**Ensemble Methods: Gradient Boosting and AdaBoost**

This subunit is all about other forms of ensemble methods, which can be powerful predictors in machine learning. AdaBoost stands for 'adaptive boosting,' and is a machine learning algorithm and ensemble method. There are four steps to AdaBoost:

* Fit a set of decision trees to a dataset
* Calculate the weighted error rate of each decision tree
* Calculate each decision tree's weight in the ensemble
* Increase the weight of the incorrectly classified data points

Then we rinse and repeat, but with the updated weights. AdaBoost is not necessarily a stand-alone method and can be used with other learning algorithms.

Gradient Boosting is another such ensemble method. It makes a prediction by producing a summation of the predictions made by all the trees in an ensemble. The crucial difference between AdaBoost and Gradient Boosting is that with the latter, learner weakness is defined by gradients, whereas with AdaBoost, it's defined by high-weight data points.

You will also look at XGBoost in this subunit. Rather than being an ensemble method, XGBoost is an open-source software library that provides a gradient boosting framework

# Basic Ensemble Learning (Random Forest, AdaBoost, Gradient Boosting)- Step by Step Explained

## A step by step explanation of top 3 tree-based ensemble learning algorithms.

[[](https://medium.com/@learningismessy?source=post_page-----95d49d1e2725--------------------------------)](https://medium.com/@learningismessy?source=post_page-----95d49d1e2725--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----95d49d1e2725--------------------------------)](https://towardsdatascience.com/?source=post_page-----95d49d1e2725--------------------------------)

[Lilly Chen](https://medium.com/@learningismessy?source=post_page-----95d49d1e2725--------------------------------)

·

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----95d49d1e2725--------------------------------)

·

6 min read

·

Jan 2, 2019

A picture containing plant, sky, tree, outdoor

Description automatically generatedPhoto by [kazuend](https://unsplash.com/@kazuend?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

We all do that. Before we make any big decisions, we ask people’s opinions, like our friends, our family members, even our dogs/cats, to prevent us from being biased😕 or irrational😍.

The model does that too. it is very common that the individual model suffers from bias or variances and **that’s why we need the ensemble learning**.

Ensemble learning, in general, is a model that makes predictions based on a number of different models. By combining individual models, the ensemble model tends to be more flexible🤸‍♀️ (less bias) and less data-sensitive🧘‍♀️ (less variance).

Two most popular ensemble methods are **bagging** and **boosting**.

* **Bagging:** Training a bunch of individual models in a parallel way. Each model is trained by a random subset of the data
* **Boosting:** Training a bunch of individual models in a sequential way. Each individual model learns from mistakes made by the previous model.

A screenshot of a computer

Description automatically generated with low confidence

With a basic understanding of what ensemble learning is, let’s grow some “trees” 🎄.

The following content will cover step by step explanation on Random Forest, AdaBoost, and Gradient Boosting, and their implementation in Python Sklearn.

# Random Forest

Random forest is an ensemble model using bagging as the ensemble method and decision tree as the individual model.

Let’s take a closer look at **the magic🔮 of the randomness**:

A screenshot of a computer screen

Description automatically generated with low confidence

Step 1: **Select n (e.g. 1000) random subsets** from the training set

Step 2: **Train n (e.g. 1000) decision trees**

* one random subset is used to train one decision tree
* the optimal splits for each decision tree are based on a random subset of features (e.g. 10 features in total, randomly select 5 out of 10 features to split)

Step 3: **Each individual tree predicts** the records/candidates in the test set, independently.

Step 4: **Make the final prediction**

For each candidate in the test set, Random Forest uses the class (e.g. cat or dog) with the **majority vote** as this candidate’s final prediction.

Of course, our 1000 trees are the parliament here.

# AdaBoost (Adaptive Boosting)

AdaBoost is a boosting ensemble model and works especially well with the decision tree. Boosting model’s key is learning from the previous mistakes, e.g. misclassification data points.

AdaBoost learns from the mistakes by increasing the weight of misclassified data points.

Let’s illustrate **how AdaBoost adapts**.

A screenshot of a computer

Description automatically generated with medium confidence

Step 0: **Initialize the weights** of data points. if the training set has 100 data points, then each point’s initial weight should be 1/100 = 0.01.

Step 1: **Train** a decision tree

Step 2: **Calculate the weighted error rate (e)** of the decision tree. **The weighted error rate (e)** is just how many wrong predictions out of total and you treat the wrong predictions differently based on its data point’s weight. **The higher the weight**, **the more the corresponding error will be weighted** during the calculation of the (e).

Step 3: **Calculate this decision tree’s weight** in the ensemble

the weight of this tree = learning rate \* log( (1 — e) / e)

* the higher weighted error rate of a tree, 😫, the less decision power the tree will be given during the later voting
* the lower weighted error rate of a tree, 😃, the higher decision power the tree will be given during the later voting

Step 4: **Update weights** of wrongly classified points

the weight of each data point =

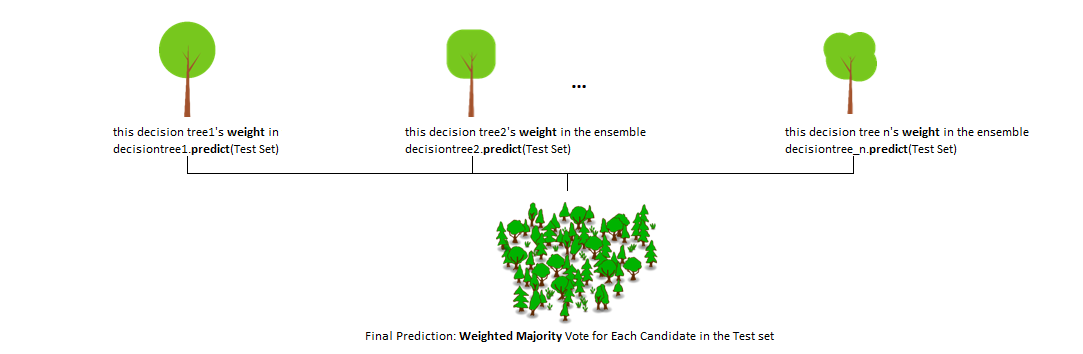
* if the model got this data point correct, the weight stays the same
* if the model got this data point wrong, the new weight of this point = old weight \* np.exp(weight of this tree)

Note: The higher the weight of the tree (more accurate this tree performs), the more boost (importance) the misclassified data point by this tree will get. The weights of the data points are normalized after all the misclassified points are updated.

Step 5: **Repeat** Step 1(until the number of trees we set to train is reached)

Step 6: **Make the final prediction**

The AdaBoost makes a new prediction by adding up the weight (of each tree) multiply the prediction (of each tree). Obviously, the tree with higher weight will have more power of influence the final decision.

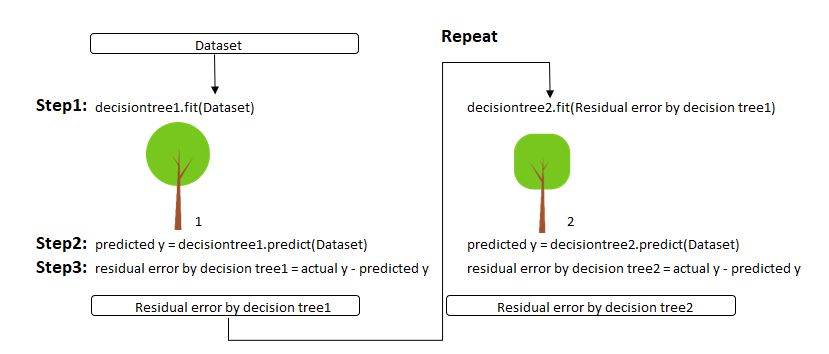


# Gradient Boosting

Gradient boosting is another boosting model. Remember, boosting model’s key is learning from the previous mistakes.

Gradient Boosting learns from the mistake — residual error directly, rather than update the weights of data points.

Let’s illustrate **how Gradient Boost learns.**



Step 1: T**rain** a decision tree

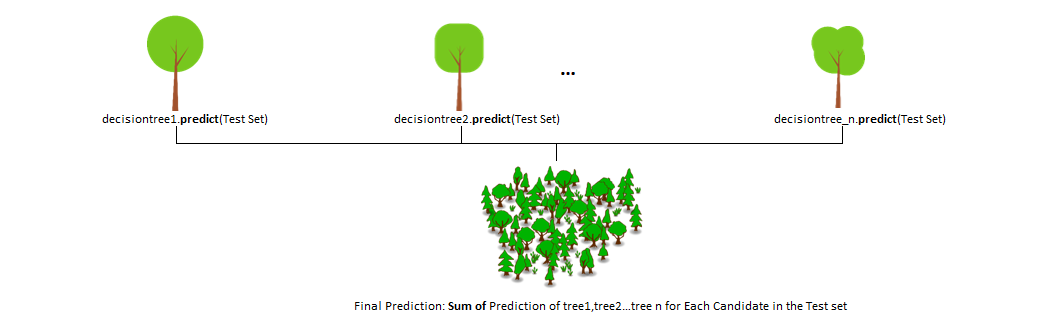
Step 2: **Apply** the decision tree just trained to predict

Step 3: **Calculate** the residual of this decision tree, Save residual errors as the new y

Step 4: **Repeat** Step 1 (until the number of trees we set to train is reached)

Step 5: **Make the final prediction**

The Gradient Boosting makes a new prediction by simply adding up the predictions (of all trees).



# **Implementation in Python Sklearn**

Here is a simple implementation of those three methods explained above in Python Sklearn.

**# Load Library**from sklearn.datasets import make\_moons  
from sklearn.metrics import accuracy\_score  
from sklearn.model\_selection import train\_test\_split  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.ensemble import RandomForestClassifier,AdaBoostClassifier,GradientBoostingClassifier**# Step1: Create data set**  
X, y = make\_moons(n\_samples=10000, noise=.5, random\_state=0)**# Step2: Split the training test set**  
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)**# Step 3: Fit a Decision Tree model as comparison**  
clf = DecisionTreeClassifier()  
clf.fit(X\_train, y\_train)  
y\_pred = clf.predict(X\_test)  
accuracy\_score(y\_test, y\_pred)**OUTPUT: 0.756# Step 4: Fit a Random Forest model, " compared to "Decision Tree model, accuracy go up by 5%**clf = RandomForestClassifier(n\_estimators=100, max\_features="auto",random\_state=0)  
clf.fit(X\_train, y\_train)  
y\_pred = clf.predict(X\_test)  
accuracy\_score(y\_test, y\_pred)**OUTPUT: 0.797# Step 5: Fit a AdaBoost model, " compared to "Decision Tree model, accuracy go up by 10%**  
clf = AdaBoostClassifier(n\_estimators=100)  
clf.fit(X\_train, y\_train)  
y\_pred = clf.predict(X\_test)  
accuracy\_score(y\_test, y\_pred)**OUTPUT:0.833# Step 6: Fit a Gradient Boosting model, " compared to "Decision Tree model, accuracy go up by 10%**clf = GradientBoostingClassifier(n\_estimators=100)  
clf.fit(X\_train, y\_train)  
y\_pred = clf.predict(X\_test)  
accuracy\_score(y\_test, y\_pred)**OUTPUT:0.834Note: Parameter - n\_estimators stands for how many tree we want to grow**

Overall, ensemble learning is very powerful and can be used not only for classification problem but regression also. In this blog, I only apply decision tree as the individual model within those ensemble methods, but other individual models (linear model, SVM, etc.)can also be applied within the bagging or boosting ensembles, to lead better performance.

The code for this blog can be found in my [GitHub Link](https://github.com/lilly-chen/Bite-sized-Machine-Learning/blob/master/Tree-BasedEsemble/Basic%20Ensemble%20Learning%20-%20Sample%20Code.ipynb) here also.

Please feel free to leave any comment, question or suggestion below. Thank you!

# Support Vector Machine — Simply Explained

## The simplistic illustration of basic concepts in Support Vector Machine

[[](https://medium.com/@learningismessy?source=post_page-----fee28eba5496--------------------------------)](https://medium.com/@learningismessy?source=post_page-----fee28eba5496--------------------------------)

[[Towards Data Science](https://towardsdatascience.com/?source=post_page-----fee28eba5496--------------------------------)](https://towardsdatascience.com/?source=post_page-----fee28eba5496--------------------------------)

[Lilly Chen](https://medium.com/@learningismessy?source=post_page-----fee28eba5496--------------------------------)

·

Published in

[Towards Data Science](https://towardsdatascience.com/?source=post_page-----fee28eba5496--------------------------------)

·

8 min read

·

Jan 7, 2019

A picture containing outdoor, sky, cloud, building

Description automatically generated

Photo by [Malte Schmidt](https://unsplash.com/@maltesimo?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

I was always kind of running away from the support vector machine chapter on my ML books. It is just intimidating, you know, the name, Support, Vector, Machine.

But, it becomes less scary once I started to think of support vector machine as a “road machine”, which separates the left,right-side cars, buildings, pedestrians and makes the widest lane as possible. And those cars, buildings, really close to the street is the support vectors.

Feeling better. Let’s get started of understanding how this “road machine” works.

This blog covers three parts:

1. What is the support vector machine
2. How does it work in linear separable scenarios
3. How does it work in linear non-separable scenarios

# What is Support Vector Machine (Classifier)

A picture containing screenshot, line, text, diagram

Description automatically generated

Back to the ML world, those cars, building, pedestrians all become dots now. Red dots stand for objects on the left side of the street; green dots stand for those on the right. The street becomes dashed and solid lines.

Support Vector Machine (the “road machine”) is responsible for finding the decision boundary to separate different classes and maximize the margin.

Marginsare the (perpendicular) distances between the line and those dots closest to the line.

# SVM in linear separable cases

Obviously, infinite lines exist to separate the red and green dots in the example above. SVM needs to find the optimal line with the constraint of correctly classifying either class:

1. Follow the constraint: only look into the **separate** **hyperplanes**(e.g. separate lines), hyperplanes that classify classes correctly
2. Conduct optimization: pick up the one that maximizes the **margin**

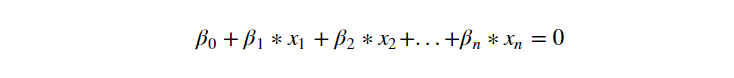
I will illustrate the concept of separate hyperplane and margin, but the explanation of solving the optimization with the constraint will not be covered in this post.

So what is a **Hyperplane**?

A picture containing diagram, line, plot, screenshot

Description automatically generated

Hyperplane is an (n minus 1)-dimensional subspace for an n-dimensional space. For a 2-dimension space, its hyperplane will be 1-dimension, which is just a line. For a 3-dimension space, its hyperplane will be 2-dimension, which is a plane that slice the cube. Okay, you got the idea.



Any Hyperplane can be written mathematically as above

A picture containing font, handwriting, typography, calligraphy

Description automatically generated

For a 2-dimensional space, the Hyperplane, which is the line.

A picture containing font, typography, handwriting, calligraphy

Description automatically generated

The dots above this line, are those x1, x2 satisfy the formula above

A picture containing font, typography, text, calligraphy

Description automatically generated

The dots below this line, similar logic.

What is a **Separating Hyperplane?**

Assuming the label y is either 1 (for green) or -1 (for red), all those three lines below are separating hyperplanes. Because they all share the same property — above the line, is green; below the line, is red.

A picture containing screenshot, line, diagram, plot

Description automatically generated

This property can be written in math again as followed:

A picture containing text, font, typography

Description automatically generated

If we further generalize these two into one, it becomes:



Separate hyperplane constraint is written mathematically above. In the perfect scenario — linear separable scenario, this constraint can be met by SVM. But in the nonseparable scenario, we will need to loosen it.

So what is **margin?**

* Let’s say we have a hyperplane — line X
* calculate the perpendicular distance from all those 40 dots to line X, it will be 40 different distances
* Out of the 40, the smallest distance, that’s our margin!

A picture containing screenshot, line, text, diagram

Description automatically generated

The distance between either side of the dashed line to the solid line is the margin. We can think of this optimal line as the mid-line of the widest stretching we can possibly have between red and green dots.

**To sum up, SVM in the linear separable cases:**

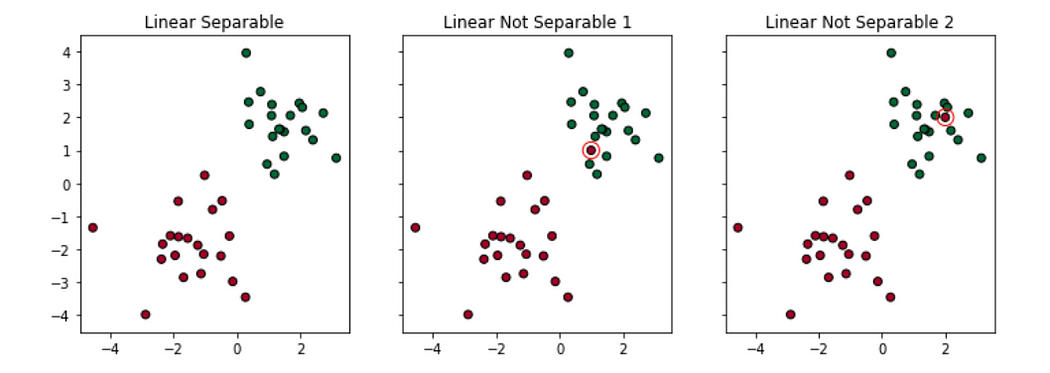
* Constrain/ensure that each observation is on the correct side of the Hyperplane
* Pick up the optimal line so that the distance from those closest dots to the Hyperplane, so-called margin, is maximized

# SVM in linear non-separable cases

In the linearly separable case, SVM is trying to find the hyperplane that maximizes the margin, with the condition that both classes are classified correctly. But in reality, datasets are probably never linearly separable, so the condition of 100% correctly classified by a hyperplane will never be met.

SVM address non-linearly separable cases by introducing two concepts: **Soft Margin** and **Kernel Tricks.**

Let’s use an example. If I add one red dot in the green cluster, the dataset becomes linear nonseparable anymore. Two solutions to this problem:



1. **Soft Margin:** try to find a line to separate, but tolerate one or few misclassified dots (e.g. the dots circled in red)
2. **Kernel Trick:** try to find a non-linear decision boundary

# Soft Margin

Two types of misclassifications are tolerated by SVM under soft margin:

1. The dot is on the wrong side of the decision boundary but on the correct side/ on the margin (shown in left)
2. The dot is on the wrong side of the decision boundary and on the wrong side of the margin (shown in right)

A picture containing line, diagram, plot, screenshot

Description automatically generated

Applying Soft Margin, SVM tolerates a few dots to get misclassified and tries to balance the trade-off between finding a line that maximizes the margin and minimizes the misclassification.

**Degree of tolerance**  
How much tolerance(soft) we want to give when finding the decision boundary is an important hyper-parameter for the SVM (both linear and nonlinear solutions). In Sklearn, it is represented as the penalty term — ‘C’. The bigger the C, the more penalty SVM gets when it makes misclassification. Therefore, the narrower the margin is and fewer support vectors the decision boundary will depend on.

# Default Penalty/Default Tolerance  
clf = svm.SVC(kernel='linear', C=1)  
# Less Penalty/More Tolearance  
clf2 = svm.SVC(kernel='linear', C=0.01)

A picture containing line, screenshot, text, plot

Description automatically generated

# Kernel Trick

What Kernel Trick does is it utilizes existing features, applies some transformations, and creates new features. Those new features are the key for SVM to find the nonlinear decision boundary.

In Sklearn — svm.SVC(), we can choose ‘linear’, ‘poly’, ‘rbf’, ‘sigmoid’, ‘precomputed’ or a callable as our kernel/transformation. I will give examples of the two most popular kernels — Polynomial and Radial Basis Function(RBF).

A picture containing line, plot, diagram

Description automatically generated

**Polynomial Kernel**

Think of the polynomial kernel as a transformer/processor to generate new features by applying the polynomial combination of all the existing features.

To illustrate the benefit of applying a polynomial transformer, let’s use a simple example:

A picture containing screenshot, line, text, plot

Description automatically generated

Existing Feature: X = np.array([-2,-1,0, 1,2])  
Label: Y = np.array([1,1,0,1,1])  
it’s impossible for us to find a line to separate the yellow (1)and purple (0) dots (shown on the left).

But, if we apply transformation X² to get:  
New Feature: X = np.array([4,1,0, 1,4])  
By combing the existing and new feature, we can certainly draw a line to separate the yellow purple dots (shown on the right).

Support vector machine with a polynomial kernel can generate a non-linear decision boundary using those polynomial features.

**Radial Basis Function (RBF) kernel**

Think of the Radial Basis Function kernel as a transformer/processor to generate new features by measuring the distance between all other dots to a specific dot/dots — centers. The most popular/basic RBF kernel is the Gaussian Radial Basis Function:



**gamma (γ)** controls the influence of new features — **Φ(x, center)** on the decision boundary. The higher the gamma, the more influence of the features will have on the decision boundary, more wiggling the boundary will be.

To illustrate the benefit of applying a Gaussian rbf (gamma = 0.1), let’s use the same example:

A picture containing line, diagram, plot, screenshot

Description automatically generated

Existing Feature: X = np.array([-2,-1,0, 1,2])  
Label: Y = np.array([1,1,0,1,1])  
Again, it’s impossible for us to find a line to separate the dots (on left hand).

But, if we apply Gaussian RBF transformation using two centers (-1,0) and (2,0) to get new features, we will then be able to draw a line to separate the yellow purple dots (on the right):  
New Feature 1: X\_new1 = array([1.01, 1.00, 1.01, 1.04, 1.09])  
New Feature 2: X\_new2 = array([1.09, 1.04, 1.01, 1.00, 1.01])

# Note for how we get New Feature 1, e.g. **1.01**:  
Φ(x1,center1) = np.exp(np.power(-(gamma\*(x1-center1)),2)) = 1.01  
# gamma = 0.1  
# center1 = [-1,0]  
# x1 = [-2,0]

Similar to the penalty term — C in the soft margin, Gamma is a hyperparameter that we can tune for when we use SVM with kernel.

# Gamma is small, influence is small  
clf = svm.SVC(kernel='rbf', Gamma=1)  
# Gamma gets bigger, influence increase, the decision boundary get wiggled  
clf2 = svm.SVC(kernel='rbf', Gamma=10)  
# Gamma gets too big, influence too much, the decision boundary get too wiggled  
clf3 = svm.SVC(kernel='rbf', Gamma=20)

A picture containing pattern

Description automatically generated

**To sum up, SVM in the linear nonseparable cases:**

* By combining the **soft margin** (tolerance of misclassifications) and **kernel trick** together, Support Vector Machine is able to structure the decision boundary for linear non-separable cases.
* Hyper-parameters like C or Gamma control how wiggling the SVM decision boundary could be.

1. the higher the C, the more penalty SVM was given when it misclassified, and therefore the less wiggling the decision boundary will be
2. the higher the gamma, the more influence the feature data points will have on the decision boundary, thereby the more wiggling the boundary will be

Finally, that’s it.

The code for this blog can be found in my [GitHub Link](https://github.com/lilly-chen/Bite-sized-Machine-Learning/blob/master/SVM/SVM-Basics.ipynb) here.

A good explanation/rationale of the optimization can be found in the [link](https://www.youtube.com/watch?v=5yzSv4jYMyI&list=PLgIPpm6tJZoShjm7r8Npia7CMsMlRWeuZ&index=1) here, the YouTube Video by Udacity for SVM.

Please feel free to leave any comment, question or suggestion below. Thank you!

**Gradient Boosting**

In the previous resource, you saw that 'gradient descent' is the process whereby each step of training is chosen so as to take the steepest downhill direction in the (mean squared) error. Gradient descent is not to be confused with 'gradient boosting'.

This resource will distinguish gradient boosting from random forest, allowing you to build off content that you already have a firm grasp of. We will see that while random forest involves the bagging methodology, parallel training, easy tuning, and unweighted voting for the final prediction, gradient boosting entails iterative training, weighted voted for the final prediction, and more difficult tuning. They share, however, the characteristics of ensemble methods and decision-tree based methods.

Please work from the 'Introducing gradient boosting' video to the end of the chapter, including the chapter quiz.

## Question 1 of 2

Which one of the following is not a benefit of ensemble methods?

* 

can be used for classification

* 

accepts various types of inputs

* 

very likely to overfit

* 

easily handles outliers

## Question 2 of 2

Accuracy is defined as \_\_\_\_\_.

* 

# predicted as spam that are actually spam / total # that are actually spam

* 

Unsupervised learning

* 

#predicted correctly / total number of observations

This is the correct formula for accuracy

* 

# predicted as spam that are actually spam / total # predicted as spam

# Gradient Boosting from scratch

[[](https://medium.com/@pgrover3?source=post_page-----1e317ae4587d--------------------------------)](https://medium.com/@pgrover3?source=post_page-----1e317ae4587d--------------------------------)

[[ML Review](https://blog.mlreview.com/?source=post_page-----1e317ae4587d--------------------------------)](https://blog.mlreview.com/?source=post_page-----1e317ae4587d--------------------------------)

[Prince Grover](https://medium.com/@pgrover3?source=post_page-----1e317ae4587d--------------------------------)

·

Published in

[ML Review](https://blog.mlreview.com/?source=post_page-----1e317ae4587d--------------------------------)

·

8 min read

·

Dec 8, 2017

Simplifying a complex algorithm

## **Motivation**

Although most of the [Kaggle](https://www.kaggle.com/datasets) competition winners use stack/ensemble of various models, one particular model that is part of most of the ensembles is some variant of Gradient Boosting (GBM) algorithm. Take for example the winner of the latest Kaggle competition: [Michael Jahrer](https://www.kaggle.com/mjahrer)’s solution with representation learning in [**Safe Driver Prediction**](https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/discussion/44629#250927)**.** His solution was a blend of 6 models. 1 [LightGBM](https://github.com/Microsoft/LightGBM) (a variant of GBM) and 5 Neural Nets. Although his success is attributed to the semi-supervised learning that he used for the structured data, but gradient boosting model has done the useful part too.

Even though GBM is being used widely, many practitioners still treat it as a complex black-box algorithm and just run the models using pre-built libraries. The purpose of this post is to simplify a supposedly complex algorithm and to help the reader to understand the algorithm intuitively. I am going to explain the pure vanilla version of the gradient boosting algorithm and will share links for its different variants at the end. I have taken base DecisionTree code from [**fast.ai**](https://github.com/fastai/fastai) library (fastai/courses/ml1/[lesson3-rf\_foundations.ipynb](http://localhost:8888/notebooks/fastai/courses/ml1/lesson3-rf_foundations.ipynb)) and on top of that, I have built my own simple version of the basic gradient boosting model.

## **Brief description for Ensemble, Bagging, and Boosting**

When we try to predict the target variable using any machine learning technique, the main causes of the difference in actual and predicted values are **noise, variance, and bias**. Ensemble helps to reduce these factors (except noise, which is an irreducible error)

An ensemble is just a collection of predictors which come together (e.g. mean of all predictions) to give a final prediction. The reason we use ensembles is that many different predictors trying to predict the same target variable will perform a better job than any single predictor alone. Ensembling techniques are further classified into Bagging and Boosting.

* **Bagging** is a simple ensembling technique in which we build many independent predictors/models/learners and combine them using some model averaging techniques. (e.g. weighted average, majority vote, or normal average)

We typically take random sub-sample/bootstrap of data for each model, so that all the models are a little different from each other. Each observation is chosen with a replacement to be used as input for each of the model. So, each model will have different observations based on the bootstrap process. Because this technique takes many uncorrelated learners to make a final model, it reduces error by reducing variance. Example of bagging ensemble is **Random Forest models.**

* **Boosting** is an ensemble technique in which the predictors are not made independently, but sequentially.

This technique employs the logic in which the subsequent predictors learn from the mistakes of the previous predictors. Therefore, the observations have an unequal probability of appearing in subsequent models and the ones with the highest error appear most. (So the observations are not chosen based on the bootstrap process, but based on the error). The predictors can be chosen from a range of models like decision trees, regressors, classifiers, etc. Because new predictors are learning from mistakes committed by previous predictors, it takes less time/iterations to reach close to actual predictions. But we have to choose the stopping criteria carefully or it could lead to overfitting on training data. **Gradient Boosting** is an example of boosting algorithm.

A picture containing text, screenshot, font, logo

Description automatically generated

**Fig 1.** Ensembling

A picture containing text, purple, violet, screenshot

Description automatically generated

**Fig 2.** Bagging (independent models) & Boosting (sequential models). **Reference:** [**https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/**](https://quantdare.com/what-is-the-difference-between-bagging-and-boosting/)

## **Gradient Boosting algorithm**

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. (Wikipedia definition)

The objective of any supervised learning algorithm is to define a loss function and minimize it. Let’s see how maths work out for Gradient Boosting algorithm. Say we have mean squared error (MSE) as loss defined as:

A black text on a white background

Description automatically generated with low confidence

We want our predictions, such that our loss function (MSE) is minimum. By using **gradient descent** and updating our predictions based on a learning rate, we can find the values where MSE is minimum.

A picture containing text, font, white, line

Description automatically generated

So, we are basically updating the predictions such that the sum of our residuals is close to 0 (or minimum) and predicted values are sufficiently close to actual values.

## **Intuition behind Gradient Boosting**

The logic behind gradient boosting is simple, (can be understood intuitively, without using mathematical notation). I expect that whoever is reading this post might be familiar with [simple linear regression](https://en.wikipedia.org/wiki/Simple_linear_regression) modeling.

A basic assumption of linear regression is that sum of its residuals is 0, i.e. the residuals should be spread randomly around zero.

A graph with blue dots and a red line

Description automatically generated with low confidence

**Fig 3.** Sample random normally distributed residuals with mean around 0

Now think of these residuals as mistakes committed by our predictor model. Although, tree-based models (considering decision tree as base models for our gradient boosting here) are not based on such assumptions, but if we think logically (not statistically) about this assumption, **we might argue that, if we are able to see some pattern of residuals around 0, we can leverage that pattern to fit a model.**

So, the intuition behind gradient boosting algorithm is to repetitively leverage the patterns in residuals and strengthen a model with weak predictions and make it better. Once we reach a stage where residuals do not have any pattern that could be modeled, we can stop modeling residuals (otherwise it might lead to overfitting). Algorithmically, we are minimizing our loss function, such that test loss reaches its minima.

In summary,   
• We first model data with simple models and analyze data for errors.   
• These errors signify data points that are difficult to fit by a simple model.   
• Then for later models, we particularly focus on those hard-to-fit data to get them right.   
• In the end, we combine all the predictors by giving some weights to each predictor.

A more technical quotation of the same logic is written in [Probably Approximately Correct: Nature’s Algorithms for Learning and Prospering in a Complex World](http://www.amazon.com/dp/0465060722?tag=inspiredalgor-20),

“The idea is to use the weak learning method several times to get a succession of hypotheses, each one refocused on the examples that the previous ones found difficult and misclassified. … Note, however, it is not obvious at all how this can be done”

## **Steps to fit a Gradient Boosting model**

Let’s consider simulated data as shown in scatter plot below with 1 input (x) and 1 output (y) variable.

A picture containing screenshot, text, diagram, line

Description automatically generated

**Fig 4.** Simulated data (x: input, y: output)

Data for above-shown plot is generated using below python code:

**Code Chunk 1.** Data simulation

**1.** Fit a simple linear regressor or decision tree on data (I have chosen decision tree in my code) **[call x as input and y as output]**

**Code Chunk 2.** (Step 1)Using decision tree to find best split (here depth of our tree is 1)

**2.** Calculate error residuals. Actual target value, minus predicted target value **[e1= y - y\_predicted1 ]**

**3.** Fit a new model on error residuals as target variable with same input variables **[call it e1\_predicted]**

**4.** Add the predicted residuals to the previous predictions  
**[y\_predicted2 = y\_predicted1 + e1\_predicted]**

**5.** Fit another model on residuals that are still left. i.e. **[e2 = y - y\_predicted2]** and repeat steps 2 to 5 until it starts overfitting or the sum of residuals become constant. Overfitting can be controlled by consistently checking accuracy on validation data.

**Code Chunk 3.** (Steps 2 to 5) Calculate residuals and update new target variable and new predictions

To aid the understanding of the underlying concepts, here is the link with complete implementation of a simple gradient boosting model from scratch. [**[Link: Gradient Boosting from scratch]**](https://www.kaggle.com/grroverpr/gradient-boosting-simplified/)

Shared code is a non-optimized vanilla implementation of gradient boosting. Most of the gradient boosting models available in libraries are well-optimized and have many hyper-parameters.

## **Visualization of working Gradient Boosting Tree**

Blue dots (left) plots are input (x) vs. output (y) • Red line (left) shows values predicted by decision tree • Green dots (right) show residuals vs. input (x) for ith iteration • Iteration represent sequential order of fitting gradient boosting tree

A picture containing text, diagram, font, screenshot

Description automatically generated

**Fig 5.** Visualization of gradient boosting predictions (First 4 iterations)

A screenshot of a graph

Description automatically generated with low confidence

**Fig 6.** Visualization of gradient boosting predictions (18th to 20th iterations)

We observe that after 20th iteration, residuals are randomly distributed (I am not saying random normal here) around 0 and our predictions are very close to true values. (iterations are called n\_estimators in sklearn implementation). This would be a good point to stop or our model will start overfitting.

Let’s see how our model look like for 50th iteration.

A picture containing line, diagram, text, plot

Description automatically generated

**Fig 7**. Visualization of gradient boosting prediction (iteration 50th)

We see that even after 50th iteration, residuals vs. x plot look similar to what we see at 20th iteration. But the model is becoming more complex and predictions are overfitting on the training data and are trying to learn each training data. So, it would have been better to stop at 20th iteration.

Python code snippet used for plotting all the above figures.

**Code Chunk 4.** Plotting predictions and residuals (fed in 1st code chunk’s loop)

I hope that this blog helped you to get basic intuition behind how gradient boosting works. To understand gradient boosting for regression in detail, I would strongly recommend you to read this amazing article by the faculty at University of San Francisco, Terence Parr (Creator of the [ANTLR parser generator](http://www.antlr.org/)) and Jeremy Howard (Founding researcher at [fast.ai](http://www.fast.ai/)) : [How to explain gradient boosting](http://explained.ai/gradient-boosting/index.html).

**More useful resources**

1. **My github repo and kaggle kernel link for GBM from scratch:**  
   <https://www.kaggle.com/grroverpr/gradient-boosting-simplified/>  
   [https://nbviewer.jupyter.org/github/groverpr/Machine-Learning/blob/master/notebooks/01\_Gradient\_Boosting\_Scratch.ipynb](https://github.com/groverpr/Machine-Learning/blob/master/notebooks/01_Gradient_Boosting_Scratch.ipynb)
2. A detailed and intuitive explanation of gradient boosting: [**How to explain gradient boosting**](http://explained.ai/gradient-boosting/index.html) by Terence Parr and Jeremy Howard
3. **Fast.ai github repo link** for DecisionTree from scratch **(Massive ML/DL related resources)**:   
   <https://github.com/fastai/fastai>
4. Video by [**Alexander Ihler**](https://www.youtube.com/watch?v=sRktKszFmSk&t=311s)**.** This video really helped me to build my understanding.