WHITE PAPER

Machine Learning in Credit Risk Modeling

Efficiency should not come at the expense of Explainability

JAMES

Published by
CrowdProcess Inc.
1177 Avenue of the Americas
10036 NY, USA

Published in July 2017

Design and execution
CrowdProcess, Inc.

Table of contents

Executive Summary

Nowadays, and though it has been around for decades, Machine Learning (ML) is the talk of the town. Yet, it is only a combination of theoretical breakthroughs and growing computing power at an affordable cost that made it possible for companies like James (The Artificial Intelligence (AI) for Credit Risk) to include it in their product.

The question is: as it is disrupting many sectors of our societies, from Medicine to the Transportation Industry, why does ML make Credit Risk estimation more efficient? Also, and most importantly, why doesn't this Efficiency come at the cost of Explainability?

In the present paper, we answer these questions thoroughly by applying ML techniques to build non-linear and non-parametric forecasting credit risk models. With the latter, we are able to significantly improve the prediction of defaults with gains in Gini reaching 27%. Further on, we explain the main models used in James in order to fight the *black box myth* around ML.

Results

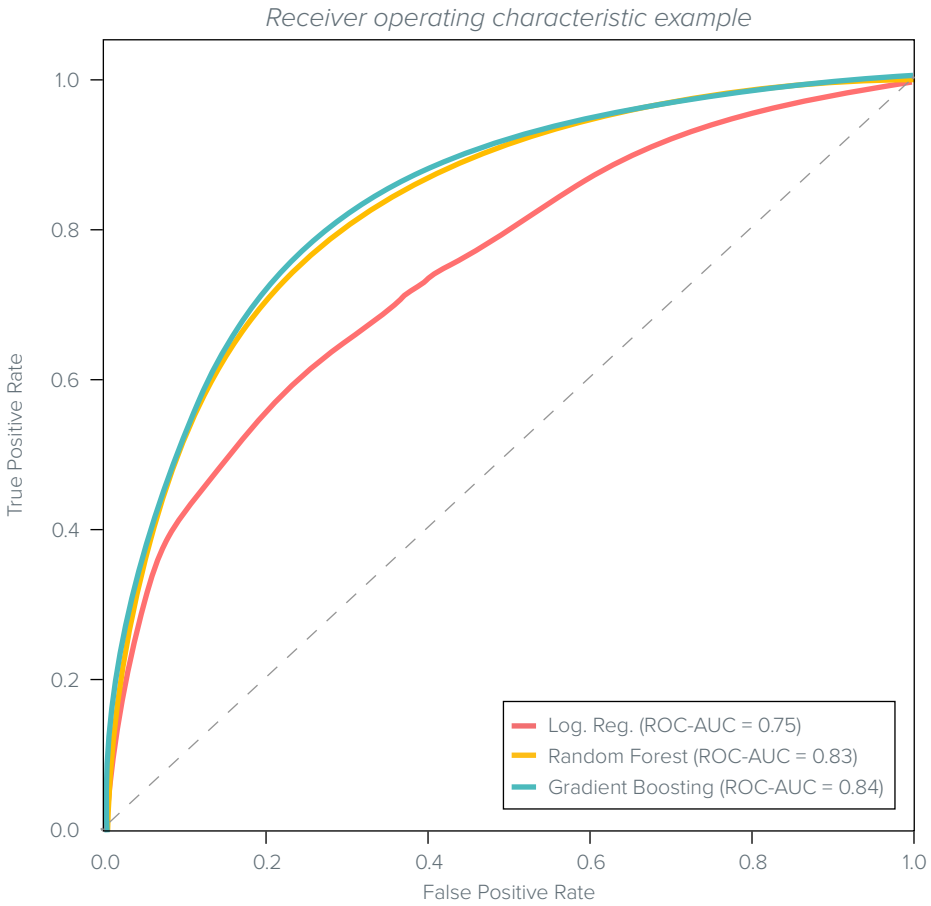
Presentation of the results

In order to prove that ML is an efficient tool when it comes to Credit Risk estimation, we work with a typical Credit Risk dataset of approximately 150,000 observations and 12 features, including the default label.

After identifying the label, we preprocess the data in James, eliminating collinear features and binning the features when needed. We then use cross-validation to train our models on one part of the dataset (80%) and test it on the remaining 20% sample. For more information on the way James works and processes the data, feel free to check the Finovate 2017 James demo on our website.

Here are the results for three selected models: the regular **Logistic Regression** and two state-of-the-art Machine Learning algorithms: **Random Forest** (RF) and **Gradient Boosting** (GB).

The ROC-AUC score (presented in the following section) equals to **0.75** for the Logit model, reaches **0.83** for RF and goes up to **0.84** for GB. The gain in prediction quality is obvious with **a gain of up to 9% in ROC-AUC score.** In parallel, the same happens with the Gini score with a **gain of up to 27%** from Logit to GB.



Scoring a model

We need to define the performance metrics that are often used in ML applied to credit risk. The most common one is the **AUC** (Area under curve) that derives from the **ROC** curve (Receiving Operator Characteristic).

Let us consider a two-class prediction problem (binary classification), in which the outcomes are labeled either as default (D) or safe (S). In Credit Scoring, we consider defaults as positives. There are four possible outcomes from a binary classifier. If the outcome from a prediction is D and the actual value is also D, then it is called a *true positive* (TP); however if the actual value is S then it is said to be a *false positive* (FP). Conversely, a *true negative* (TN) has occurred when both the prediction outcome and the actual value are S, and *false negative* (FN) is when the prediction outcome is S while the actual value is D.

The four outcomes can be summarized in a 2x2 *confusion matrix,* as follows:

The contingency table can derive several evaluation "metrics". To draw an ROC curve, the **true positive rate** (TPR) and **false positive rate** (FPR) are needed (as functions of some classifier parameter). The TPR defines how many correct positive results occur among all positive samples available during the test. The FPR, on the other hand, defines how many incorrect positive results occur among all negative samples available during the test.

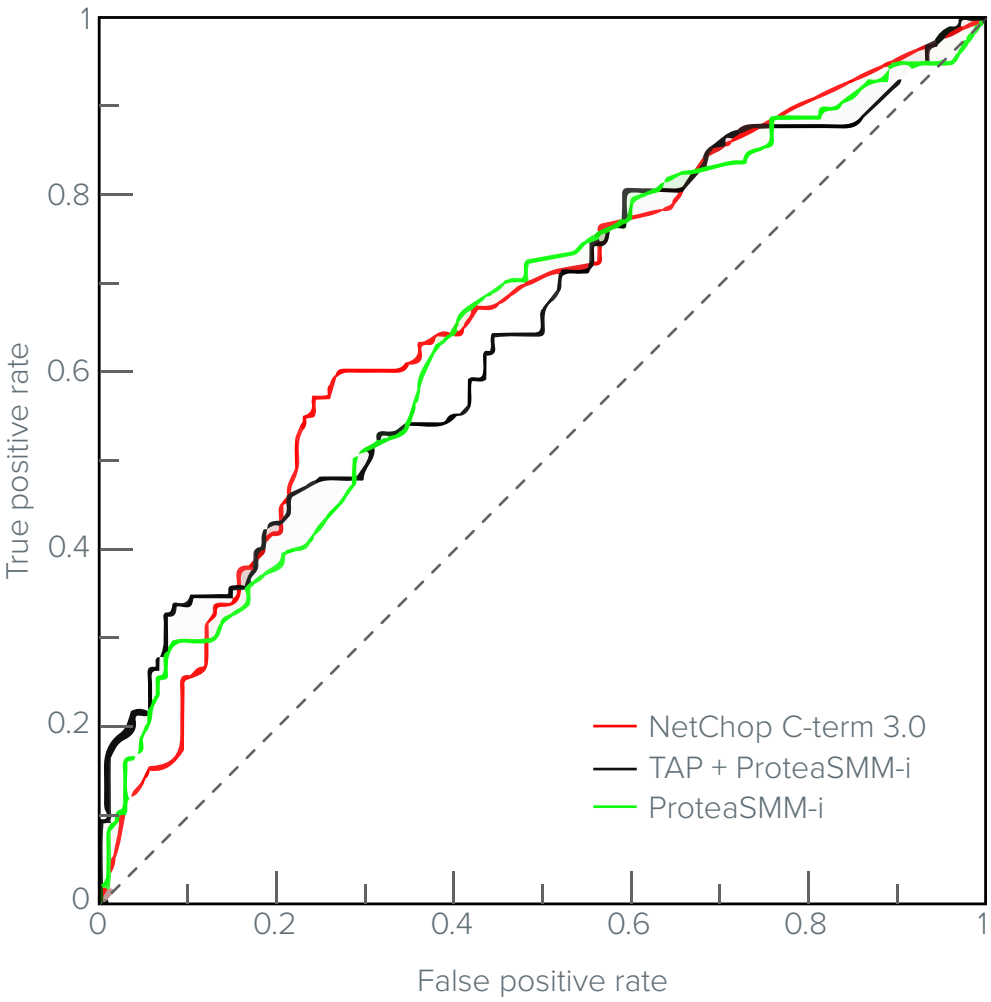
In binary classification, the class prediction for each instance is often made based on a continuous random variable X, which is a "score" computed for the instance (e.g. estimated probability in logistic regression). Given a threshold parameter T, the instance is classified as "positive" if $X > T$ and "negative" otherwise. X follows a probability density $f_1(x)$ if the instance actually belongs to class "positive", and $f_0(x)$ if otherwise. Therefore, the true positive rate is given by $TPR(T) = \int_T^{\infty} f_1(x).dx$ and the false positive

rate is given by $FPR(T) = \int_T^{\infty} f_0(x).dx$. The ROC curve plots parametrically $TPR(T)$ versus $FPR(T)$ with T as the varying parameter.

Here is what the result looks like:

Example of ROC curve from 3 different predictive models.

(source: Wikipedia)



The further the curve is from the diagonal (random picking, often called the *no-discrimination line*), the better the classifier is. That's why it makes sense to calculate the AUC in order to determine the quality of a classifier. The AUC is related to the **Gini coefficient** G_1: $G_1 = 2AUC - 1$

Models and parameters: how does this work?

Introduction to Machine Learning

What is Machine Learning?

ML is a subfield of computer science that "gives the computers the ability to learn without being explicitly programmed" (Arthur Samuel, 1959). More precisely, the main idea is to build algorithms that **can learn from and make predictions on** data.

There are two main types of machine learning:

- the first type of learning is called **supervised learning.** We have a dataset with an input describing the characteristics of given individuals and an output with the **labelling** of these individuals given by a "teacher" (a doctor for instance who defines a tumor as cancerous/non-cancerous) or by reality and experience (in reinforcement learning, a robot might "fall" and learn from its falls). The algorithm learns from the association between the label and the characteristics and is then able to predict the label of an individual given his characteristics.
- the second type of learning is called **unsupervised learning.** This time, no labels are given to the algorithm who has to find structure in its input on its own.

In the case of Credit Risk, we focus on supervised learning since the loans are **labelled** (in this case, "default/safe" or "good/bad") and we are trying to predict whether one given customer is likely to default or not, given their characteristics.

Supervised Learning

Explanation

All algorithms used by James are based on a Machine Learning field called supervised learning.

Example of data file with history (each row represents a loan)

# Loan	Age	Income	Debts	Purpose	Amount	...	Defaulted
1	28	$1000	$20k	Car	$4000		Yes
2	49	$2300	-	Edu	$2000		No
3	55	$600	-	Car	$4000		No
4	67	$4000	$1k	Home	$500		Yes
5	35	$1100	-	Edu	$1000		Yes

Given a **history** of loans, we want to understand which variables (e.g. Age, Income, etc..) will help to predict if the client defaulted on the loan.

With supervised learning, we will create a model which, given the characteristics of the loan (input variables), will estimate its **probability of default.**

Objective Function

In Supervised Learning, a fundamental function is the **objective function** defined as follows:

$$Obj\left(\Theta\right) = L\left(\Theta\right) + \Omega\left(\Theta\right)$$

$L\left(\Theta\right)$ is the **loss function,** measuring how well our model fits to our training data: it measures the quality of the prediction and therefore the **bias** of the latter. We can write the function this way:

$$L\left(\Theta\right) = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i\right)$$

n is the number of individuals, y_i is the label of individual i and $\hat{y}_i$ is the prediction of individual i's label.

Several loss functions can be used depending on the cases:

- the Squared Loss: $l\left(y_i, \hat{y}_i\right) = \left(y_i - \hat{y}_i\right)^2$
- the Absolute Loss: $l\left(y_i, \hat{y}_i\right) = \left|y_i - \hat{y}_i\right|$
- the Logistic Loss: $l\left(y_i, \hat{y}_i\right) = y_i \ln\left(1 + e^{-\hat{y}_i}\right) + \left(1 - y_i\right) \ln\left(1 + e^{\hat{y}_i}\right)$

One advantage of the Absolute Loss over the Squared Loss is that it is **more robust to outliers** since it gives the same weight to all observations. On the other hand, the Squared Loss is more useful if we are concerned about large errors whose consequences are much bigger than equivalent smaller ones.

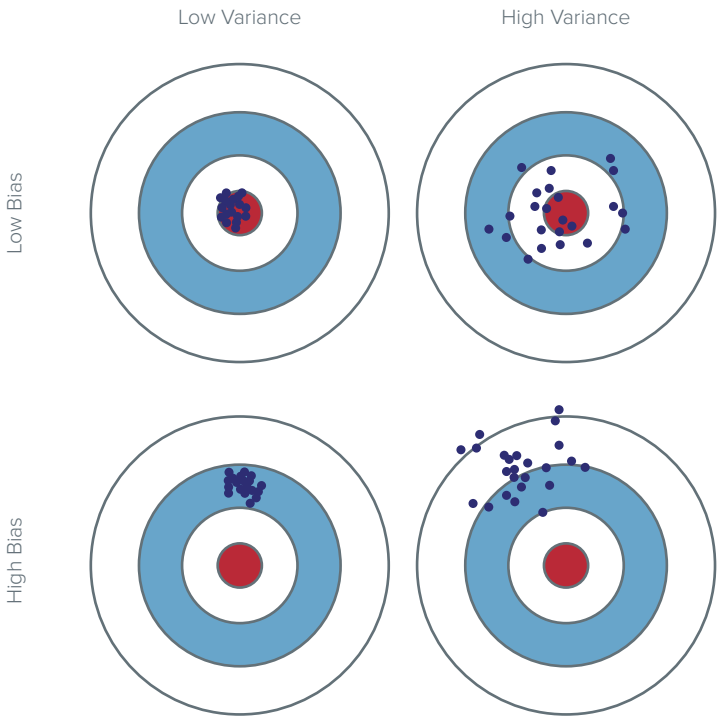
If the problem is a binary classification (i.e. labels are -1 and +1), Logistic loss is convenient although Squared loss may work as well.

$\Omega\left(\Theta\right)$ is the **regularization term,** calculating the complexity of the model and therefore its variance.

For regularization terms, we use different **norms** depending on the purposes:

- L1 norm: $\Omega\left(\beta\right) = \alpha \left\|\beta\right\|_1$
- L2 norm: $\Omega\left(\beta\right) = \alpha \left\|\beta\right\|^2$

In supervised learning, we face a **bias-variance tradeoff:** we want our model to be **predictive (low bias)** without **being too complex (high variance).**

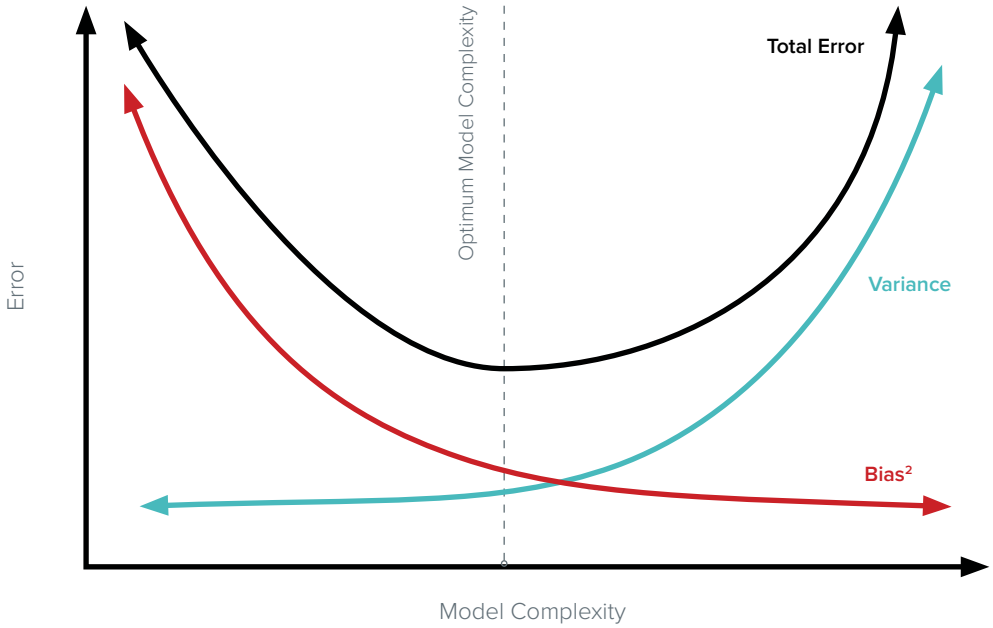


Graphical illustration of bias and variance.

(source: Scott Fortmann–Roe)

Decreasing the bias may lead to high variance and **overfitting**[1]. We therefore have to find a **tradeoff** between the two terms of the objective function we are minimizing in order to reach the optimum model complexity.

[1] Overfitting – "learn by heart" the training sample, building a very complex model that will not make efficient predictions on unseen data.



Bias and variance contributing to total error.

(source: Scott Fortmann–Roe)

Model, Variables, Parameters and Hyperparameters

We refer to the relations which supposedly describe a certain physical situation, as a **model.** Typically, a model consists of one or more equations. The quantities appearing in the equations are classified into **variables** and **parameters.** The distinction between these is not always clear cut, and it frequently depends on the context in which the variables appear. Usually, a model is designed to explain the relationships that exist among quantities which can be measured independently in an experiment; these are the **variables** of the model. To formulate these relationships, however, one frequently introduces "constants" which stand for inherent properties of nature (or of the materials and equipment used in a given experiment). These are the **parameters.**

We use the term **hyperparameter** in order to distinguish from standard model **parameters** that are obtained by fitting the model with the data.

Hyperparameters are the kind of parameters that cannot be directly learned from the regular training process. These parameters express "higher-level" properties of the model, such as its complexity or how fast it should learn, and are usually either fixed before the actual process begins or learned by hyperparameter optimizers such as Grid Search. For non-parametric models (RF, Boosting, ...), parameters and hyperparameters are the same thing (though "hyper" is more proper).

Linear models

When we talk about linear models, there are a lot of models that come to mind. In this paper, we will focus on linear and logistic regression. We won't spend much time on **linear** regressions: feel free to check this Yale course which introduces this model quickly and efficiently.

The baseline to start from in a **binary classification problem** is the binary **logistic regression.** This model, like the ones to follow, is used to estimate the probability of a binary response based on one or more predictors.

Why is logistic regression more successful than linear regression in our case?

Let Y be our target variable with X being a vector including the observed values of all the features.

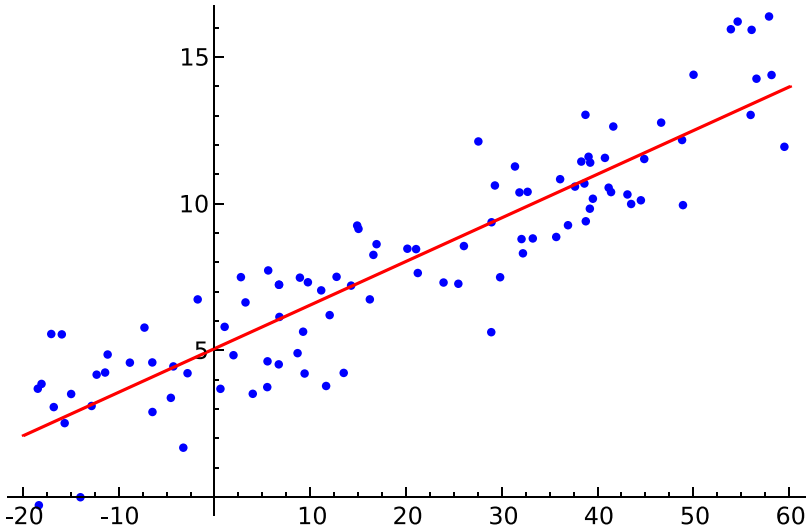
Like in any statistical problem, we assume that the observations are a partial description of a wider phenomenon that we want to know more about.

In a **linear regression,** this phenomenon will be shaped by a straight line whose parameters are found by minimizing the square error (an example is shown in the next figure).

$$Y = \beta_0 + \beta_1 X + \epsilon$$

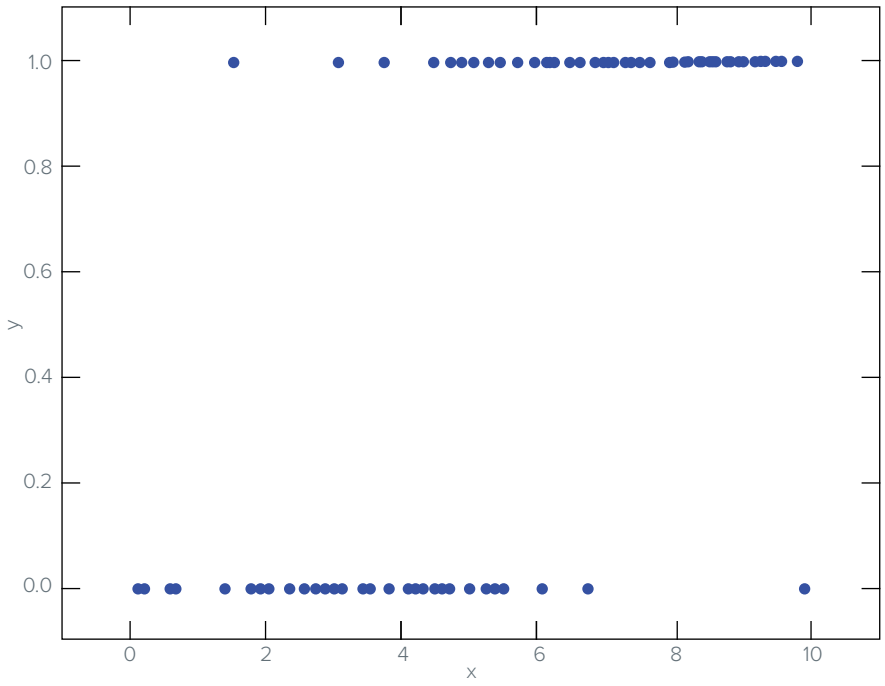
with β_0 as the intercept (5 on the graphic below) and β_1 as the slope.

The graphic below represents the observations (in blue) and the regression line (in red) on a (X,Y) axis.



(source: Wikipedia)

Here, with a binary variable as our target, drawing a straight line would not make much sense. Indeed, our observations look more like this:



How does logit work?

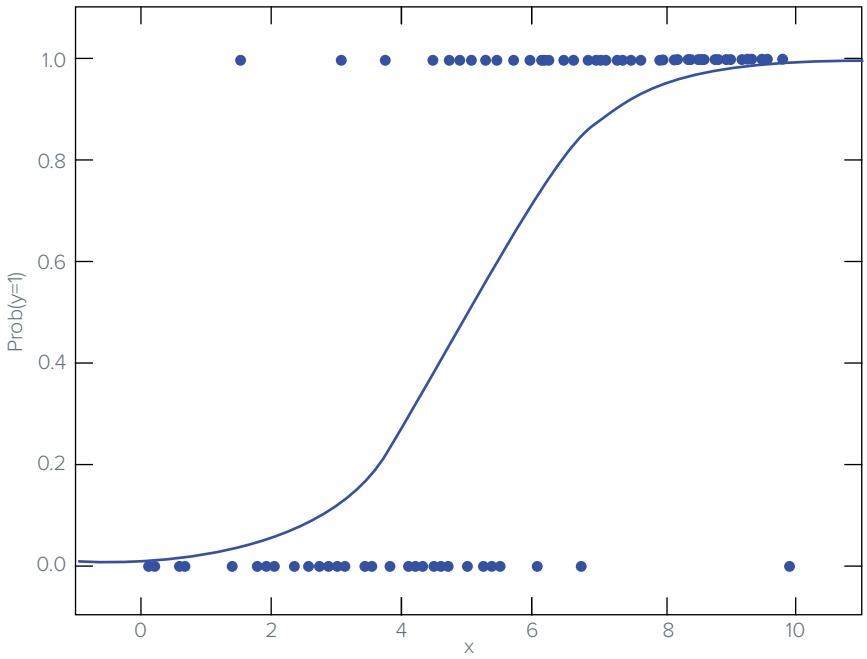
The way to go in this case is to make an assumption about the probability law that defines the probability of Y being 1 given the characteristics of an individual (otherwise written $P(Y = 1 \mid X = x)$).

In the case of a **logistic regression,** we assume that this probability follows a **logistic law** defined by the following cumulative distribution function:

$$F(x) = \frac{1}{1 + exp^{-x}}$$

Let's look at a graphic showing our observations and the logistic cumulative distribution function.

(source: Analytics Vidhya)



Notice that the Y axis is no longer defined by the dependent variable but by the probability that this dependent variable equals to 1 (for instance, it could be the probability that a customer defaults). This probability is what we want to estimate for each customer. Let's get back to the model to see how we will do this:

$$P(Y = 1 \mid X) = \frac{1}{1 + exp^{-\beta^T X}}$$

Indeed, we made the assumption that our model follows a logistic law but we still don't know what the parameters of this law are. Therefore, the parameter we need to estimate here is β of the logistic law.
The way to go about this is to find the parameter β that maximizes the likelihood[2] of Y. Many statistical softwares do this calculation for us and with the estimated parameters, we're able to define **the probability for each individual to default given its characteristics.**

[2] Likelihood – the joint probability of having every Y_i equal to its observed value, either 0 or 1, given the characteristics X.

From the probability we found, we build a **decision boundary** defined as follows: if a customer has a probability higher than a particular value (our threshold, 0.5 by default) to default, the model will predict that this customer will default. Otherwise, the model will predict that it is a safe customer.
This is why the logistic regression is actually not a regression model but a **classification** model and a **linear classifier** to be more precise.

To illustrate this with an example, let's assume that we only have two regressors X_1 and X_2. The model is therefore:

$$P(Y = 1 \mid X) = \frac{1}{1 + exp^{-(\beta_0 + \beta_1 X_1 + \beta_2 X_2)}}$$

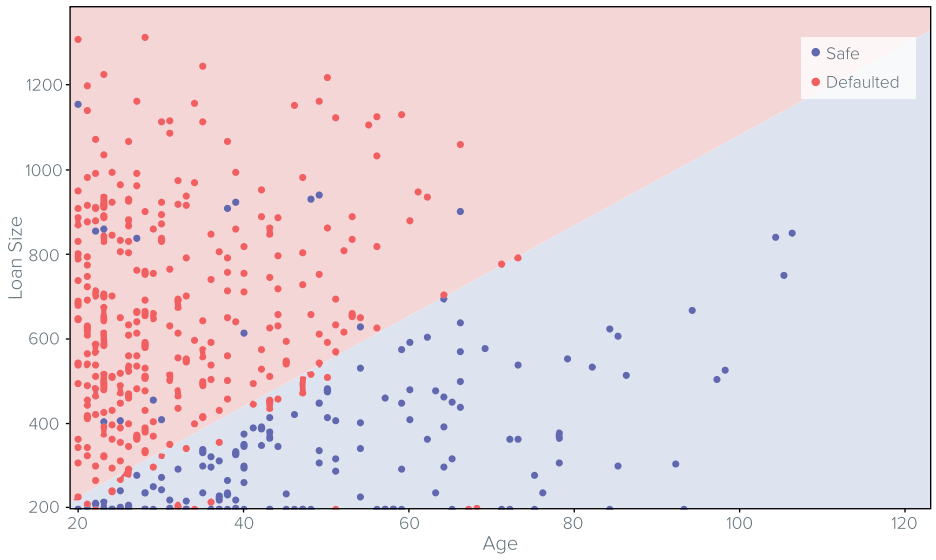
We then estimate β_0, β_1 and β_2 and in order to build the decision boundary line, we choose: $P(Y = 1 \mid X) = 0.5$ which implies that $\beta_0 + \beta_1 X_1 + \beta_2 X_2 = 0$

We then find the equation of the decision boundary:

$$X_2 = -\frac{\beta_1}{\beta_2} X_1$$

Results overview

If we take the loan size and the age as regressors and an example dataset, the result looks like the following graphic. Each point is an individual located by his age (X axis) and loan size (Y axis). The color defines the individual's situation: a blue dot refers to a safe individual and a red dot refers to an individual that defaulted. The background color refers to the algorithm's prediction: a red background implies that the algorithm predicted a default for the individuals located in this zone and vice versa.

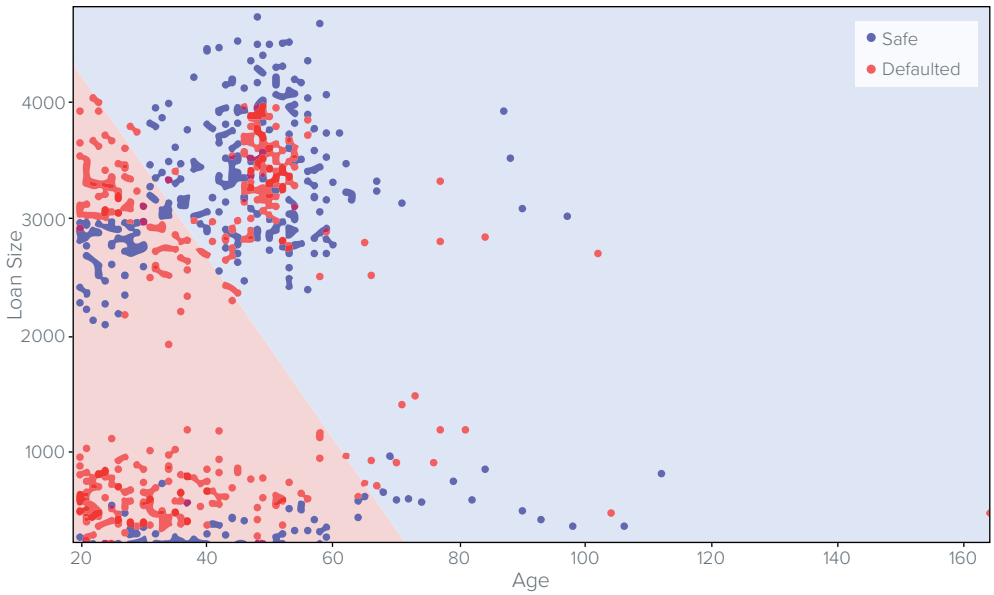


The result looks fairly good on this dataset and will look fairly good in general as long as you expect **your features to be roughly linear** and **the problem to be linearly separable** as it is more or less here.

Logistic regression can also be useful in its **Least Absolute Shrinkage and Selection Operator (LASSO)** and/or **Ridge** version(s). The idea is to create models in the presence of a great number of features. As model complexity increases, the number of coefficients goes up very fast. This is why putting a constraint on the magnitude of coefficients is a good way to reduce the **model's complexity.** It therefore allows one to reduce overfitting and to reduce the probable computational challenges of big datasets (complexity and speed). This is basically what **Ridge** does

by adding a penalization term x (L2 norm defined earlier) to the log-likelihood in the maximization part. The particularity of **LASSO** is that it also allows one to do a **feature selection** by putting down to zero the least important features after a certain value of α (use of L1 norm defined earlier).

Nevertheless, logistic regression has disadvantages: for instance, high correlation between features can become a problem above a certain threshold (multicollinearity). Also and more simply, what if our features are not linear?



We observe in the above graphics that in this case, the logistic regression is not an efficient tool to separate our data.

Conclusion on linear models

- Model: how to make prediction $\hat{y}_i$ given x_i.
  - Linear model: $\hat{y}_i = \sum_j \beta_j x_{ij}$ (include linear/logistic regression);
  - The prediction score $\hat{y}_i$ can have different interpretations depending on the task:
    - In linear regression, $\hat{y}_i$ is the predicted score;
    - In logistic regression, $1/(1+e^{-\hat{y}_i})$ is the probability of the instance being positive.

- Parameters:
  - Linear model: $\Theta = \{\beta_j \mid j = 1, ..., d\}$, with d being the number of regressors.

- Hyperparameters:
  - C (*classifier__C*) - Strength of regularization, a smaller value implies stronger regularization;
  - Class weight (*classifier__class_weight*) - If null, default and non-default observations have entered the model with the same weight; otherwise it indicates the respective weights;
  - Fit intercept (*classifier__fit_intercept*) - If True, a constant intercept was added to the logistic function;
  - Intercept scaling (*classifier__intercept_scaling*) - The value of the intercept, if one was used;
  - Regularization type (*classifier__penalty*) - L1 or L2 regularization.

- **Advantages:**
  - Good results if you expect your features to be linear and your problem to be linearly separable;
  - Explainability.

- **Disadvantages:**
  - Does not handle categorical features very well;
  - High correlation between variables create issues;
  - Does not work well if the features are not linear;
  - Prone to underfitting.

Decision Trees

Parametric models such as logits make certain assumptions on the data structure that may not be fully correct. This is when other **nonparametric** models come in handy. **Nonparametric** means that we do not make any assumptions as to the form or parameters of a frequency distribution.

Decision Trees are one of them and are an important type of algorithm in Machine Learning.

The classical decision tree algorithm has been around for decades and ensemble models which are based on decision trees, such as Random Forest, are among the most powerful techniques available today.

The goal of a decision tree is to create a model **that predicts the value of a target variable by learning simple decision rules inferred from the historical data.**

In other words, a decision tree is a set of rules used to classify data into categories. It looks at the variables in a data set, determines which are most important, and then comes up with a tree of decisions which best partitions the data. The tree is created by splitting data up by one feature at each step and then counting to see how many are in each bucket after each split.

The splitting rule is explained in the next section and is based on

$$E(T) = -\sum_{i=1}^{d} p_i log(p_i)$$

which variable (e.g., Income) will best divide the population in 2 new segments (ex: Salary above and below 1000).

Splitting rules and examples

There are several possible splitting criterion in decision trees. **Entropy** is one of them and it is the one we will present here. Gini or node impurity is another such possible criteria.

There are two kinds of entropy we must calculate at each step in order to build a decision tree:

- **the target entropy:** we look at our whole data set and define the number of individuals in each class, which allows us to calculate the p_i (percentage of total individuals in class i) and then the target entropy:

  $$E(T, X) = \sum_{c \epsilon X} P(c) E(c)$$

  d is the number of classes, in our case d=2 (Safe and Default).
- **each feature's entropy:** same principle but different mode of calculation: for each feature X, we use its frequency table.

As an example, if “Play golf” is the target and “Outlook” one feature:

(source: Saed Sayad)

Play golf

Outlook	YES	NO	
Sunny	3	2	5
Overcast	4	0	4
Rainy	2	3	5
			14

$$E(\text{PlayGolf, Outlook}) = P(\text{Sunny})*E(3,2) + P(\text{Overcast})*E(4,0) + P(\text{Rainy})*E(2,3)$$
$$= (5/14)*0.971 + (4/14)*0.0 + (5/14)*0.971$$
$$=0.693$$

So we end up with the target entropy and the entropy of each feature. The **information gain** is defined as follows (with the target T and a feature X):

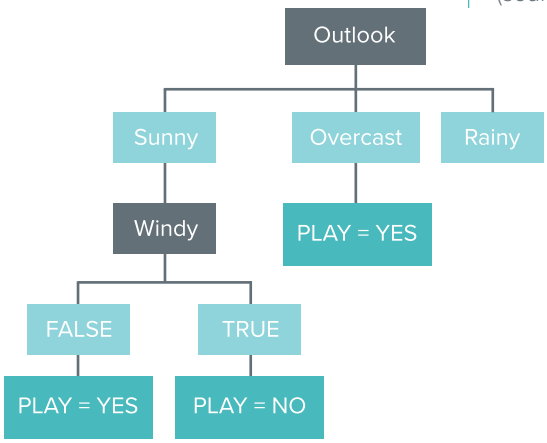
$$Gain(T, X) = E(T) - E(T, X)$$

We choose the feature corresponding to the highest information gain and we use it to make the **split.** In case there is a branch that only has individuals belonging to a unique class (entropy=0), this branch is a **leaf node,** i.e., does not need more splitting.

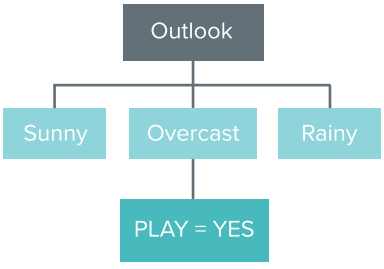
As long as the entropy is strictly higher than 0, the branch benefits from additional splitting.

(source: Saed Sayad)

Temp	Humidity	Windy	Play Golf
Mild	High	FALSE	Yes
Cool	Normal	FALSE	Yes
Mild	Normal	FALSE	Yes
Cool	Normal	TRUE	No
Mild	High	TRUE	No



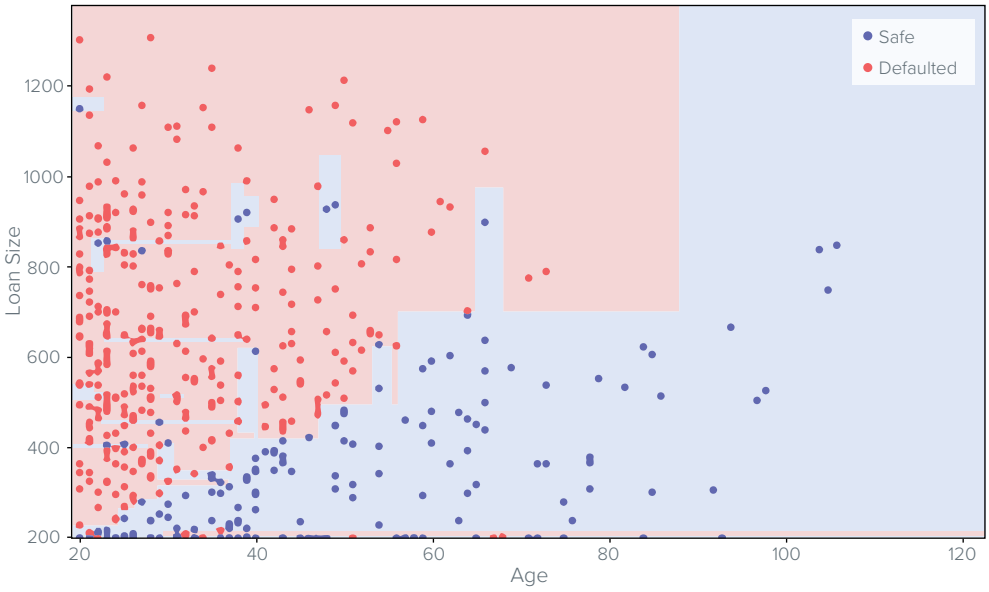
Temp	Humidity	Windy	Play Golf
Hot	High	FALSE	Yes
Cool	Normal	TRUE	Yes
Mild	High	TRUE	Yes
Hot	Normal	FALSE	Yes



To make predictions on new data, the algorithm will apply the set of rules to the new observations. This will tell us to which "endpoint" and which class the observation belongs to.

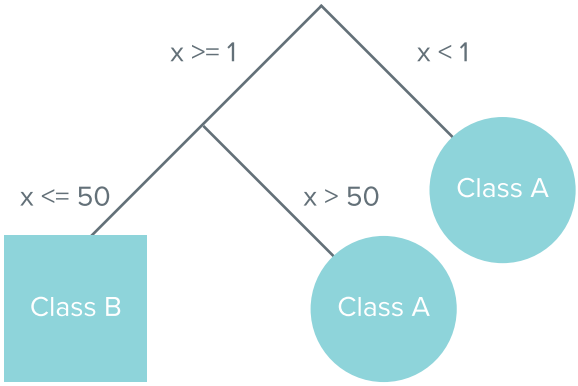
Results overview

Here is an example of a classification made by a decision tree, with the same dataset that was used earlier for the logistic regression example.



[3] Outliers – individuals with very different characteristics from the rest of the population.

We can see that the classification is already more precise, partly because **it does not expect the features to be linear. Outliers**[3] are also handled well by decision trees: the trees will easily take care of the cases where you have class A at the low end of feature X, class B in the mid-range of feature X, and A again at the high end.



In general, decision trees are easy to interpret and to explain but one of their main disadvantages is that they tend to **overfit**. Indeed, the trees are grown to the maximum possible extent so that they reach the lowest possible **bias** but it also implies a big dependence on the training set's characteristics, hence a very **complex** model and therefore a high **variance**.

Conclusion on decision trees

- Model:
  - For *regression,* output is a real figure;
  - For *classification* (which is our case here); output: class that the individuals belong to. If the trees are not grown to their fully extent, the output of each leaf node is the proportion of individuals that belong to the most represented class. In this case, by default, an individual ending up in one leaf node will be classified in the class that is most represented in this node.

- Parameters:
  - Criterion: function used to measure the quality of a split (we presented *entropy* here; *Gini impurity* may alternatively be used);
  - Max depth of the tree: if None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_ samples_split samples;
  - min_samples_split: the minimum number of samples to make a split. (default = 2).

- **Advantages:**
  - Explainability;
  - It does not expect the features to be linear;
  - Handles outliers reasonably well.

- **Disadvantages:**
  - Tends to overfit: poor predictive performance.

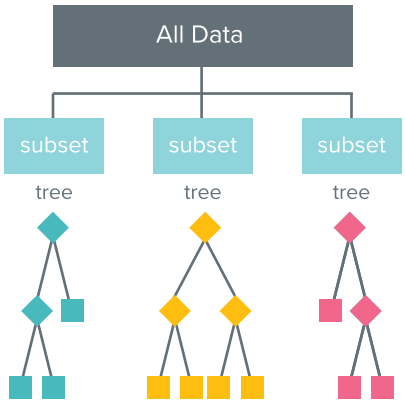
Ensemble Models

To handle the overfitting tendency of the decision trees and reduce the overall variance, ensemble models come in handy. We will present here the two main ensemble models: **Random Forest** and **Gradient Boosting.**

Random Forest

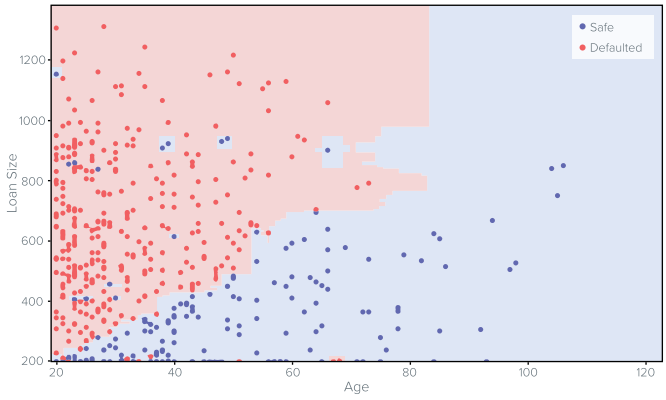
The Random Forest algorithm builds decision trees out of **random samples of the data** and a **random selection of features.** The production phase of random samples is called **bootstrap aggregating** (also known as **bagging**): the idea is to produce m new training sets D_i of size n out of one initial training set D of size n by sampling from D uniformly and with replacement. With the selected population, the tree is built with a different splitting method: **each split is made based on a random subset of features.** Each tree outputs a a class for each individual. In the end, a majority vote takes place: we calculate the proportion of trees that classify one individual in each class and the biggest proportion defines the predicted class membership

(source: Mapr)



Results

With the same example dataset, the results look like this:



In many scenarios, Random Forest gives much more accurate predictions when compared to simple decision trees or linear models.

This is because this technique implies the creation of different decision trees that have **small bias and very (very) high variance.** Thus, taking the average prediction of all these different classifiers **lowers the variance by a lot** while **keeping a fairly small bias.**

Conclusion on Random Forest

- Model:
  - Use bagging to sample the data for each tree;
  - Solve each split using a random subset of features;
  - Majority vote in order to average the prediction.

- (Hyper)parameters:
  - Bootstrap (*classifier__bootstrap*) - indicates if the samples used for building the trees were drawn with the bootstrapping method;
  - Class weight (*classifier__class_weight*) - if 'None', default and non-default observations have entered the model with the same weight; otherwise it indicates the respective weights. 'Balanced' indicates that weights are automatically adjusted to match the proportion of defaults in the training data set;
  - Criterion (*classifier__criterion*) - see Decision Trees;
  - Max depth (*classifier__max_depth*) - maximum depth of the decision trees;
  - n_estimators: the number of trees in the forest;
  - Max features(*classifier__max_features*) - the number of features to use when looking for the best split in decision tree nodes;
  - Min samples leaf (*classifier__min_samples_leaf*) - the minimum number of samples required at each leaf node (as fraction of total number of samples).

- **Advantages:**
  - It is one of the most accurate learning algorithms available. For many data sets, it produces a highly accurate classifier.
  - It runs efficiently on large databases.
  - It can handle thousands of input variables without variable deletion.
  - It gives estimates of which variables are important in the classification.
  - It generates an internal unbiased estimate of the generalization error as the forest building progresses.
  - It has an effective method for estimating missing data and maintains accuracy when a large proportion of the data is missing.
  - It has methods for balancing error in class population unbalanced data sets.
  - Prototypes are computed that give information about the

relation between the variables and the classification.

- It computes proximities between pairs of cases that can be used in clustering, locating outliers, or (by scaling) give interesting views of the data.
- The capabilities of the above can be extended to unlabeled data, leading to unsupervised clustering, data views and outlier detection.
- It offers an experimental method for detecting variable interactions.
- Does not require feature scaling.

- Disadvantages:
  - Random forest has been observed to overfit for some datasets with noisy classification/regression data;
  - Harder to explain relatively to Decision Trees and Logit;
  - Hard to calibrate.

Gradient Boosting

Gradient Boosting is an ensemble technique that is rooted in the concept of **Gradient descent.** The latter is a first-order optimization algorithm that is usually used to calculate a function's local minimum.

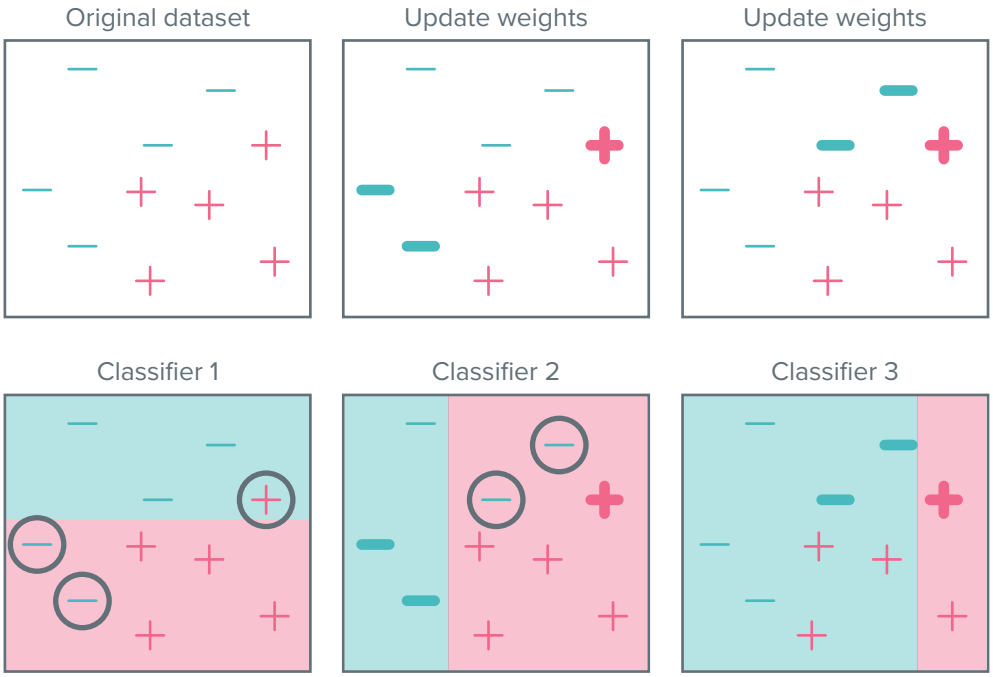
The idea of boosting came from the idea about whether a **weak learner**[4] can be modified to become better. The classifiers are built in a sequential manner and each member of the ensemble is an expert on the errors of its predecessors.

[4] Weak learner – classifier that does slightly better than randomly picking the class.

Boosting: Adaboost

Before explaining **Gradient Boosting** itself, let's focus on an application of Boosting to decision trees: **Adaboost.**

Adaboost is the first practical Boosting algorithm and was introduced by **Y. Freund** and **R. Schapire** in 1996. As you can see on the image below, we first build one classifier h_1 out of our training data. This classifier will perform a good prediction for some individuals and a worse prediction for others (errors circled in black in the image below). The error rate is calculated with a **loss function** defined earlier in the process. From this classifier, we want to iteratively build other classifiers that will do better than the weak learner h_1. To do so, we will give a **higher weight to the individuals that the first classifier misclassified in the training sample of the second classifier** so that this one will **pay more attention to these individuals and classify them in a better way.** The evolution in weights will influence the loss function minimization phase and emphasize the importance of predicting the *heavier* individuals well.



(source: "Ensemble Classifiers", Roberto Henriques)

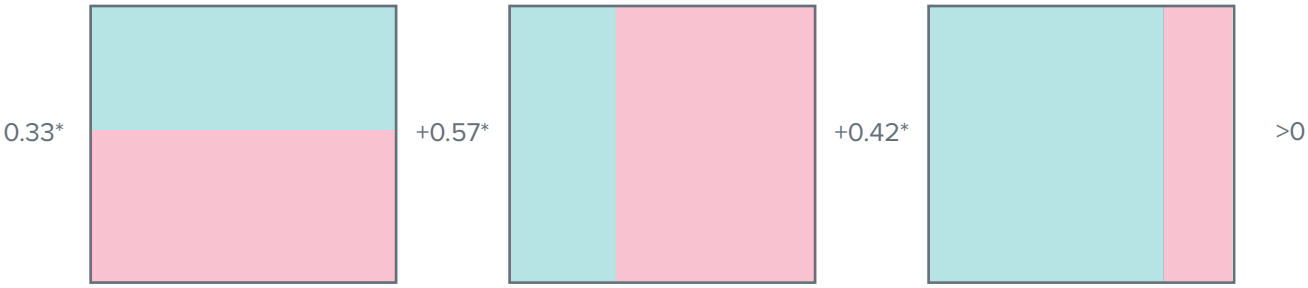
In the end, we have M classifiers that equal to the number of iterations we chose when launching the algorithm. We then evaluate the weight α_t of each of them, based on their error rate ϵ_t . The final classifier is made from the weighted sum of all our classifiers.

- Weigh each classifier:

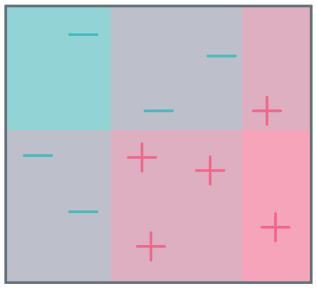
Negative logit function multiplied by 0.5.

$$\alpha_t = \frac{1}{2}.\ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$$

- Weigh each classifier and combine them:



Combined classifier



Adaboost had a lot of success, in face recognition for instance, and Freund and Schapire won the Gödel Prize in 2003 for their work. This raised the attention of the statistics community on Boosting methods and Jérôme Friedman came up in 1999 with **Gradient Boosting** which is a **generalization of Boosting to arbitrary loss functions.**

Introduction to Gradient Boosting: Regression Method (Ben Gorman)

Based on the idea of boosting, we present the intuition of Gradient Boosting in **pseudo-code:**

1. Fit the model to the data: $F_1(x) = y$
2. Fit a model to the residuals: $h_1(x) = F_1(x) - y$
3. Create a new model $F_2(x) = F_1(x) + h_1(x)$

Repeating this model iteratively will keep on improving our classifier.

$$F(x) = F_1(x) \rightarrow F_2(x) = F_1(x) + h_1(x) \rightarrow \cdots \rightarrow F_M(x) = F_{M-1}(x) + h_{M-1}(x)$$

Note that h_M is just a model and **nothing in our definition requires it to be a tree-based model.** This is one of the broader concepts and advantages of Gradient Boosting: the latter is a framework for iteratively improving **any weak learner.** In theory, a Gradient Boosting module would **allow to plug in various classes of weak learners. In practice however, h_M is almost always a regression tree.** Also, note that the number of iterations M is found by testing different values via **cross-validation**.

In order to present an algorithm that is conform to most gradient boosting implementations, we initialize the model with a single prediction value:

$$F_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{x=1}^{n} L(y_i, \gamma)$$

Our first classifier minimizes the total loss. Note that this works with any loss function. For instance, we can define L as the **squared error:** this is what the regression tree does by default. In this case, making the prediction one unit closer to its target will reduce the error a lot more if the prediction is far from the target (check the "Objective Function" part in "Supervised Learning" for more details). Therefore, the tree will focus on improving the predictions that are far from their respective target. On the contrary, the **absolute error** does not discriminate predictions since the improvement of one unit of a predictor towards its target reduces the error by one, no matter the quality of prediction.

With this in mind, suppose that instead of training h_0 on the residuals, we instead train h_0 on **the gradient of the loss function** $L(y, F_0(x))$ with respect to the prediction values produced by $F_0(x)$. **Essentially, we will train h_0 on the cost reduction for each sample if the predicted value were to become one unit closer to the observed value.** In the case of absolute error, h_m will simply consider the sign of every F_m residual (as opposed to squared error which would consider the magnitude of every residual). After samples in h_m are grouped into leaves, an average gradient can be calculated and then scaled by some factor, γ , so that $F_m + \gamma h_m$ minimizes the loss function for the samples in each leaf. (Note that in practice, a different factor is chosen for each leaf.)

Gradient Descent (GD) - a short introduction (Abishek Ghose)

To go on in our explanation, we need to understand the concept of **Gradient Descent.**

Let *f(x)* be the function you want to minimize. Assume x to be a scalar. One way to **iteratively** minimize, and find a (possibly local) minimizer x, is to follow this update rule at the i-th iteration:

$$x_{(i)} = x_{(i-1)} - \eta \frac{\delta f\left(x_{(i-1)}\right)}{\delta(i-1)}$$

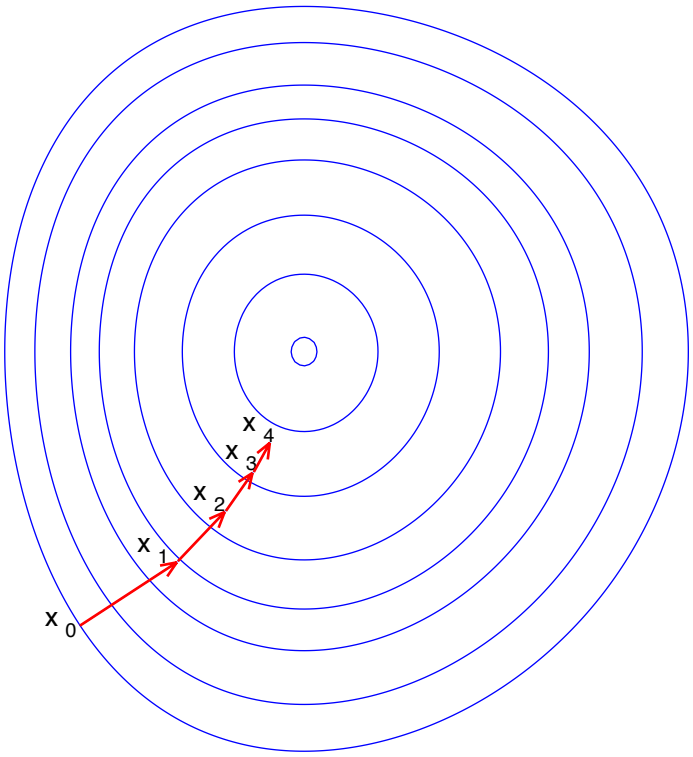
The term that multiplies η is called the **gradient.** Graphically, for a single-variable function, it represents its steepness. It is positive if f is rising and negative if f is decreasing. η is called the **step magnitude** and is a positive constant.

The next value of *x, x(i+1)* is the current value *x(t)* plus a small step against the gradient, that is, towards the direction of steepest descent. That is why it will reach a (local) minima. We stop when $x(i) = x(i-1)$. We may start with an arbitrary value for *x(0)*.

The following figure shows how this works:

Illustration of Gradient Descent on a series of level sets.

(source: Wikipedia)



In the case where x is a vector, the principle remains the same. Now, we adjust every *individual dimension* of *x* based on the slope along that direction. For the *i*-th iteration and the *j*-th dimension, this is what the update rule looks like:

$$x_j^{(i)} = x_j^{(i-1)} - \eta \frac{\delta f(x^{(i-1)})}{\delta x_j^{i-1}}$$

At each iteration *all* dimensions are adjusted. The idea is, we want to move the vector *x,* as a whole, in a direction where *each individual component minimizes f(x).* This is **important** - this form will show up in the subsequent discussion.

Leveraging Gradient Descent: Gradient Boosting for regression

We can now use gradient descent for our gradient boosting model. The objective function we want to minimize is L. Our starting point is $F_0(x)$. For iteration $m = 1$, we compute the gradient of L with respect to $F_0(x)$. Then we fit a weak learner to the gradient components. In the case of a regression tree, leaf nodes produce an average gradient among samples with similar features. For each leaf, we step in the direction of the average gradient (using line search to determine the step magnitude). The result is F_1. Then we can repeat the process until we have F_m.

We modified our gradient boosting algorithm so that it works with any differentiable loss function. Let's clean up the ideas above and reformulate our gradient boosting model once again.

Initialize the model with a constant value:

$$F_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{n} L(y_i, \gamma)$$

For m = 1 to M:

- Compute pseudo residuals,

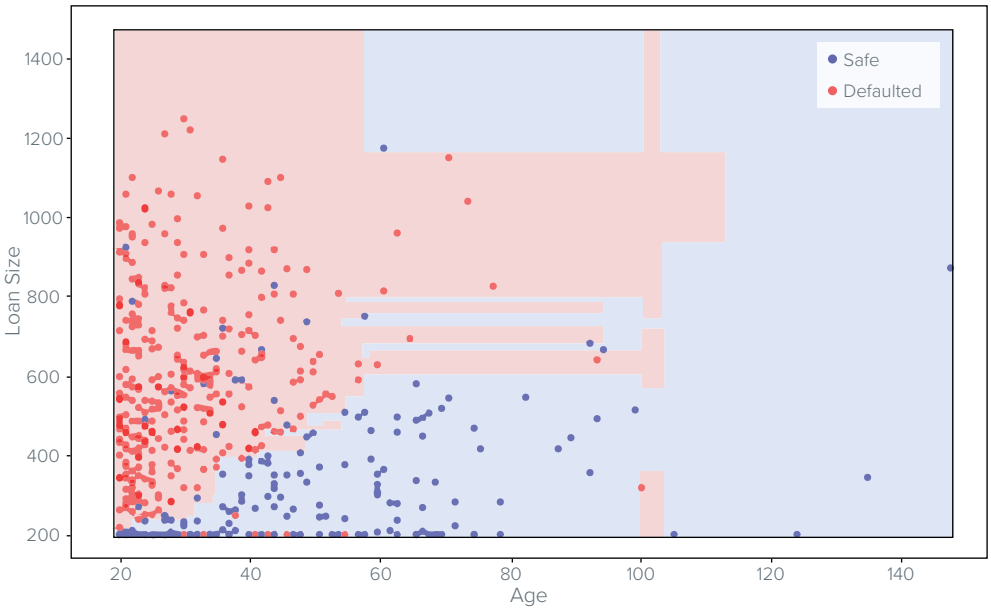
$$r_{im} = -\left[\frac{\delta L(y_i, F(x_i))}{\delta F(x_i)}\right]_{F(x)=F_{m-1}(x)} for i = 1, \cdots, n.$$

- Fit base learner, $h_m(x)$ to pseudo residuals
- Compute step magnitude multiplier γ_m. (In the case of tree models, compute a different γ_m for every leaf.)
- Update $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x)$

For a classification problem, the main difference is that we **don't improve just one model but as many models as the number of classes**. Feel free to read Cheng Li's "Gentle Introduction to Gradient Boosting" for more information on GB for classification.

Results

The results are the following and look similar to Random Forest in this case:



After presenting these two ensemble methods, one could righteously ask themselves which works the best. In 2005, **Caruana & al.** made an empirical comparison of supervised learning algorithms. They included Random Forest and Boosted decision trees and concluded that: *"With excellent performance on all eight metrics, **calibrated boosted trees** were the best learning algorithm overall. Random Forest are close second."*

Conclusion on Gradient Boosting

- Model:
  - Many weak learners together make a strong one;
  - Trees built iteratively relative to the former ones;
  - Each new tree aims at doing better than the former one in the areas it was failing.

- (Hyper)parameters:
  - Learning rate (*classifier__learning_rate*) - degree of contribution of each new decision tree;
  - Loss (*classifier__loss*) - loss function to be optimized by the gradient descent algorithm;
  - n_estimators: number of boosting stages to perform (basically, number of trees);
  - Subsampling (*classifier__subsample*) - Fraction of the number of samples to be used in creating the decision trees;
  - Max depth - see Random Forest;
  - Max features - see Random Forest;
  - min samples leaf - see Random Forest.

- Advantages:
  - Handle heterogeneous data very well (features measured on different scale);
  - Support different loss functions;
  - Automatically detects (non-linear) feature interactions.

- Disadvantages:
  - Longer training (since it's iterative);
  - Requires careful tuning;
  - Prone to overfitting (however there are strategies to avoid it: good tuning of parameters and a big number of boosting stages);
  - Cannot be used to extrapolate.

General conclusion

Overview of algorithms:

This table summarizes everything that has been said on the binary classifiers.

	Logistic Regression	Random Forest	Gradient Boosting
Output	PDs	PDs	PDs
Accuracy	Low	High	Very High
Base Classifier	Log. Reg.	Decision Trees	Decision Trees
Data assumptions	Linear Structure	No assumptions	No assumptions
Main advantage	Traditionally considered more intuitive	High performance with reduced overfitting	High performance with reduced overfitting
Main disadvantage	Low accuracy	Cannot be described as an equation	Cannot be described as an equation, slow to train

Final words

In this paper, we presented the results of the main Machine Learning algorithms applied to Credit Risk estimation. With gains in Gini of **up to 27%, Random Forest** and especially **Gradient Boosting** perform a lot better than **Logistic Regression.**

The aim of this paper was also **to show our customers that the models we use are not black-boxes** and can be interpreted just as a score-card or a logistic regression can be.

Finally, not only are the results better with ML algorithms but most importantly, **James allows you to do in a few hours what is usually done in several weeks.**

This is why our product is **such a valuable tool for financial institutions who wish to get the most out of the growing and promising AI sector.**

We hope you enjoyed learning more about Machine Learning and Credit Risk: if you wish to know more about us, **feel free to contact us at james@james.finance.**

References

Breiman, Leo; "Random Forests" (2001).

Caruana & al, "An Empirical Comparison of Supervised Learning Algorithms" (2005).

Freund, Yoav & Schapire, Robert E.; "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting" (1996).

Gorman Ben, Gradient Boosting Explained (2017).

Henriques, Roberto; "Ensemble Classifiers" (2016 class at Universidade Nova).

Kingsford, Carl & Salzberg, Steven L. ; "What are Decision Trees?" (2008).

Li, Cheng; "A Gentle Introduction to Gradient Boosting" (2015).

Unreferenced content is a product of James' work!