off or Bias Variance Trade-off.

This tradeoff in complexity is why there is a tradeoff between bias and variance. An algorithm can't be more complex and less complex at the same time. For the graph, the perfect tradeoff will be like.

15) Give short description each of Linear, RBF, Polynomial kernels used in SVM.

Ans: The **Support Vector Machine** is a **supervised learning algorithm** mostly used for **classification** but it can be used also for **regression**. The main idea is that based on the labeled data (training data) the algorithm tries to find the **optimal hyperplane** which can be used to classify new data points. In two dimensions the hyperplane is a simple line.

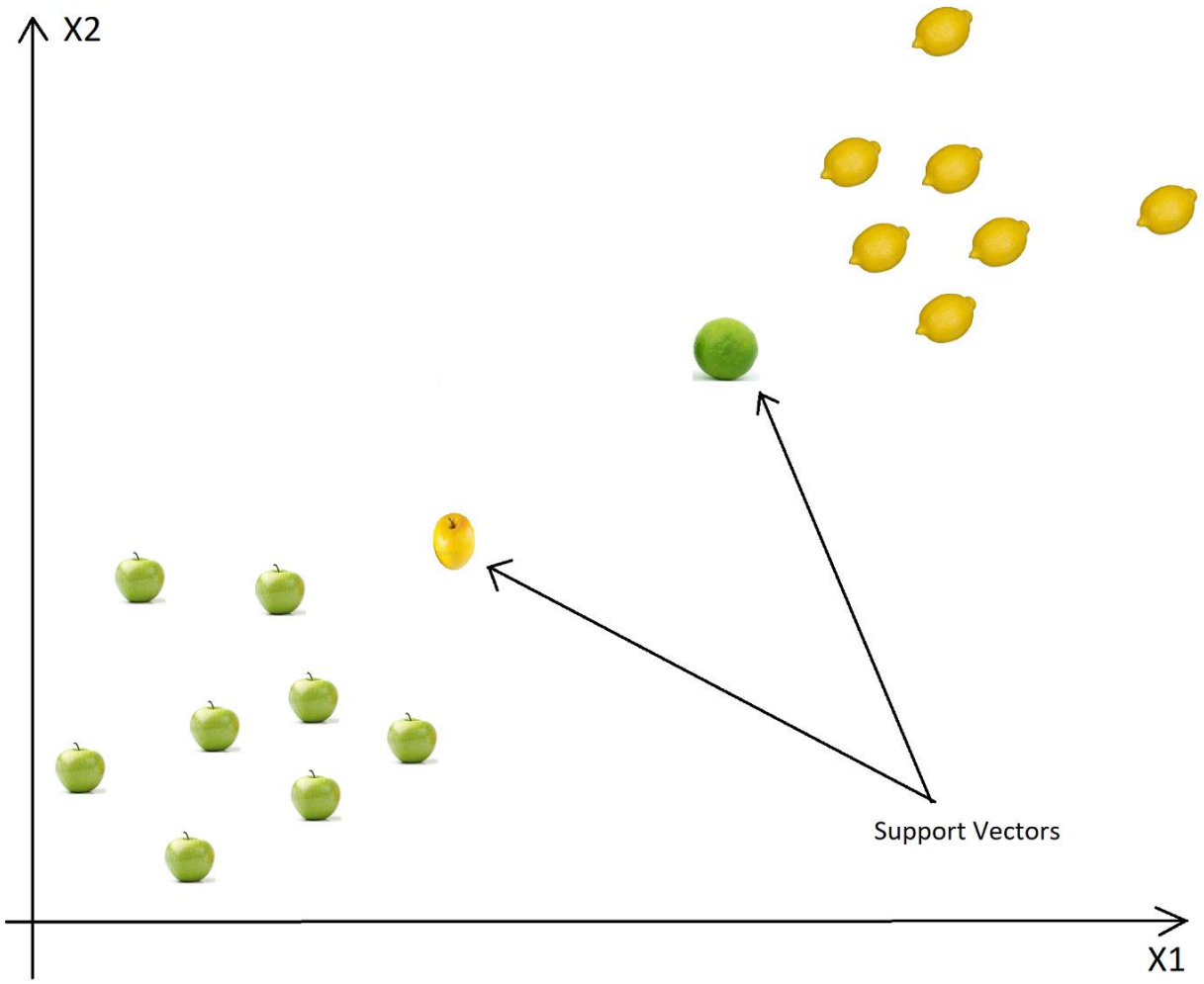
Usually a learning algorithm tries to learn the **most common characteristics (what differentiates one class from another)** of a class and the classification is based on those representative characteristics learnt (so classification is based on differences between classes). The **SVM** works in the other way around. It **finds** the **most similar examples** between classes. Those will be the **support vectors**.

As an example, lets consider two classes, apples and lemons.

Other algorithms will learn the most evident, most representative characteristics of apples and lemons, like apples are green and rounded while lemons are yellow and have elliptic form.

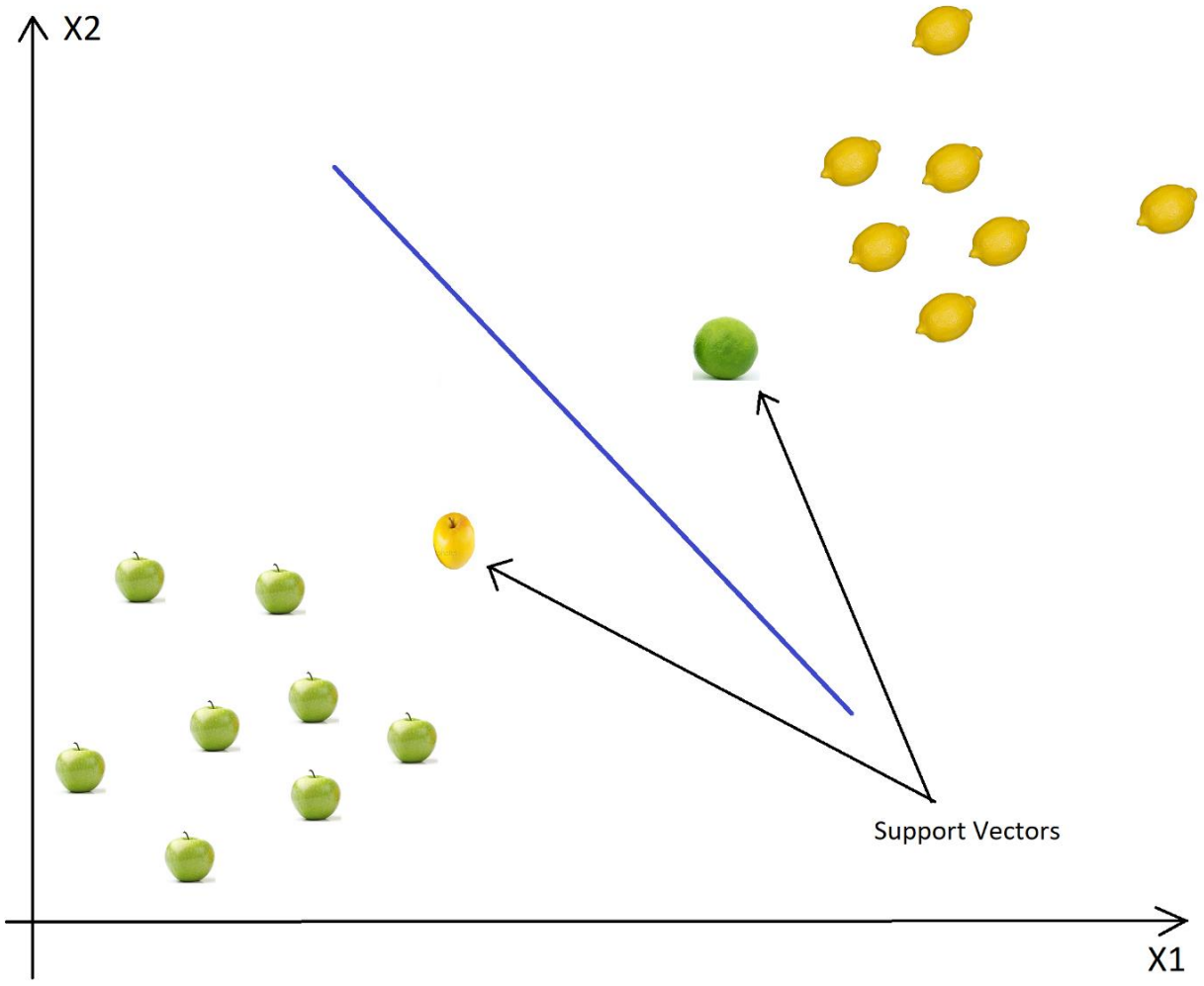
In contrast, SVM will search for apples that are very similar to lemons, for example apples which are yellow and have elliptic form. This will be a support vector. The other support vector will be a lemon similar to an apple (green and rounded). So **other algorithms** learns the **differences** while **SVM** learns **similarities**.

If we visualize the example above in 2D, we will have something like this:

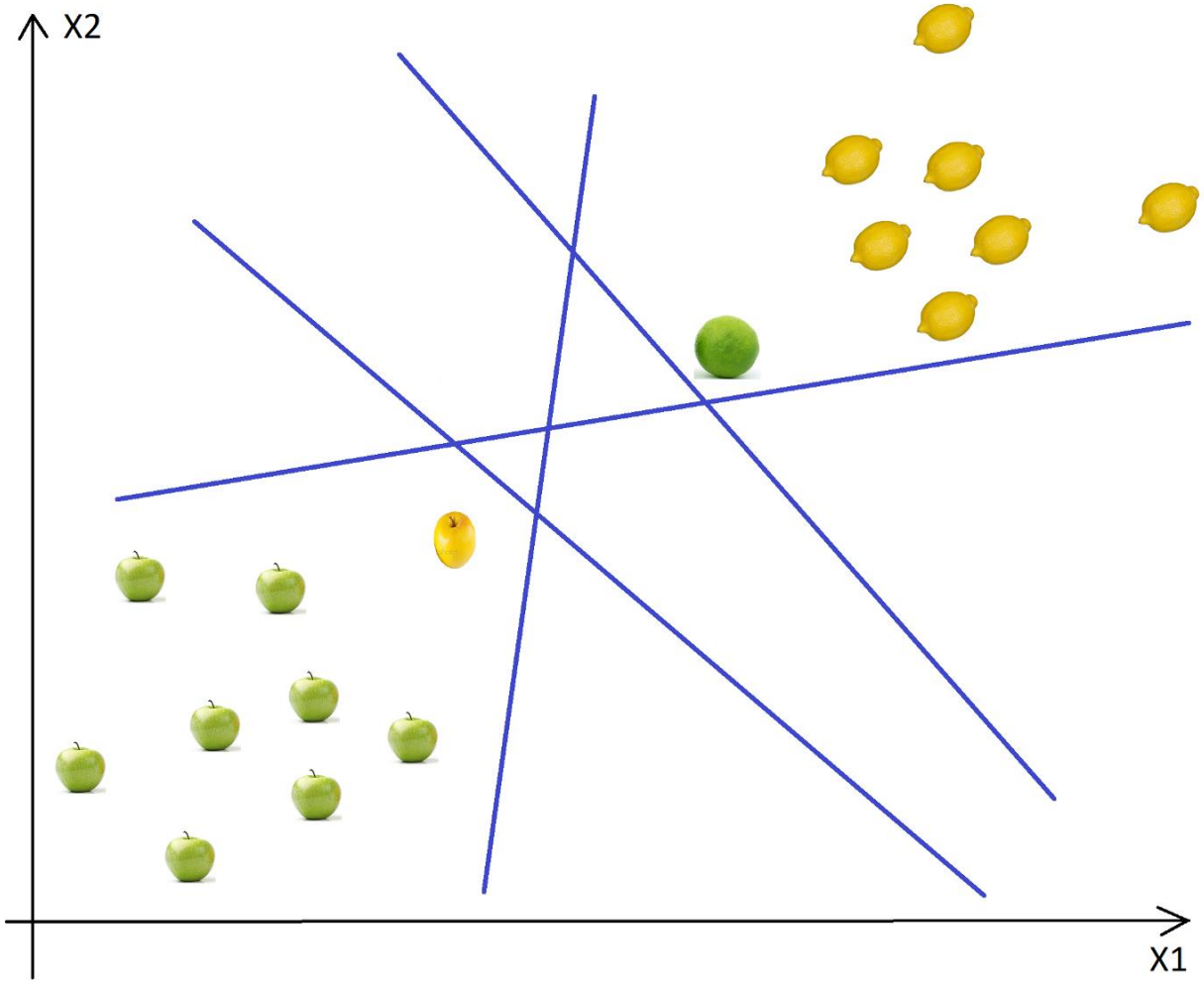


As we go from left to right, all the examples will be classified as apples until we reach the yellow apple. From this point, the confidence that a new example is an apple drops while the lemon class confidence increases. When the lemon class confidence becomes greater than the apple class confidence, the new examples will be classified as lemons (somewhere between the yellow apple and the green lemon).

Based on these support vectors, the algorithm tries to find **the best hyperplane that separates the classes**. In 2D the hyperplane is a line, so it would look like this:



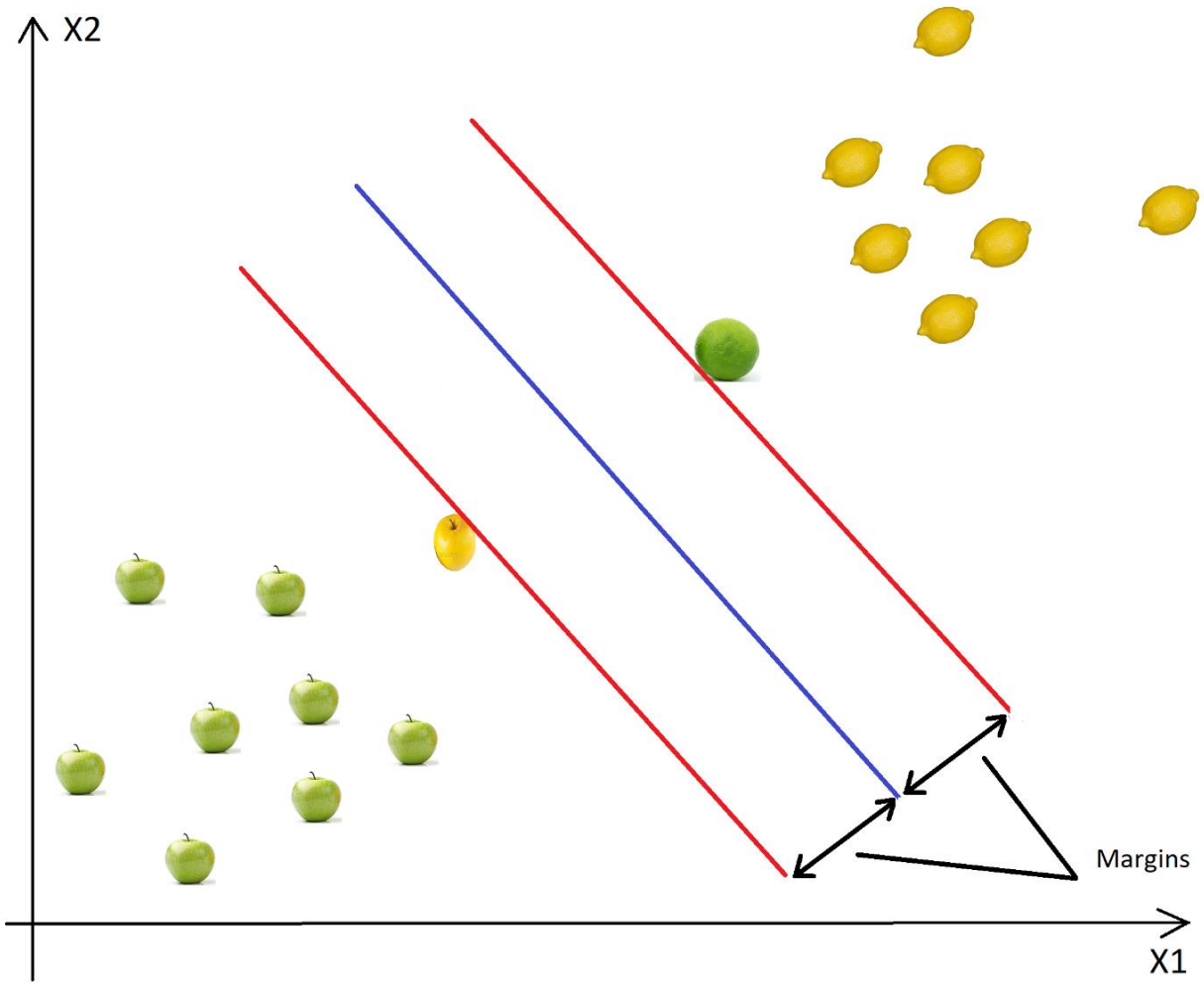
Ok, but **why did I draw the blue boundary like in the picture above?** I could also draw boundaries like this:



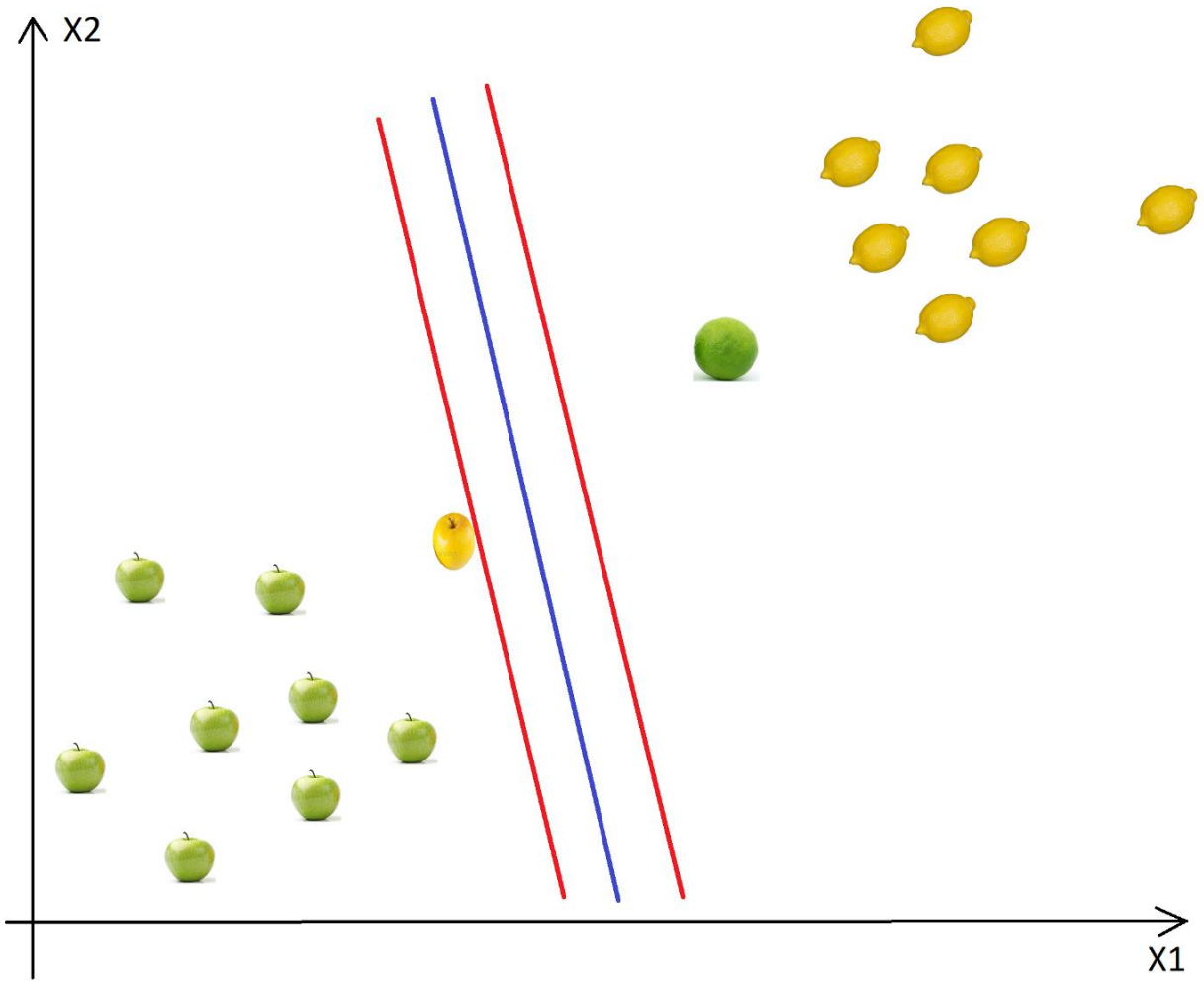
As you can see, we have **an infinite number of possibilities to draw the decision boundary**. So how can we find the optimal one?

Finding the Optimal Hyperplane

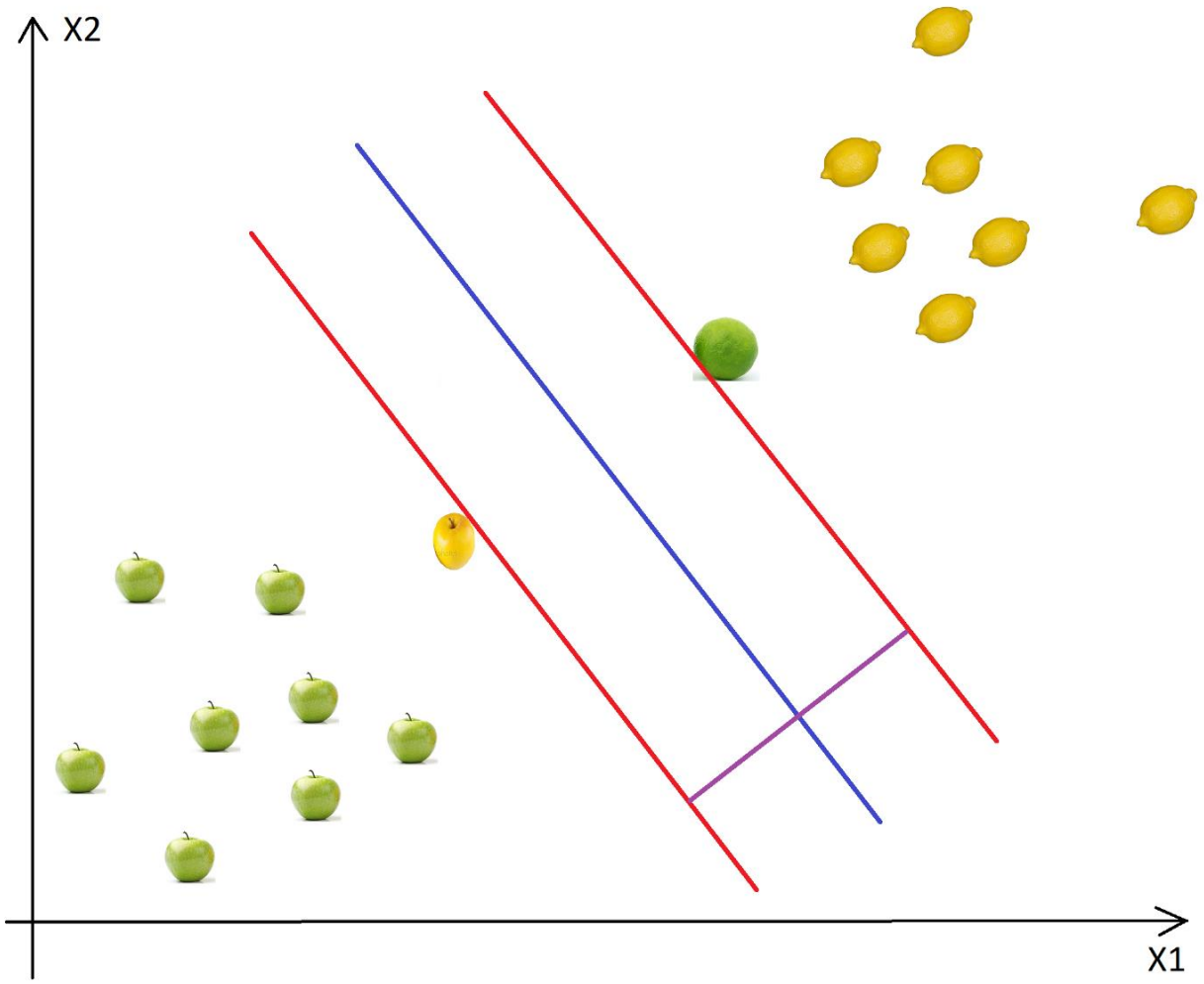
Intuitively the **best line** is the line that is **far away from both apple and lemon examples** (has the largest margin). To have optimal solution, we have to **maximize the margin in both ways** (if we have multiple classes, then we have to maximize it considering each of the classes).



So if we compare the picture above with the picture below, we can easily observe, that the first is the optimal hyperplane (line) and the second is a sub-optimal solution, because the margin is far shorter.



Because we want to maximize the margins taking in consideration **all the classes**, instead of using one margin for each class, **we use a "global" margin**, **which takes in consideration all the classes**. This margin would look like the purple line in the following picture:



This margin is **orthogonal** to the boundary and **equidistant** to the support vectors.

So where do we have vectors? Each of the calculations (calculate distance and optimal hyperplanes) are made in **vectorial space**, so each data point is considered a vector. The **dimension** of the space **is defined by the number of attributes** of the examples. To understand the math behind, please read this brief mathematical description of vectors, hyperplanes and optimizations: SVM Succintly.

All in all, **support vectors** are data points that **defines the position and the margin of the hyperplane**. We call them **"support" vectors**, because these

are the representative data points of the classes, **if we move one of them, the position and/or the margin will change**. Moving other data points won't have effect over the margin or the position of the hyperplane.

To make classifications, we don't need all the training data points (like in the case of KNN), we have to save only the support vectors. In worst case all the points will be support vectors, but this is very rare and if it happens, then you should check your model for errors or bugs.

So basically the **learning is equivalent with finding the hyperplane with the best margin**, so it is a simple **optimization problem**.

Basic Steps

The basic steps of the SVM are:

9. select **two hyperplanes** (in 2D) which separates the data **with no points between them** (red lines)
10. **maximize their distance** (the margin)
11. the **average line** (here the line half way between the two red lines) will be the **decision boundary**

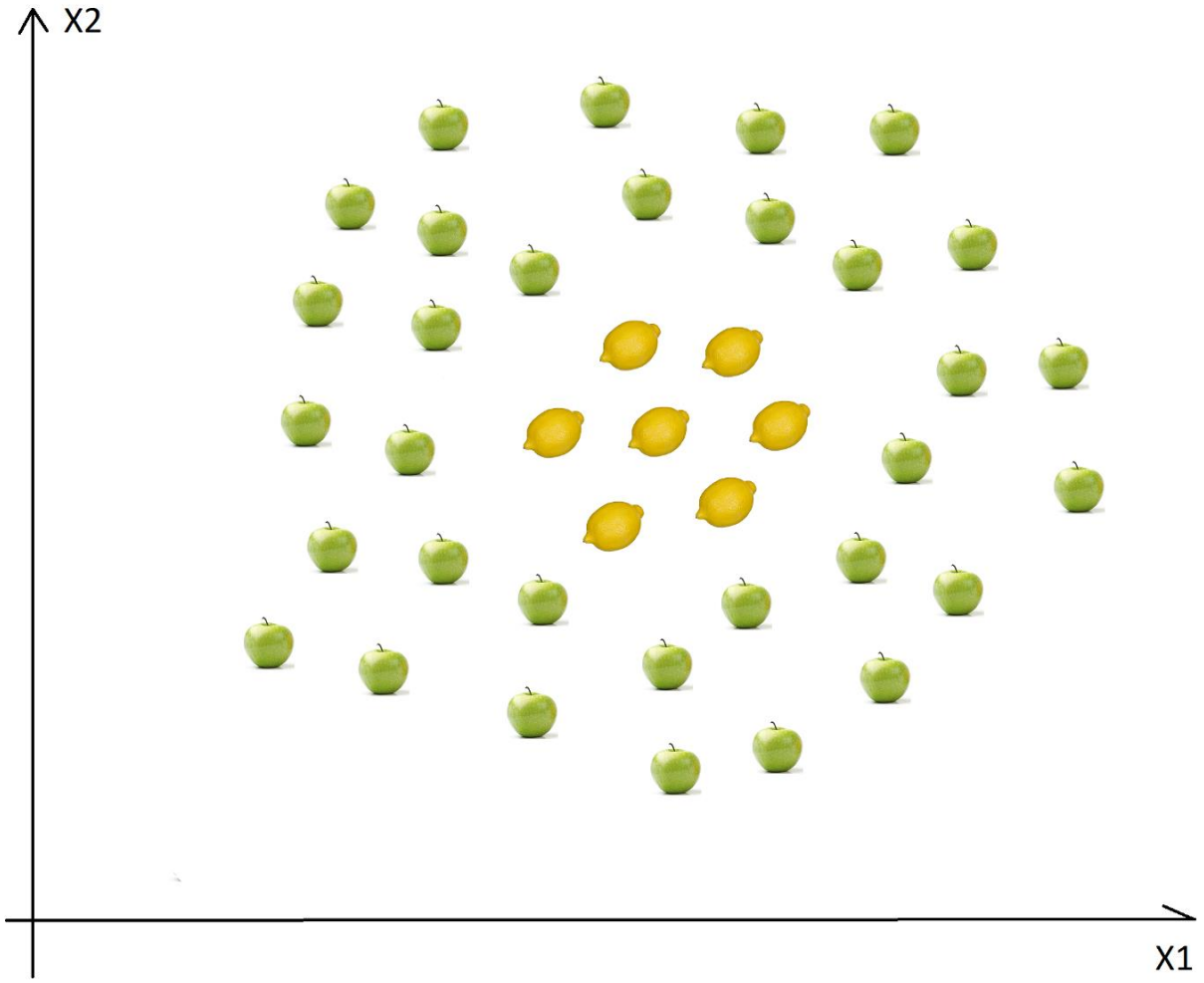
This is very nice and easy, but finding the best margin, the optimization problem is not trivial (it is easy in 2D, when we have only two attributes, but what if we have N dimensions with N a very big number)

To solve the optimization problem, we use the **Lagrange Multipliers**. To understand this technique you can read the following two articles: Duality Langrange Multiplier and A Simple Explanation of Why Langrange Multipliers Wroks.

Until now we had linearly separable data, so we could use a line as class boundary. But what if we have to deal with non-linear data sets?

SVM for Non-Linear Data Sets

An example of non-linear data is:



In this case **we cannot find a straight line** to separate apples from lemons. So how can we solve this problem. We will use the **Kernel Trick!**

The basic idea is that when a data set is inseparable in the current dimensions, **add another dimension**, maybe that way the data will be separable. Just think about it, the example above is in 2D and it is inseparable, but maybe in 3D there is a gap between the apples and the lemons, maybe there is a level difference, so lemons are on level one and

apples are on level two. In this case, we can easily draw a separating hyperplane (in 3D a hyperplane is a plane) between level 1 and 2.

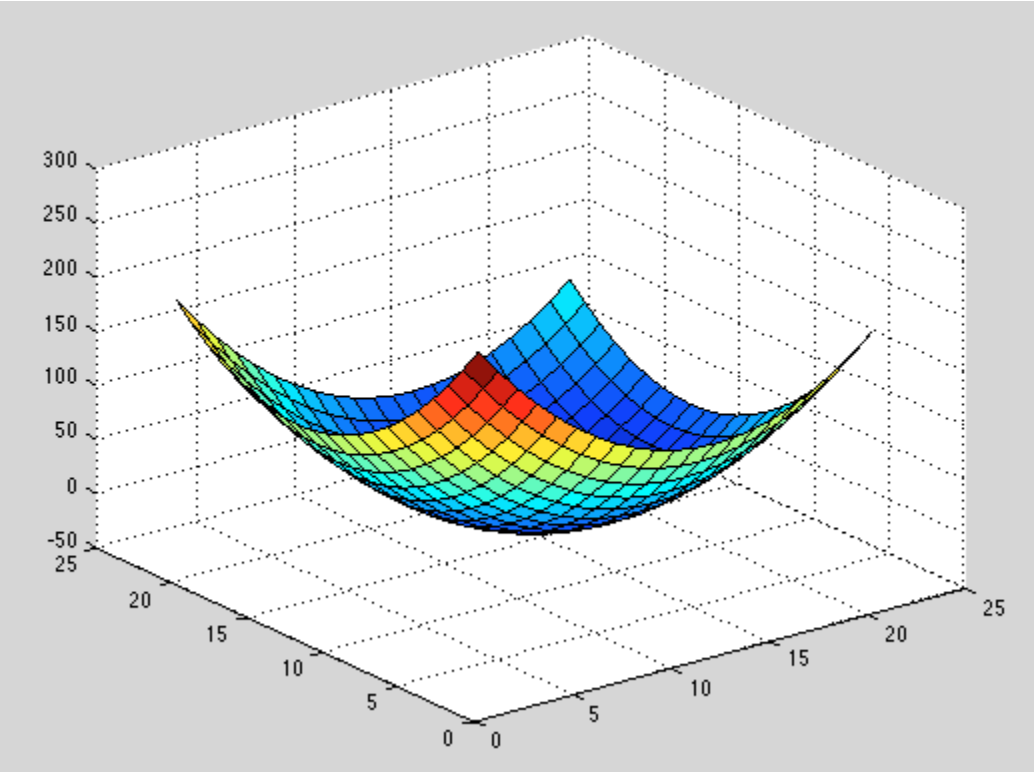
Mapping to Higher Dimensions

To solve this problem we **shouldn't just blindly add another dimension**, we should transform the space so we generate this level difference intentionally.

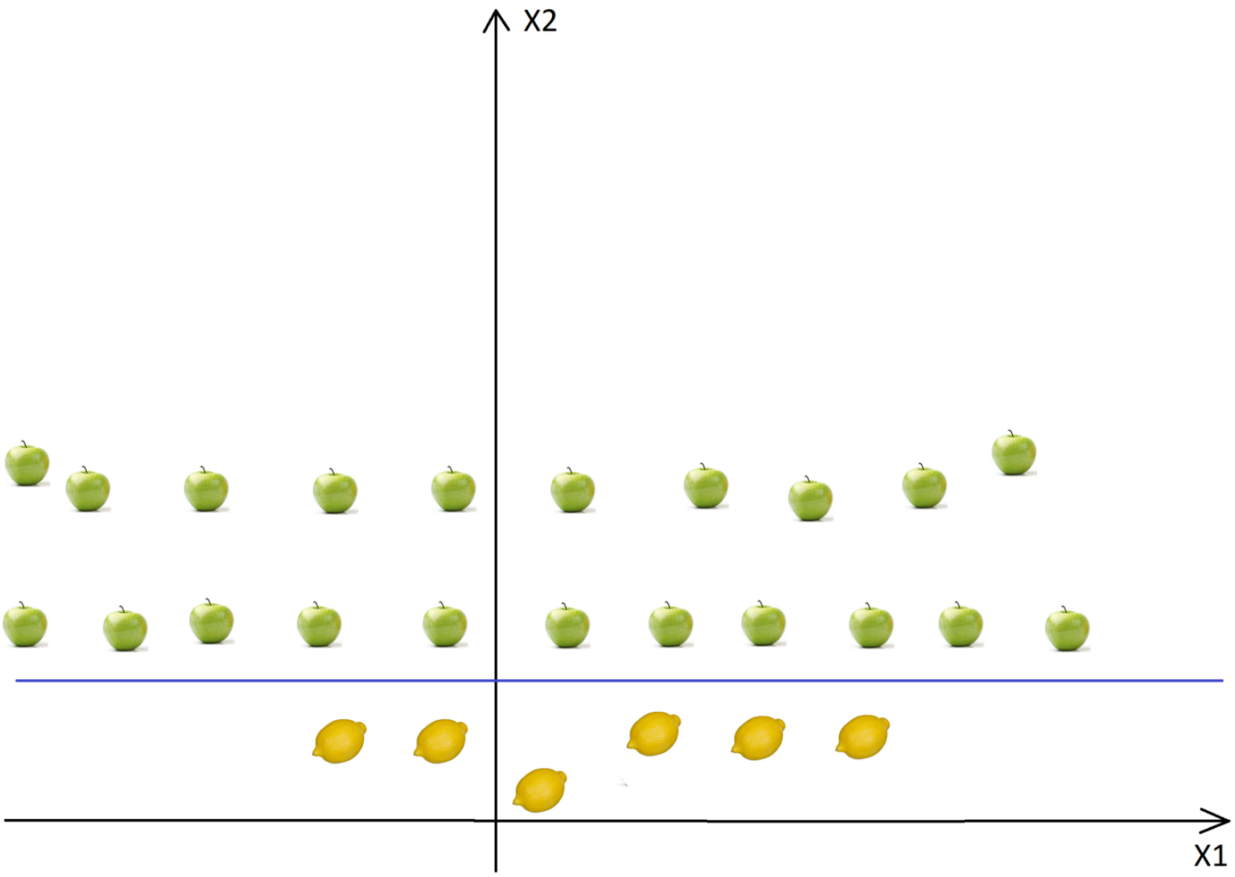
Mapping from 2D to 3D

Let's assume that we add another dimension called **X3**. Another important transformation is that in the new dimension the points are organized using this formula $\mathbf{x1^2 + x2^2}$.

If we plot the plane defined by the $\mathbf{x^2 + y^2}$ formula, we will get something like this:



Now we have to map the apples and lemons (which are just simple points) to this new space. Think about it carefully, what did we do? We just used a transformation in which **we added levels based on distance**. If you are in the origin, then the points will be on the lowest level. As we move away from the origin, it means that we are **climbing the hill** (moving from the center of the plane towards the margins) so the level of the points will be higher. Now if we consider that the origin is the lemon from the center, we will have something like this:

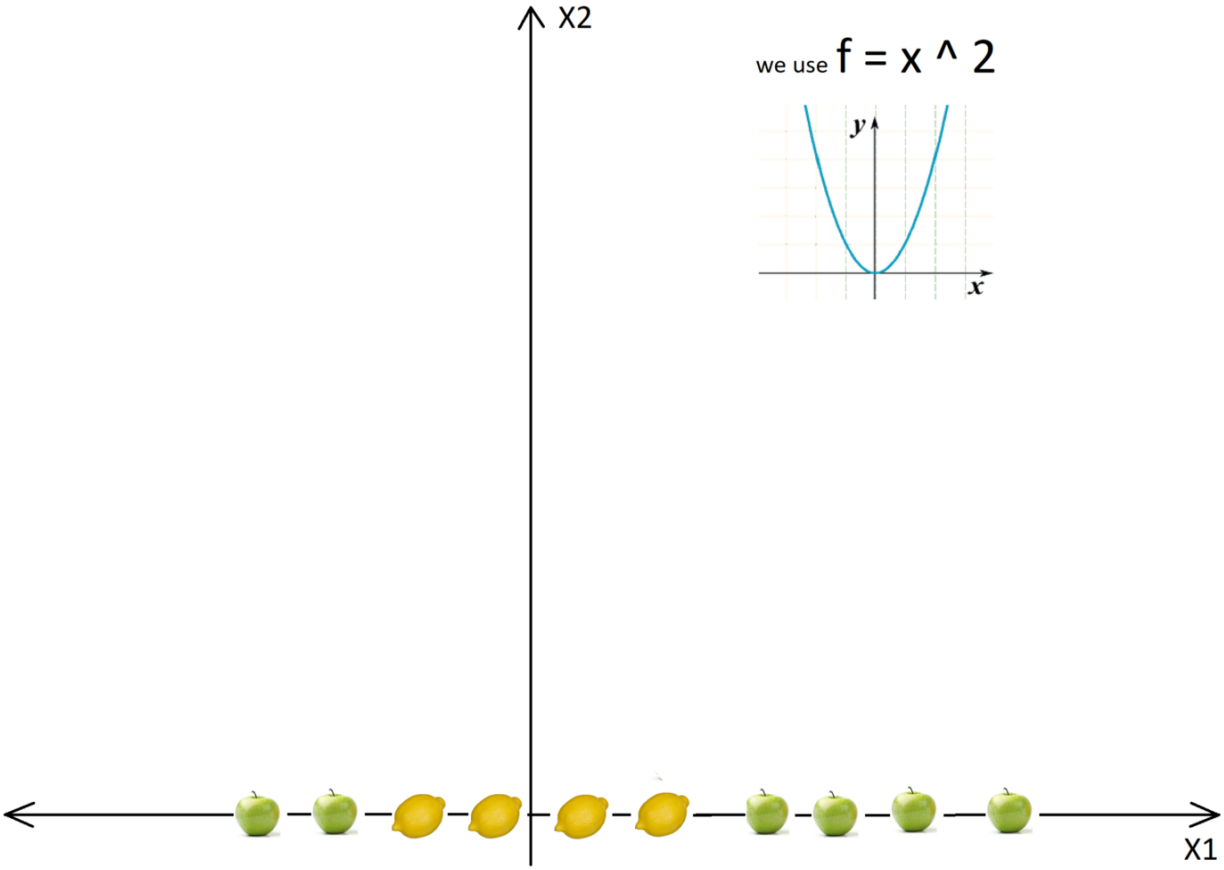


Now we can easily separate the two classes. These transformations are called **kernels**. Popular kernels are: **Polynomial Kernel, Gaussian Kernel, Radial Basis Function (RBF), Laplace RBF Kernel, Sigmoid Kernel, Anove RBF**

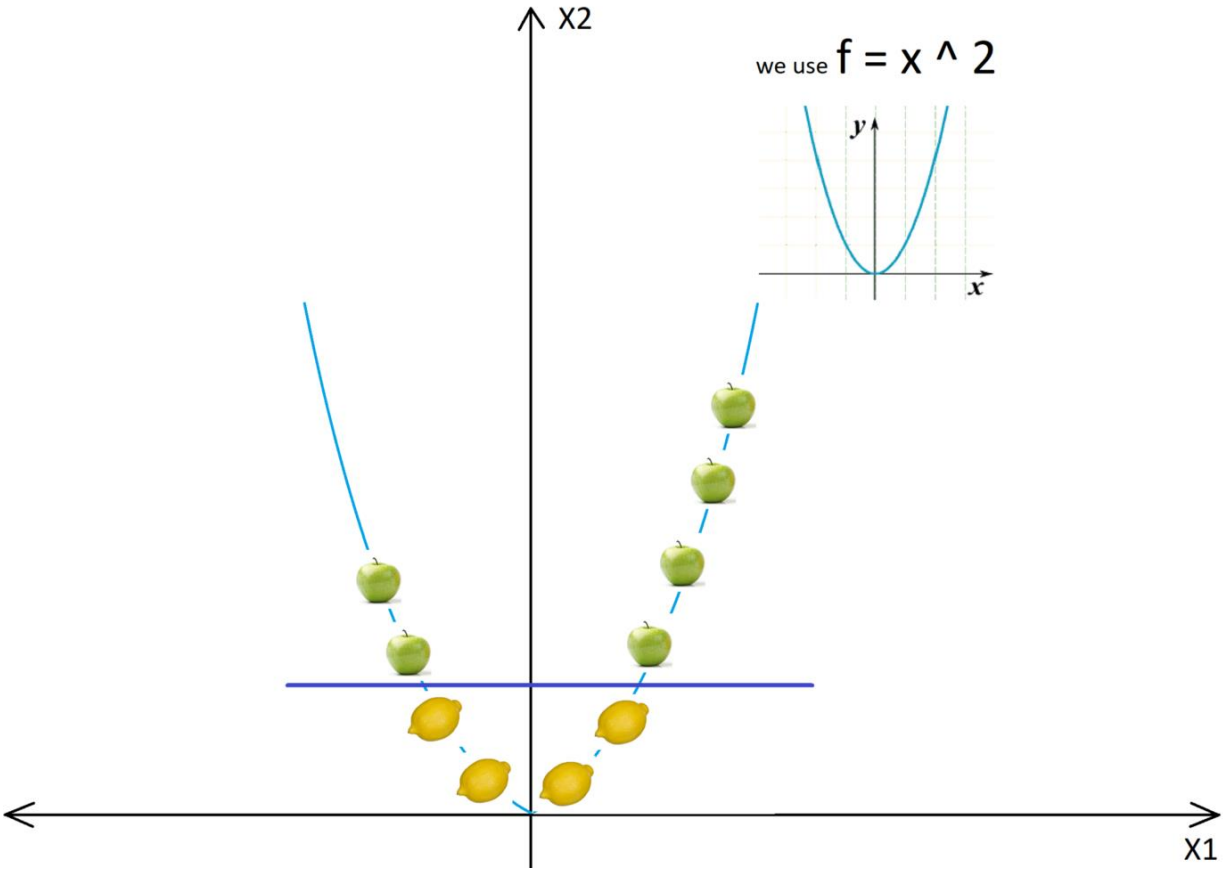
Kernel, etc (see Kernel Functions or a more detailed description Machine Learning Kernels).

Mapping from 1D to 2D

Another, easier example in 2D would be:



After using the kernel and after all the transformations we will get:



So after the transformation, we can easily delimit the two classes using just a single line.

In real life applications we won't have a simple straight line, but we will have lots of curves and high dimensions. In some cases we won't have two hyperplanes which separates the data with no points between them, so **we need some trade-offs, tolerance for outliers**. Fortunately the SVM algorithm has a so-called **regularization parameter** to configure the trade-off and to tolerate outliers.

Tuning Parameters

As we saw in the previous section **choosing the right kernel is crucial**, because if the transformation is incorrect, then the model can have very

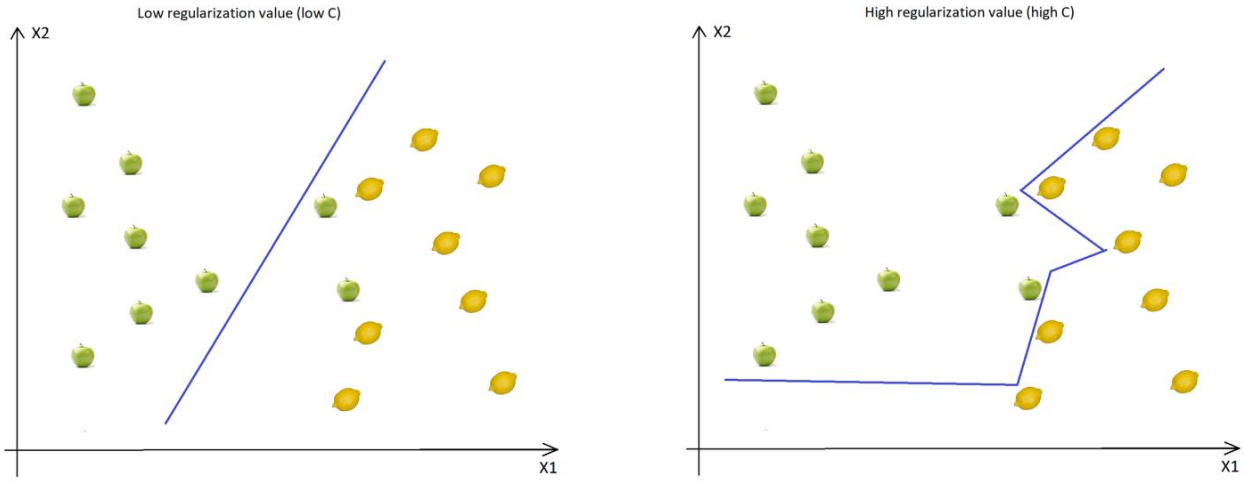
poor results. As a rule of thumb, **always check if you have linear data** and in that case always **use linear SVM** (linear kernel). **Linear SVM is a parametric model**, but an **RBF kernel SVM isn't**, so the complexity of the latter grows with the size of the training set. Not only is **more expensive to train an RBF kernel SVM**, but you also have to **keep the kernel matrix around**, and the **projection into** this "infinite" **higher dimensional space** where the data becomes linearly separable is **more expensive** as well during prediction. Furthermore, you have **more hyperparameters to tune**, so model selection is more expensive as well! And finally, it's much **easier to overfit** a complex model!

Regularization

The **Regularization Parameter** (**in python it's called C**) tells the SVM optimization **how much you want to avoid miss classifying** each training example.

If the **C is higher**, the optimization will choose **smaller margin** hyperplane, so training data **miss classification rate will be lower**.

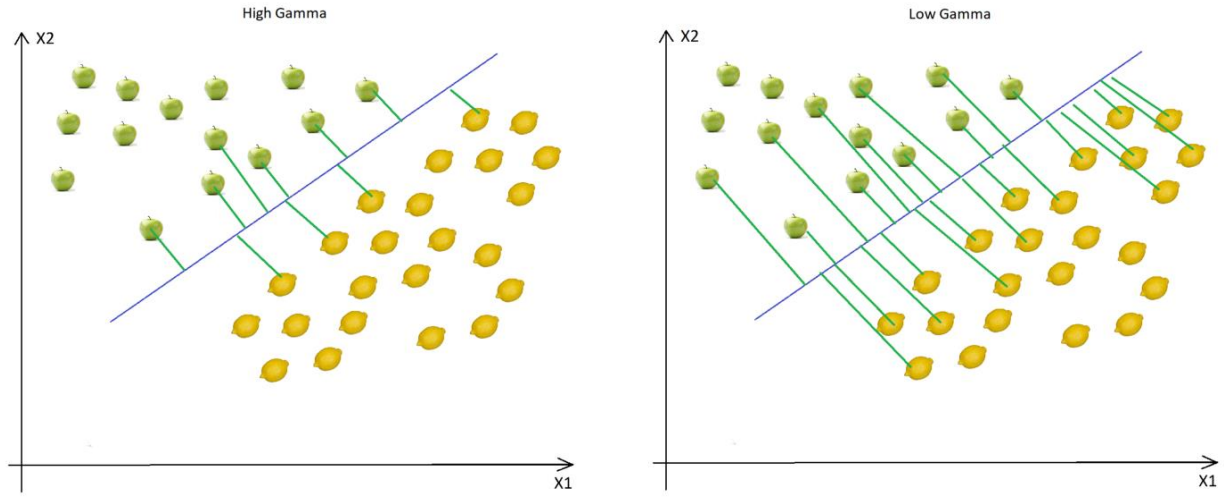
On the other hand, if the **C is low**, then the **margin will be big**, even if there **will be miss classified** training data examples. This is shown in the following two diagrams:



As you can see in the image, when the C is low, the margin is higher (so implicitly we don't have so many curves, the line doesn't strictly follows the data points) even if two apples were classified as lemons. When the C is high, the boundary is full of curves and all the training data was classified correctly. **Don't forget**, even if all the training data was correctly classified, this doesn't mean that increasing the C will always increase the precision (because of overfitting).

Gamma

The next important parameter is **Gamma**. The gamma parameter defines **how far the influence of a single training example reaches**. This means that **high Gamma** will consider only points **close** to the plausible hyperplane and **low Gamma** will consider **points at greater distance**.



As you can see, decreasing the Gamma will result that finding the correct hyperplane will consider points at greater distances so more and more points will be used (green lines indicates which points were considered when finding the optimal hyperplane).

Margin

The last parameter is the **margin**. We've already talked about margin, **higher margin results better model**, so better classification (or prediction). The margin should be always **maximized**.

SVM Example using Python

In this example we will use the Social_Networks_Ads.csv file, the same file as we used in the previous article, see KNN example using Python.

In this example I will write down only the **differences** between SVM and KNN, because I don't want to repeat myself in each article! If you want the **whole explanation** about how can we read the data set, how do we parse and split our data or how can we evaluate or plot the decision boundaries, then please **read the code example from the previous chapter** (KNN)!

Because the **sklearn** library is a very well written and useful Python library, we don't have too much code to change. The only difference is that we have to import the **SVC** class (SVC = SVM in sklearn) from **sklearn.svm** instead of the KNeighborsClassifier class from sklearn.neighbors.

```
# Fitting SVM to the Training set
from sklearn.svm import SVC
classifier = SVC(kernel = 'rbf', C = 0.1, gamma = 0.1)
classifier.fit(X_train, y_train)
```

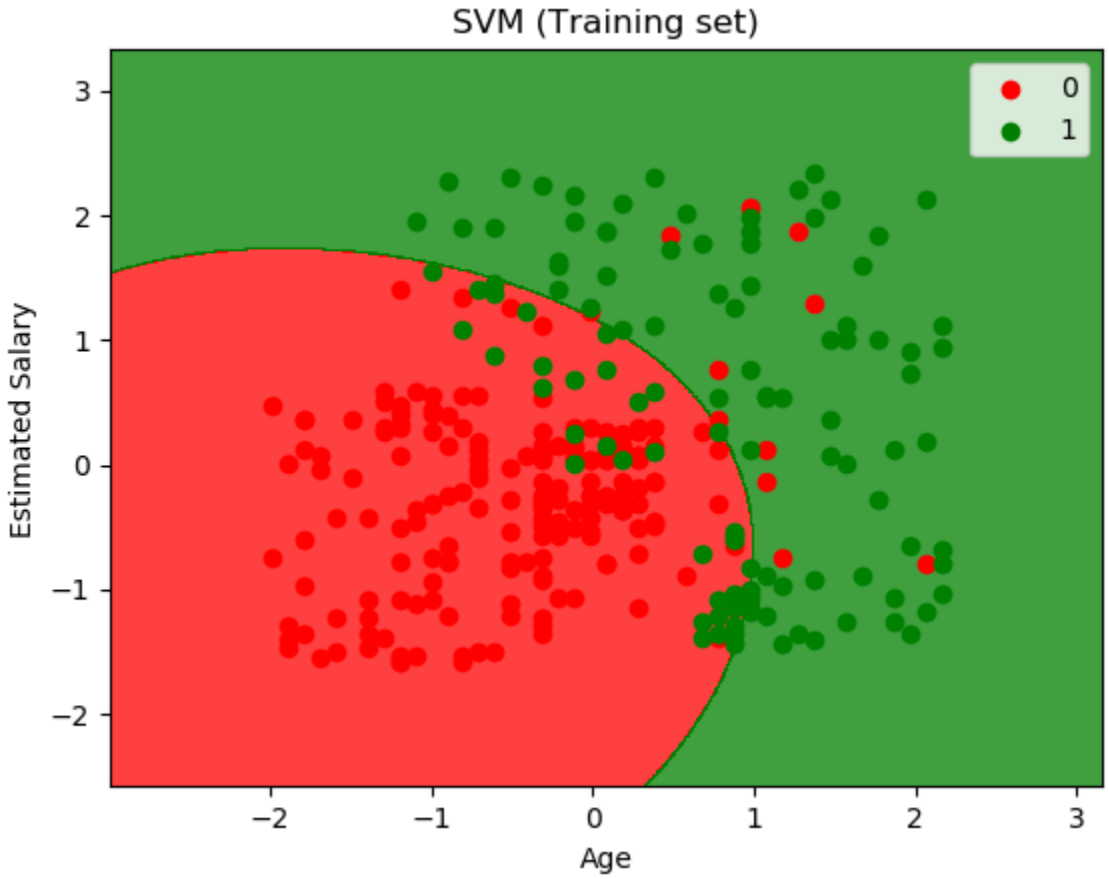
After importing the SVC, we can create our new model using the predefined constructor. This constructor has many parameters, but I will describe only the most important ones, most of the time you won't use other parameters.

The most important parameters are:

12. **kernel:** the kernel type to be used. The most common kernels are **rbf** (this is the default value), **poly** or **sigmoid**, but you can also create your own kernel.
13. **C:** this is the **regularization parameter** described in the Tuning Parameters section
14. **gamma:** this was also described in the Tuning Parameters section
15. **degree:** it is used **only if the chosen kernel is poly** and sets the degree of the polinom
16. **probability:** this is a boolean parameter and if it's true, then the model will return for each prediction, the vector of probabilities of belonging to each class of the response variable. So basically it will give you the **confidences for each prediction**.
17. **shrinking:** this shows whether or not you want a **shrinking heuristic** used in your optimization of the SVM, which is used in Sequential Minimal Optimization. It's default value is true, an **if you don't have a good reason, please don't change this value to false**,

because shrinking will **greatly improve your performance**, for very **little loss** in terms of **accuracy** in most cases.

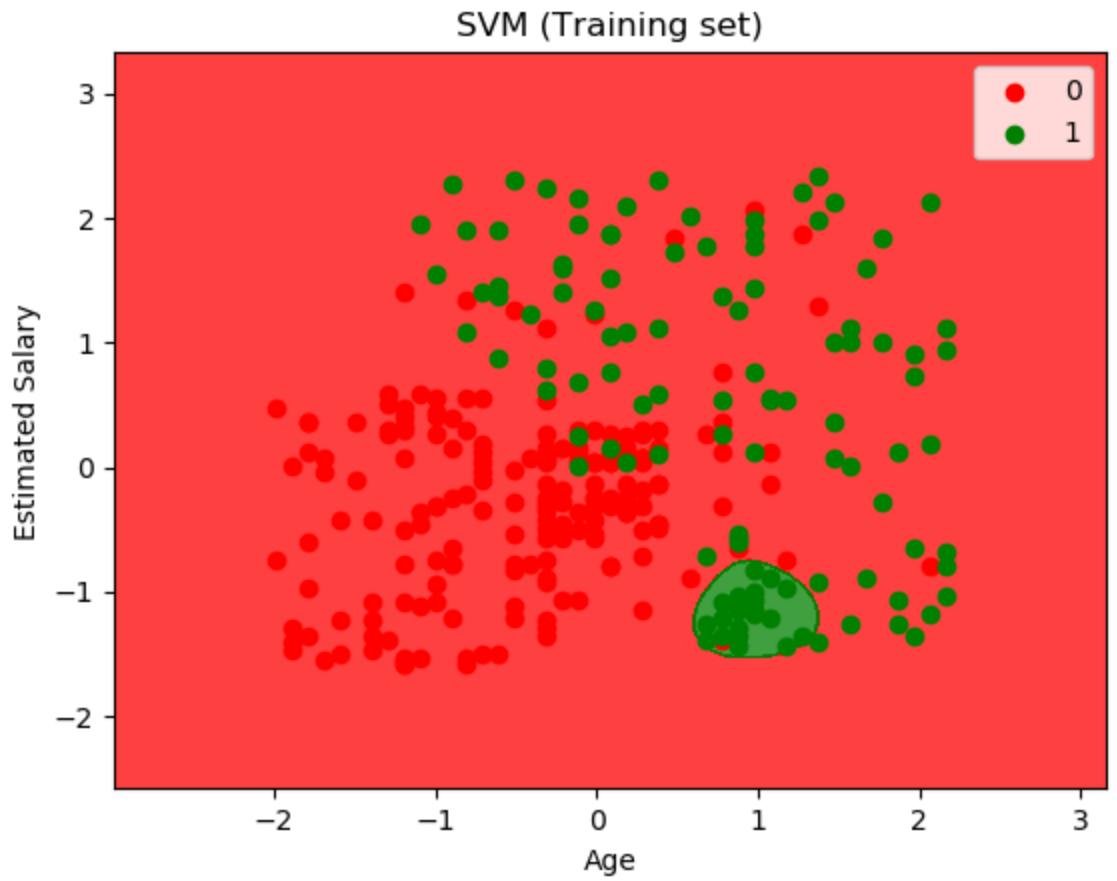
Now lets see the output of running this code. The decision boundary for the training set looks like this:



As we can see and as we've learnt in the Tuning Parameters section, because the C has a small value (0.1) the decision boundary is smooth.

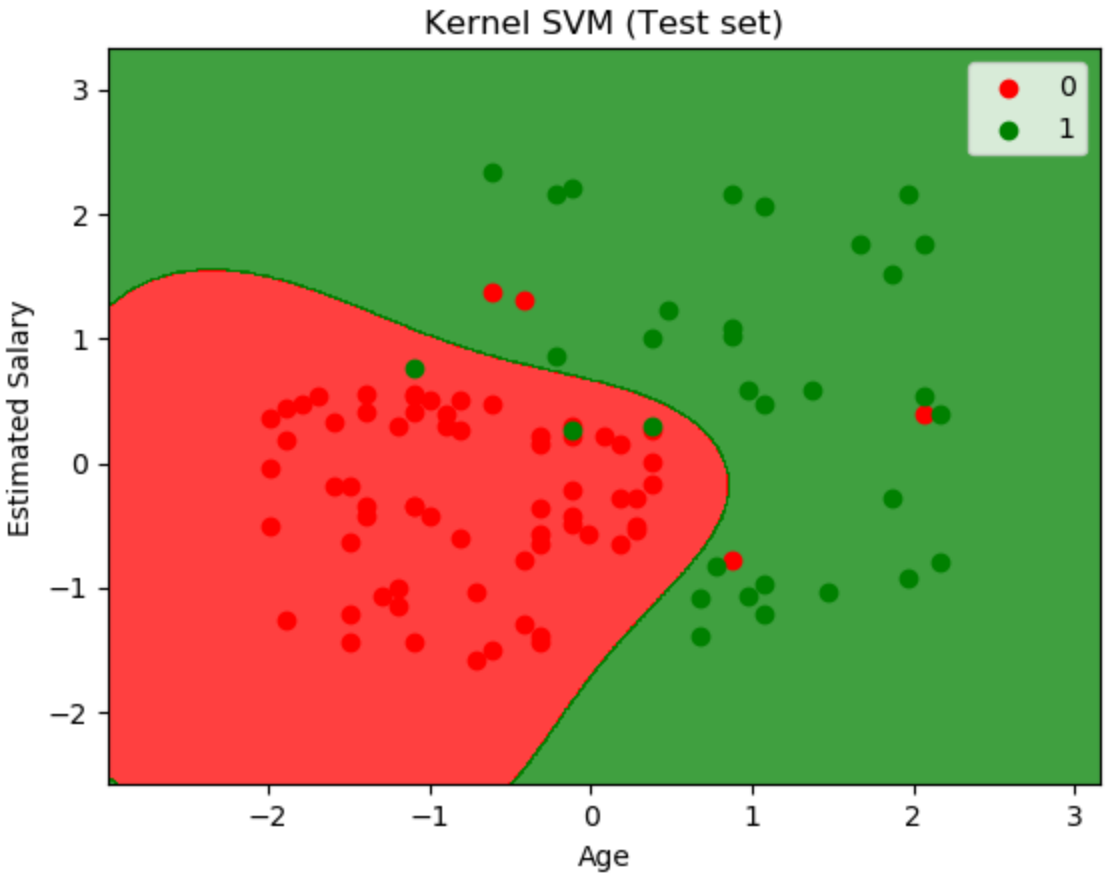
Now if we increase the C from 0.1 to 100 we will have more curves in the decision boundary:

What would happen if we use C=0.1 but now we increase Gamma from 0.1 to 10? Lets see!



What happened here? Why do we have such a bad model? As you've seen in the Tuning Parameters section, **high gamma** means that when calculating the plausible hyperplane we consider **only points which are close**. Now because the **density** of the green points **is high only in the selected green region**, in that region the points are close enough to the plausible hyperplane, so those hyperplanes were chosen. Be careful with the gamma parameter, because this can have a very bad influence over the results of your model if you set it to a very high value (what is a "very high value" depends on the density of the data points).

For this example the best values for C and Gamma are 1.0 and 1.0. Now if we run our model on the test set we will get the following diagram:



And the **Confusion Matrix** looks like this:

64	4
3	29

As you can see, we've got only **3 False Positives** and only **4 False Negatives**. The **Accuracy** of this model is **93%** which is a really good result, we obtained a better score than using KNN (which had an accuracy of 80%).

NOTE: accuracy is not the only metric used in ML and also **not the best metric to evaluate a model**, because of the Accuracy Paradox. We use this metric for simplicity, but later, in the chapter **Metrics to Evaluate AI Algorithms** we will talk about the **Accuracy Paradox** and I will show other very popular metrics used in this field.

Conclusions

In this article we've seen a very popular and powerful supervised learning algorithm, the **Support Vector Machine**. We've learnt the **basic idea**, what is a **hyperplane**, what are **support vectors** and why are they so important. We've also seen lots of **visual representations**, which helped us to better understand all the concepts.

Another important topic that we touched is the **Kernel Trick**, which helped us to **solve non-linear problems**.

To have a better model, we saw techniques to **tune the algorithm**. At the end of the article we had a **code example in Python**, which showed us how can we use the KNN algorithm.