RZ 3401 (# 93464) 02/04/02
Computer Science 21 pages

Research Report

Boosted Decision Trees for Project Risk Assessment and Pricing

Abderrahim Labbi and Michel Cuendet

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland
abl@zurich.ibm.com

Boosted Decision Trees for Project Risk Assessment and Pricing

Abderrahim Labbi Michel Cuendet

Intelligent Business Infrastructure
IBM Zurich Research Laboratory
Saeumerstrasse 4
CH-8803 Rueschlikon

Email: abl@zurich.ibm.com

Abstract

In this paper, we present a predictive modeling approach to project risk assessment and pricing using modern machine learning techniques, namely boosting. In the first section, we present a broad overview of boosting fundamentals and techniques used to build predictive classifiers. We focus in particular on "Stumps", which are one-level decision trees, and Alternating Decision Trees (ADT), which are more complex tree models. We also show how to use boosting techniques for multi-class prediction problems as well as for multi-class probability estimates. In section 2, we present the results of the experiments conducted with various boosting methods using project management data. The implications of using boosted decision trees on risk factor identification and prioritization is presented. We also show the main contribution of the prediction part to the estimation of project outcome probability and severity. In section 3, the estimated probabilities are fed into a Monte Carlo simulation algorithm which provides a better estimate of project financials. The impact of prior estimates on project performance, as estimated by the Monte Carlo process, is measured through a variation of the Kullback-Leibler divergence that we introduce for such purpose. The derived estimates can be used as an additional input to a decision support system for the profiling and the pricing of complex project portfolios.

1 Boosting Decision Trees

1.1 Classification and Decision Trees

The problem is the following. We are given a set of N *examples* or *instances* described by a number of *attributes*. These attributes can be real valued numbers or nominal descriptors. The attribute values characterizing an instance are grouped in the input vector x. Each instance is affected to a class, described by the *label* y. The set of all possible attributes is noted X, and the set of labels Y. In the first part of this work, we only consider the two-class or binary problem, where the labels can only take values in $Y = \{-1; +1\}$. The examples are noted $z = (x, y)$, and they constitute the *training set* Z. We suppose that the examples in Z are drawn independently at random according to an unknown but fixed distribution D over $X \times Y$.

The goal of a classification algorithm is, given the training set, to be able to predict the labels y of unknown instances drawn from D when only x is known. For this, the algorithm builds an hypothesis $H(x)$ such that the *training error*

$$E^{train} = \frac{1}{N} \sum_{i=1}^{N} I(H(x_i) \neq y_i)$$

is minimum. Here $I(.)$ stands for the indicator function.

We assume that D is stationary, and that the training set is representative of this distribution. Then the hypothesis minimizing the training error should also minimize the prediction error on any set of examples not used in the learning process. To assess this in practice, a fraction of the available data is often kept out of the training set, to constitute a *test set* on which the *generalization error* or *test error* can be evaluated.

There are many types of classification algorithms, among which the neural networks, nearest neighbor algorithms, support vector machines, and decision trees. In this work, we only focus on decision trees, of which we give a short description.

In figure 1-1 is plotted a simple decision tree with two decision nodes (rectangles) and three prediction leaves (ellipses). The tree defines a binary classification rule which maps instances of the form $(a, b) \in \mathbb{R}^2$ into one of the two classes denoted by -1 or +1. According to the results of the tests performed at each decision node, an instance is mapped into a path along the tree from the root to one of the leaves.

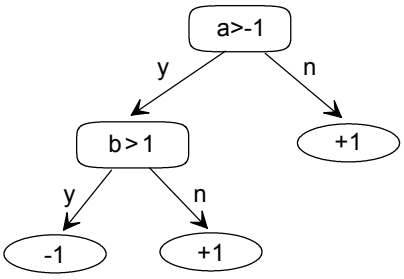

Figure 1-1 : Sketch of a basic decision tree for a binary problem.

The use of a decision tree to classify an instance is straightforward, but the construction of the optimal tree from the data is the challenging part. Usually, the algorithms proceed iteratively, starting at the root, and adding one decision node at a time. The optimal split is found by an extensive search

over all possible splits, and the decision rule minimizing or maximizing a given criterion is kept. We give a few examples of criterions that are commonly used to build decision trees.

1.1.1 The Gini index criterion

Let S be a set of examples representing m classes. Let also p_j be the relative frequency of class j. The *Gini measure of impurity* of S is defined as

$$gini(S) = 1 - \sum_{j=1}^{m} p_j^2$$

It reaches zero if only one class is present in S. The *Gini index* of a split between subsets S_1 and S_2 is a weighted average of the Gini measures of the two subsets:

$$gini[split] = \frac{N_1}{N} gini(S_1) + \frac{N_2}{N} gini(S_2)$$

where N_1 and N_2 are the size of the corresponding subsets, and N is the total number of examples. The split with the lowest value of the Gini index is chosen. The Gini index is used by the popular CART algorithm to grow trees (Breiman et al., 1983). Note that this index does not allow to work with a weighted dataset.

1.1.2 The information gain criterion

Again, let p_j be the relative frequency of class j in a set S of N examples. The information needed to identify the class of an element of S is equivalent to the entropy of the class probability distribution $\{p_1, ..., p_m\}$. Accordingly, the *information* provided by S on the m classes is defined as

$$Info[S] = -[\, p_1 \log(p_1) + ... + p_m \log(p_m) \,].$$

If the classes are uniformly represented in S, the information is $Info(S) = 1$. If all examples belong to the same class, $Info(S) = 0$. Now we define the *split information* for a split of S into subsets S_1 and S_2 containing N_1 and N_2 examples as

$$Info[S_1, S_2] = \frac{N_1}{N} Info(S_1) + \frac{N_2}{N} Info(S_2).$$

The *information gain* is defined by

$$Gain[S, S_1, S_2] = Info[S] - Info[S_1, S_2]$$

It represents the difference between the information needed to identify an element of S before and after the split. It is a measure of the information gain obtained by a the test on one attribute. The decision rule yielding the greatest information gain is adopted as the next node in the tree.This criterion is used in the famous C4.5 algorithm (Quinlan, 1986).

In this work, we don't use any of those two criteria to build decision trees. We use another construction rule, that optimizes the trees for use with boosting. Furthermore, we will in fact only build one-level decision trees (stumps), which relieves us from the problem of *pruning*.

Pruning refers to the limitation of the final size of the tree. It is often required that the iterative splitting stops when a minimum number of instances is reached in a leaf node. Even with this constraint, the trees obtained by repeating the Gini or information gain techniques are quite complex with long and uneven paths. The pruning of the decision tree is done by replacing a whole subtree by

a leaf node, thus canceling all further splits. The replacement takes place if the expected error rate in the subtree is greater than in the single leaf.

1.2 Boosting Techniques

In the previous section, we have seen a way to build a tree classifier directly from the data. However, the generalization accuracy of such a classifier is limited. One can wonder if there is a way to improve this accuracy, for example by combining different decision trees. More generally, can a "weak" learning algorithm which performs just slightly better than random guessing be boosted into an arbitrarily accurate "strong" learning algorithm?

The technique of *boosting* brings an answer to this question. The term boosting was introduced by (Freund and Schapire, 1995) in a milestone paper. The concept of boosting relies on two main intuitive ideas :

- Averaging the predictions of different weak learners to get a more accurate hypothesis.
- Letting the weak learners focus on "hard to learn" examples

We will see shortly how these two benefits are efficiently combined in the AdaBoost algorithm. The sensitive point is how to compute weights for the averaging of the weak learners, and how to make the weak learners focus on hard examples. The boosting algorithm operates iteratively. An example is called "hard" if it has not been classified correctly by the previous weak learners. This can be specified to the next weak learner in two ways. One can use selective resampling of the training set. But the best way is to attribute weights to the examples, especially is their number is limited.

The first versions of AdaBoost were constructed based only on the two intuitive ideas above, and gave surprisingly good experimental results. For example, the generalization error was observed to continue to decrease even after the training error had reached zero. When used with decision trees, AdaBoost seemed to solve the problem of pruning by unraveling the unnecessary complexity. It also showed remarkable results with weak learners as simple as a single decision node.

Several justifications for this appeared later in the literature. First, the theory of margins gave some light on these outstanding capabilities. Theoretical convergence bounds on the training and generalization errors were derived. AdaBoost was proved to iteratively minimizes a loss functional over all possible weak hypotheses. It turned out to have also grounding in the game theory, as well as an independent probabilistic justification, as we will see in the following. Extensive work is still in progress in a number of research groups to settle the theoretical framework of AdaBoost, and to find even more powerful refinements.

1.2.1 The AdaBoost algorithm

In this paragraph we reproduce the original AdaBoost algorithm as it was proposed by (Freund and Schapire, 1995), and refined in subsequent papers from the same authors [5], [6] and [8]. The algorithm is given a training set of N examples $Z = \{z_i = (x_i, y_i), i = 1, ..., N\}$. On this set AdaBoost maintains a weight distribution $W = \left\{ w(z_i) \mid \sum_{i=1}^{N} w(z_i) = 1 \right\}$. It is assumed that we have at hand a weak learning algorithm that guarantees a weighted training error $\varepsilon < 1/2$ for any weights distribution.

AdaBoost

Initialize: $w(z_i) = 1/N$ for all $i = 1...N$.

Do for $t = 1...T$ **:**

1. Train the weak learner with the weighted training set $\{Z, W\}$.
   Obtain hypothesis $h_t(\mathrm{x}) \in \{-1, 1\}$.
2. Calculate the weighted training error of h_t

$$\varepsilon_t = \sum_{i=1}^{N} w_t(z_i)\, I(h_t(x_i) \neq y_i). \tag{1.2-1}$$

3. Set the voting weight

$$a_t = \tfrac{1}{2} \ln \frac{1-\varepsilon_t}{\varepsilon_t}. \tag{1.2-2}$$

4. Update the weights:

$$w_{t+1}(z_i) = \frac{w_t(z_i) \exp\{-y_i\, a_t h_t(x_i)\}}{Z_t}, \tag{1.2-3}$$

   where Z_t is a normalization constant such that $\sum_{i=1}^{N} w_{t+1}(z_i) = 1$.

Output: Final Hypothesis

$$H(x) = \sum_{t=1}^{T} a_t h_t(x).$$

In (Cuendet and Labbi, 2001) we consider some improved versions of AdaBoost. Namely, this version only accepts boolean hypotheses from the weak learner, whereas versions dealing with real valued hypotheses show better performances.

The most general weak hypothesis that we will consider in this work is a rule of the following form

$$h_t : X \mapsto \{h_-; 0; h_+\}, \quad h_- < 0,\ h_+ > 0,$$

where the predicted label is the sign of h_t, and $|h_t|$ is a measure of the confidence of the prediction. In this case, the algorithm remain the same, except that the training error becomes

$$\varepsilon_t = \sum_{i=1}^{N} \omega_t(z_i)\, I(\mathrm{sign}[h_t(x_i)] \neq y_i).$$

1.3 Extension to Multiclass Prediction

The above algorithm is described for binary classification problems. However, many applications require to assign elements to a set of more than two classes. Designing a boosting algorithm that operates directly on multiclass problems is difficult. The reason for this is that the weak learner is

required to guarantee a training error of at most $1/2$. This is easy in the binary case, because it implies only that the weak learner is slightly better than random. An error less than $1/2$ is much harder to achieve in the case of multiple labels.

Another approach to the problem consists in reducing the multiclass problem to multiple binary problems that can be solved separately. There are many ways to decompose multiclass problems to binary and to recombine the classifiers generated. The simplest approach is to create a binary problem for each of the k classes. That is, for each $r \in Y$, all examples labeled $y = r$ are considered positive examples, and all others are considered negative. This is called the *one-versus-all* coding. This approach was applied to AdaBoost by Schapire and Singer [15] and Freund and Schapire [7]. In the following, we derive similar algorithms, but we prefer to use the more general framework of Allwein, Schapire and Singer [1] for the reduction to binary problems. These authors extend their scope to any learning algorithm, whereas we will restrict ourselves to the case of AdaBoost.

Besides the one-versus-all approach, we can cite some other codings. The *all-pairs* coding uses the given binary algorithm to distinguish between each pair of classes. Thus for each $r_1, r_2 \in Y$, the examples labeled $y = r_1$ are positive, and those labeled $y = r_2$ are negative, and all other examples are simply ignored. The *all-partitions* coding uses all possible partitions of Y in two subsets, which are associated to positive and negative labels. Other codings, that can even be chosen at random, also work for decomposing the problem.

The most general approach is to associate each class $r \in Y$ with a row of a *coding matrix*

$$M \in \{-1; 0; +1\}^{k \times \ell}$$

for some ℓ. The binary learning algorithm is run once for each column of the matrix, on the binary problem in which the label of each example labeled y is mapped to $M(y, s)$. That is, for $s = 1 \ldots \ell$, the binary learner is provided with data of the form $(x_i, M(y_i, s))$, for all examples i in the training set, but omitting all examples for which $M(y_i, s) = 0$. This yields ℓ hypotheses $H_s : X \to \mathbb{R}$. Given an example x, we predict the label r for which $(H_1(x), \ldots, H_\ell(x))$ is "closest" to row r of matrix M.

At this point, we have to specify a distance that clarifies the meaning of "closest". We need a distance that takes the magnitude of the predictions of AdaBoost into account, because it is a level of confidence for the binary prediction. This distance should be based on the loss function L of the algorithm. L is a function of the margin, so the loss of H_s on an example x_i labeled $M(y_i, s)$ is $L(\, M(y_i, s) H_s(x_i)\,)$. For AdaBoost, we have already stated that $L(u) = e^{-u}$.

The idea is to choose the label r that is most consistent with the predictions $H_s(x_i)$ in the sense that, if example x was labeled r, the total loss on example (x, r) would be minimal over all choices of $r \in Y$. Formally, this means that our distance measure is the total loss on a given example (x, r):

$$d(M(r), H(x)) = \sum_{s=1}^{\ell} L(\, M(r, s) H_s(x)\,).$$

Here $M(r)$ denotes the row r of M. Then, the predicted label for an instance x is

$$\hat{y} = \arg \min_r \; d(M(r), H(x)).$$

This approach is called the *loss-based decoding*. In other words, in the case of AdaBoost the algorithm is aimed at minimizing

$$G_M(H) = \sum_{s=1}^{\ell} \sum_{i=1}^{N} \exp\{- M(r, s) H_s(x)\,\} \tag{1.3.-1}$$

In the *one-versus-all* framework, M is a $k \times k$ matrix in which all diagonal elements are $+1$, and all others are -1. For the all-pairs approach, M is a $k \times \binom{k}{2}$ matrix in which each column corresponds to a pair (r_1, r_2). For this column, M has $+1$ in row r_1, -1 in row r_2 and zeros on all other rows. In fact, the algorithm can work with any coding matrix, even chosen at random, provided

that there are enough columns, and that the rows are sufficiently "distinct". This last point can be assessed by the *Hamming distance* between two rows $u, v \in \{-1; 0; 1\}^{\ell}$:

$$\Delta(u, v) = \frac{\ell - u \cdot v}{2}.$$

More precisely, we define the minimum distance between two rows

$$\rho = \min\{\Delta(M(r_1), M(r_2)) : r_1 \neq r_2\}.$$

For the one-versus-all code, $\rho = 2$. Intuitively, the larger ρ, the more likely it is that decoding will compensate for errors made by individual hypotheses. For this reason, the method using a random matrix M is called *error correcting output codes* (ECOC). However, there is a tradeoff between the redundancy and correcting properties of the output codes and the difficulty of the binary learning problems it induces. In fact, the one-versus-all code is not much worse than others that have greater ℓ and much greater ρ, because it induces very simple binary problems.

An alternative to calling the weak learner repeatedly, we can instead add the column index s as a separate argument passed with the instances to the weak learner, and then learn a single hypothesis on this larger problem rather than ℓ hypotheses on a smaller problem. That is, we provide the weak learner with instances of the form $((x_i, s), M(y_i, s))$ for all training examples i and all columns s for which $M(y_i, s) \neq 0$. Then, the weak learner produces a single weak hypothesis $h : X \times \{1; ...; \ell\} \to \mathbb{R}$. The boosting algorithm maintains a weight distribution W_t over $X \times \{1; ...; \ell\}$. The algorithm obtained by this method is similar to AdaBoost.MO in Schapire and Singer [15].

This change requires only minor changes to the weak learner described in section 1.8. Suppose that the weak hypothesis is associated with a partition X_1, X_2 of space X. It is natural then to create partitions of $X \times \{1; ...; \ell\}$ consisting of all sets $X_j \times \{s\}$ for $j = 1, 2$ and $s = 1...\ell$. An appropriate hypothesis can then be formed which predicts $h(x, s) = h^{j\ell}$ for $x \in X_j$. According to the results of section 1.8, we should choose

$$h^{j\ell} = \tfrac{1}{2} \ln\left(\frac{W_+^{j\ell}}{W_-^{j\ell}}\right),$$

where $W_{\pm}^{j\ell} = \Sigma_i \, w(x_i, s) I(x \in X_j \wedge M(y_i, s) = \pm 1)$. This gives

$$Z = W_0 + 2 \sum_j \sum_l \sqrt{W_+^{j\ell} W_-^{j\ell}},$$

where $W_0 = \Sigma_i \, w(x_i, s) I(x_i \text{ missing})$.

1.3.1 Multiclass Stumps

Initialize:

1. Weights $w(z_i, s) = 1/(N, \ell)$ for all $i = 1...N$, and $s = 1...\ell$.
2. Build coding matrix $M(y, s)$ for $y \in Y$ and $s = 1...\ell$.

Do for $t = 1...T$ **:**

1. Train the weak learner with the weighted training set $\{(X \times \{1; ...; \ell\}, M), W_t\}$. Obtain hypothesis $h_t(x, s) \in \mathbb{R}$.

2. Update the weights:

$$w_{t+1}(x_i, s) = \frac{w_t(x_i, s) \exp\{-M(y_i, s)\, h_t(x_i, s)\}}{Z_t},$$

where Z_t is a normalization constant such that $\sum_{i=1}^{N} \sum_{s=1}^{\ell} w_{t+1}(x_i) = 1$.

Output:

1. Binary Hypothesis

$$H_{\text{bin}}(x, s) = \sum_{t=1}^{T} h_t(x, s).$$

2. Final Hypothesis

$$H(x) = \arg \min_{y \in Y} \sum_{s=1}^{\ell} \exp\{-M(y, s) H_{\text{bin}}(x, s)\}$$

The Alternating decision tree (Freund and Mason, 1999) can also be generalized to the multiclass problem. We don't give the algorithm here because the adaptation is very similar to stumps. Note that to the best of our knowledge the literature does not report any experiments with a multiclass ADT.

1.3.2 Getting Probabilities in the Multi-class Case

We follow here Friedman et al. [10] in a natural generalization of the two-class case. It is based on the one-vs-all approach, where k binary hypotheses are built for a k class problem. Let $y_s = M(y, s),\ s = 1...k$, according to the notations introduced above, which means here $y_s = +1$ if $y = s,\ y_s = -1$ otherwise. We note $p_s(x) = P(y_s = 1 \mid x)$.

According to equation (1.3.-1), the multiclass AdaBoost minimizes $\Sigma_s E(e^{-y_s H_s})$. The multiclass problem is equivalent to k binary problems minimizing $E(e^{-y_s H_s})$. We can apply lemma 3 [10] to this case. The proof is essentially the same, noting that $P(y_s = -1 \mid x) = 1 - p_s(x)$. This yields result 5 of Friedman et al. [10]: AdaBoost for a k-class problem fits k uncoupled additive logistic models

$$H_s(x) = \tfrac{1}{2} \log \frac{p_s(x)}{1 - p_s(x)}.$$

Hence, the class probabilities are

$$p_s(x) = \frac{e^{H_s(x)}}{e^{-H_s(x)} + e^{H_s(x)}}.$$

A problem remains here, because we are estimating the p_s in an uncoupled fashion, and there is no guarantee that the implied probabilities sum up to 1. Friedman et al. explain that AdaBoost is in fact an approximation for fitting the symmetric multiple logistic transformation is defined by

$$H_s(x) = \log p_s(x) - \tfrac{1}{k} \sum_{s=1}^{k} \log p_s(x)$$

This solves the problem in a coupled fashion, and the resulting probabilities are normalized

$$p_s(x) = \frac{e^{H_s(x)}}{\sum_s e^{H_s(x)}}.$$

2 Boosting Project Risk Assessment

In this section, we show some experimental results that we have obtained using boosting techniques for multi-class prediction. We present the results for several variations of the original AdaBoost algorithm described earlier in this paper. These variations have been introduced and discussed in a separate study. Further details can be found in (Cuendet and Labbi, 2001).

To test the effectiveness of the boosting algorithm, we have focused our study on a project management database which contains data entries describing various types projects. Each entry is represented by a set of attributes which describe project demographics, financials, management, performance, etc. Among the attributes are performance attributes which one would like to estimate a priori so that pre-emptive management actions can be undertaken. We focus our prediction tasks here on `Status` of projects, but other parameters can also be predicted similarly.

The labels are the `Status` ratings (similar to ratings of a Balanced ScoreCard system) which are usually described by an ordered set such as $\{1;2;3;4\}$ or $\{A;B;C;D\}$.

The boosting methods tested here are the following (Cuendet and Labbi, 2001):

- Plain Stumps adapted to the multilabel case
- Favoring Stumps : the approximate cost-sensitive solution
- Alternating Decision Trees adapted to the multilabel framework.

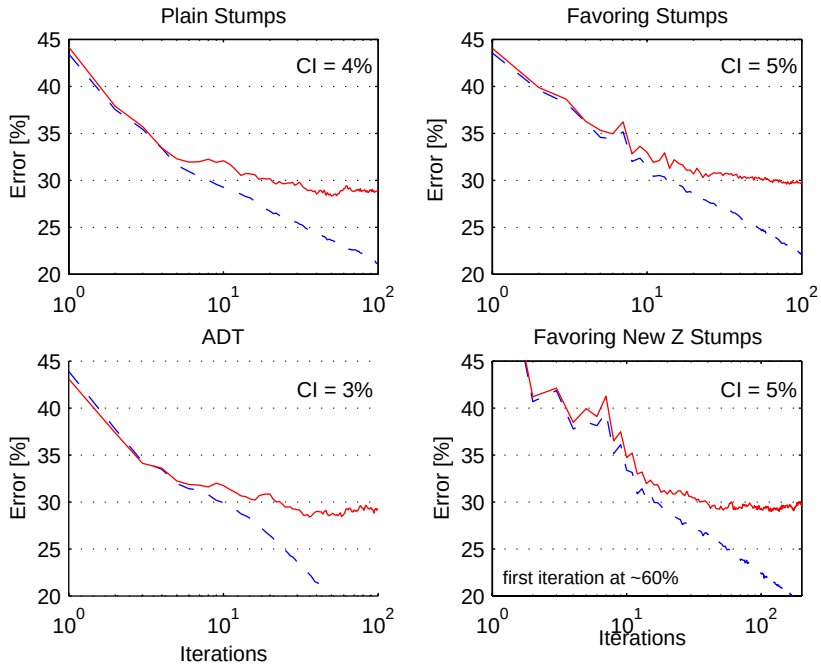

Figure 2-1 : Four different boosting methods applied to the multilabel project data. The dashed line is the training error, and the plain line is the test error, both averaged on 20 trials.

All four methods on figure 2-1 show very similar results. The ADT reaches a minimum generalization error slightly before the plain stumps. We can say that these two methods perform 1% better than the favoring stumps. This is however a tiny difference as compared to the confidence interval of 5%. The favored stumps with the new Z criterion on the lower right graph of figure 2-1 seem to reduce the training error better than with the regular criterion. This leads however to an insignificant generalization improvement. The most striking feature of this lower right graph is the high starting point of the boosting process (60%) and the subsequent instability during the first 10 iterations.

2.1 Attribute global contribution

It is interesting from a data mining point of view to determine on which attributes the predicted labels mostly depend. This allows for example to point out the most important risk factors. A good measure of the attribute relevance is how often in the boosting process the weak learner chooses this attribute, and what its contribution to the final hypothesis is. At a given boosting step, the confidence level of a weak hypothesis, and thus its weight in the final hypothesis is expressed by the amplitude $|h_t(x_i)|$.

Letís consider a given attribute x, and let N be the number of examples in the training set. Let t_x represent the steps where attribute x was chosen. We define the *Absolute Contribution* of x, C_x^{abs} as

$$C_x^{abs} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t_x} |h_t(x_i)|.$$

C_x^{abs} is the magnitude of the weak hypothesis averaged over all examples and summed over all t_x. It happens however that at least for some values of the attribute, the weak hypotheses are contradictory in the voting process, i.e. they have opposite signs. In this case, their effective contribution is lesser, because they donít add up to a large value of $H(x_i)$. Thus C_x^{abs} is overestimated. Therefore we introduce the *Balanced Contribution* of attribute x

$$C_x^{bal} = \frac{1}{N} \sum_{i=1}^{N} \left| \sum_{t_x} h_t(x_i) \right|.$$

This time we first make all the weak hypotheses of t_x vote together, and get the partial final hypothesis based on attribute x. This partial hypothesis is then averaged over the N examples. In the regions where the $h_t(x_i)$ are contradictory, $|\Sigma h_t(x_i)|$ is smaller than $\Sigma|h_t(x_i)|$. Thus $C_x^{bal} \leq C_x^{abs}$, and C_x^{bal} is a more objective measure of the contribution of the attribute x.

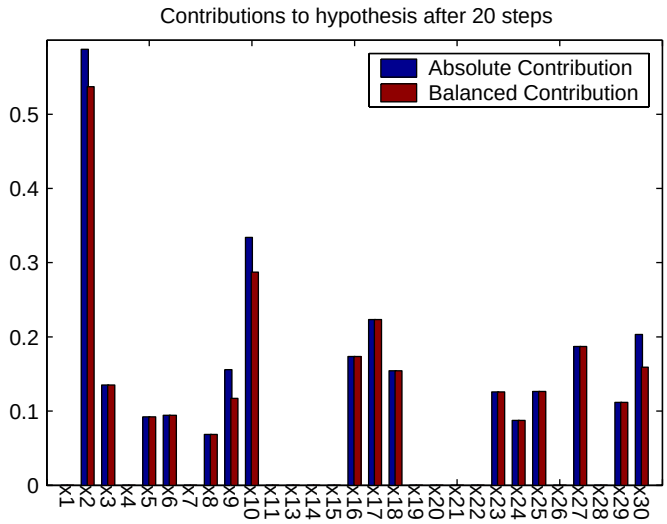

Figure 2.1.1 : Contributions C_x^{abs} and C_x^{bal} after 20 steps of boosting stumps on project data with label `Status`.

Figure 2.1.1 is a bar plot of C_x^{abs} and C_x^{bal} for 20 boosted stumps on the project Data. We see that some attributes donít participate at all in the final hypothesis, and `variable $x_2$` has a heavy weight in the final prediction. Furthermore, C_x^{abs} and C_x^{bal} donít differ much, showing that there is only little contradiction between the weak hypothesis up to step 20.

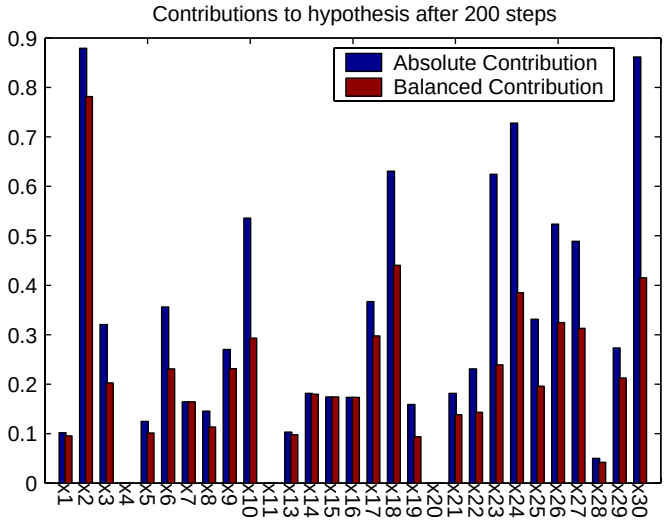

Figure 2.1.2 : Contributions C_x^{abs} and C_x^{bal} after 200 steps of boosting stumps on the project data with label `Status` .

Figure 2.1.2 shows the same plot as figure 2.1.1, but after 200 iterations. Many more attributes have contributed to the final hypothesis. But the balanced contribution C_x^{bal} is often much lower than the absolute contribution C_x^{abs}. This indicates that at least for some regions in the attribute space, the weak hypotheses contradict a lot. This feature is possibly related to overfitting.

2.2 Attribute local contribution

To have a better insight on how the weak hypotheses based on a given attribute interact to contribute to the final hypothesis, we can plot their contributions as a function of the x_i. Figure 2.2.1 is an example of such a plot for the ordinal attribute `x30`. The total contribution of the attribute to the final hypothesis is represented by the black line, and the dots denote the data points. Each weak hypothesis is represented by a colored line. The vertical part indicates the pivot chosen by the weak learner, and the horizontal parts the weights associated to the positive and negative predictions. The number associated to a color curve indicates the boosting step where the weak hypothesis appeared.

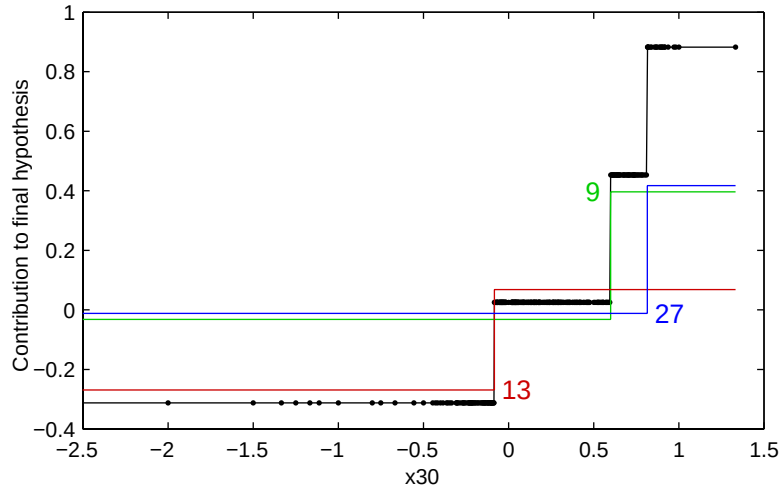

Figure 2.2.1 : Contribution of the ordinal attribute `x30` to the final hypothesis for `Status` after 35 iterations.

The same kind of analysis can be made for nominal variables. Figure 2.2.2 shows an example with the attribute `x2`. The histogram bars show the total contribution to the final hypothesis, and the color segments indicate the participation of each weak hypothesis. The numbers indicate at which boosting step the contribution was added. Note that for every weak hypothesis, only two values are possible, one negative, one positive.

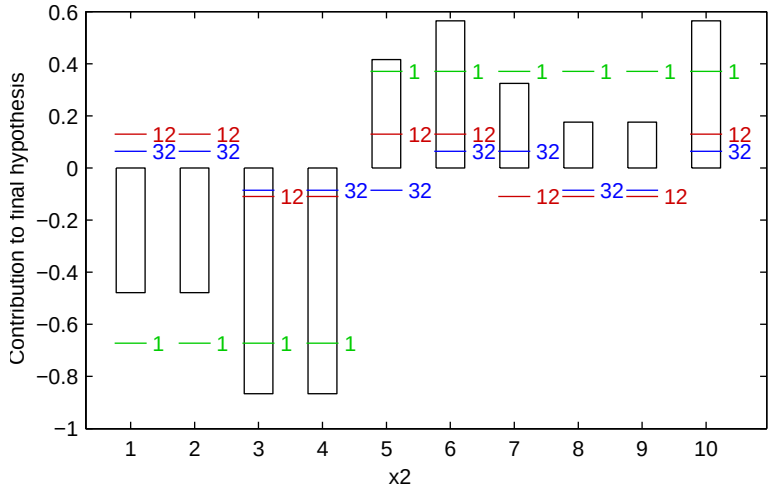

Figure 2.2.2 : Contribution of the nominal attribute `x2` to the final hypothesis for `Status` after 35 iterations

This type of analysis can be made for every attribute by the risk analyst interested in the particular influence of a variable to the labels. It becomes however more difficult to interpret as the number of iterations increases.

3 Monte Carlo Simulation

3.1 Framework

First, we introduce briefly a general description of the key financials of a project. The prefix *AP_* refers to approved or planned values, and the prefix *AC_* to actual values. For example, the planned gross profit is $AP\_GP = AP\_Rev - AP\_Cost$. The planned gross profit margin is then $AP\_GP\% = AP\_GP / AP\_Rev$. We define two ratios relating the approved and actual values: $\alpha = AC\_Rev / AP\_Rev$ and $\beta = AC\_Cos / AP\_Cost$. A last variable is a second order ratio $\delta = \beta/\alpha$.

Given the planned and actual financial performance of a project, one can measure the actual loss (or gain) in project profitability. We call this measure ***Δ GP*** (referred to sometimes as *profit erosion*). Few calculations lead to the following expression for ***Δ GP***:

$$\begin{aligned} \Delta GP &= (AC\_GP\% - AP\_GP\%).AC\_Rev \\ &= a.(1 - \delta).AP\_Cost \end{aligned} \qquad (3.1\text{-}1)$$

This suggests that the actual GP can be rewritten as:

$$AC\_GP = a.\left(\frac{1}{AP\_GP\%} - \delta\right).AP\_COST$$

AC_GP varies as a function of four variables: a, δ, *AP_Cost,* and *AP_GP%.*

For a given project, the business manager has to make decisions about the project cost and GP margin. However, there always remain two sources of uncertainty (a, δ) which could significantly affect the project's outcome. Using predictive data mining techniques for categorization, the goal of our risk management task is to exploit the prior information available on each project and derive its most likely profile. This will allow us to derive more accurate (or homogeneous) distribution information about the factors (a, δ), which can strongly reduce the corresponding uncertainty, and therefore allow much more accurate estimate of actual project loss (gain).

Similarly, one can define the loss (gain) in profit margin as:

$$\begin{aligned} \Delta GP\% &= (AC\_GP\% - AP\_GP\%) \\ &= (1 - \delta).(1 - AP\_GP\%) \end{aligned} \qquad (3.1\text{-}2)$$

Therefore, one can summarize the actual loss (or gain) in a project profitability using the following two expressions:

$$\begin{aligned} \Delta GP &= a.(1 - \delta).AP\_Cost \\ \Delta GP\% &= (1 - \delta).(1 - AP\_GP\%) \end{aligned} \qquad (3.1\text{-}3)$$

Given historical data about a large set of projects, for every project one can compute the above described parameters and losses (gains), which we will denote *Loss(P)*. The second step in the

proposed process is to analyze the whole set of projects and segment it into subgroups $\{G_1, G_2, ..., G_k\}$ where projects in each group share some common characteristics (e.g. quality, health, etc.). In each risk group the risk factors (a, δ) have probability distributions which are estimated from historical data. A natural task then is to characterize membership of each project to any of these groups. This means, on the one hand, finding the most informative project features which allow to discriminate between the different groups, and on the other hand, finding probabilistic decision rules which combine these features and assign group membership probabilities to each project. This is achieved by using a predictive modeling algorithm which is based on *boosted decision trees* as described earlier. This allows us to estimate group membership probabilities $(p_1, p_2, ..., p_k)$ for a any given project *P*.

In the next step of the process, we estimate the expected loss (or gain) of a new project *P*, given its characteristics and its group-membership probabilities $(p_1, p_2, ..., p_k)$.

This is given by:

$$E[Loss(P)] = \sum_{j=1}^{k} Loss(P \mid P \in G_j).p_j \qquad (3.1\text{-}4)$$

where $Loss(P \mid P \in G_j)$ is the expected loss on project *P*, given that it belongs to group G_j. This is defined as:

$$Loss(P \mid P \in G_j) = \int_{G_i} DGP.g_a(a).g_d(\delta)\, da.d\delta$$

where $g_a(a)$ and $g_d(\delta)$ are probability density functions of factors *a* and δ as estimated from data of group G_j.

The formula (3.1-4) can be used to evaluate different pricing scenarios by estimating (using formula (3.1-3)) the expected margin and loss for a given project priced at some planned margin. The estimate could be used by the pricer either in order to adapt the profit margin, or to set up a reserved contingency to mitigate the underlying risk.

The above described analysis is concerned with assessing and pricing the risk of a single project. A natural extension is to extend the analysis to portfolios of projects (see figure (3-1)). This can be of high value in business environments where one cannot afford to take only low risk projects but still wants to ensure some level of profitability; or if one wants to price a project at low margin for competition but still wants to know what should be the margins of other projects in order to keep some average profitability.

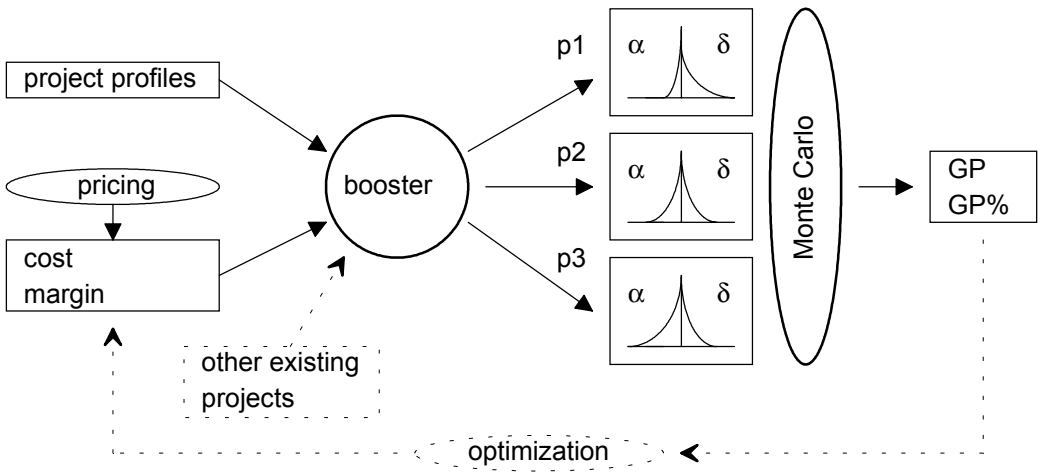

Figure 3-1 : Illustration of project/portfolio assessment and optimization process.

It is therefore more appropriate to design optimal portfolios of projects which have different levels of risk. By doing so, a service provider can be more competitive in the marketplace as she would be able to increase market share by bidding on high risk projects if she can back up the underlying risk with other less risky projects. Therefore, we would like to answer the following question:

*Given an existing portfolio **Q** of ongoing projects* $\{Q_1, Q_2, ..., Q_k\}$*, if one has to make a decision about pricing a new set **P** of projects* $\{P_1, P_2, ..., P_n\}$*, given their estimated risks, what would be the most competitive (acceptable) margins to charge for these new projects which would produce an overall desired profit?*

Using the estimates of individual projects losses (or gains) computed earlier in equation (3.1-4), the overall (cumulative) loss (gain) for the portfolio *(Q, P)* is given by:

$$\begin{aligned} E[Loss] &= E[Loss(Q)] + E[Loss(P)] \\ &= E[Loss(Q)] + \sum_i E[Loss(P_i)] \\ &= E[Loss(Q)] + \sum_i \sum_j Loss(P_i \mid P_i \in G_j).p(P_i \in G_j) \end{aligned}$$

For the ongoing projects ***Q***, *E[Loss(Q)]* is fixed, while *E[Loss(P)]* varies with characteristics of projects $\{P_1, P_2, ..., P_n\}$. Such variation is reflected in the estimates of $Loss(P_i \mid P_i \in G_j)$ and of $p(P_i \in G_j)$.

Therefore, if we want to ensure some profit margin distribution over the overall portfolio, we would need to price the new projects $\{P_1, P_2, ..., P_n\}$ in such a way that their expected margins be distributed according to some desired distribution and the overall loss (resp. gain) be within some bounds with some probability. Therefore we use a Monte Carlo simulation process which uses prior project information to assign project probabilities p_k of membership to each group G_k, and estimates the expected portfolio loss by sampling from each group's (a, δ) distribution proportionally to probabilities p_k.

3.2 The Monte Carlo Algorithm

Input : a project *P* which is described only by demographic features

Initialize :

1. Draw two random numbers from project *AP_Cost* and project *AP_GP%* distributions (if not fixed a priori).
2. Estimate the probabilities $\{p_1, p_2, ..., p_n\}$ of project membership to groups $\{G_1, G_2, ..., G_k\}$ using the predictive model discussed above.

Repeat until stop condition reached :

1. Generate a random integer *i* in the interval [1, *k*] according to probability distribution $\{p_1, p_2, ..., p_n\}$.
2. Generate two random numbers, a_i and δ_i, from distributions of *a* and δ respectively for subgroup G_i.
3. Compute the project loss (*Loss(P)*) using formula (3.1-3).

Output : Draw the distribution of *Loss(P)* and estimate expected values.

We use the above simulation algorithm to estimate the expected loss for a given project *P*. For a portfolio of projects, the same algorithm is used for each single project and the expected portfolio loss is just the sum of the expected project losses.

This algorithm could be extended to estimate the expected cost and margin when only a distribution is assumed for *AP_Cost* and project *AP_GP%*. The algorithm can also be used to optimize the pricing of a project portfolio by finding the approved costs and margins that maximize *Loss(P)*. In both cases, an additional external loop would be added to the algorithm, superposing another layer of Monte Carlo dealing with random *AP_Cost* and *AP_GP%*. However, multiple factors are to be taken into account for the pricing of new projects. Therefore, we keep our simulation as simple as possible, aimed at bringing insights to the question: "Given this pricing strategy, what are the expected outcomes of a project portfolio?".

In the next sections, we analyze the improvements brought to the simulation by the predictive model. The boosted decision tree classifier adds information to the process, by specifying from which distributions α_i and δ_i should be drawn. What is the impact of this on the expected GP and GP%? First of all, we need to be able to measure the similarity between two probability distributions.

3.3 Distance between probability distributions

The Kullback-Leibler divergence is a measure of the "distance" between two distributions. The word "distance" here cannot be taken in its formal meaning, because the Kullback-Leibler divergence is not a symmetric function. In fact, it compares a distribution P to a reference distribution Q. In the case of two discrete distributions with p_i the probability associated with i,

$$K(P,Q) = \sum_i q_i \log\left(\frac{p_i}{q_i}\right).$$

This quantity measures the ratio of distribution P to distribution Q for each i, weighted by the importance of the reference distribution Q at point i and averaged. This weighting is reasonable since it is more important to have the two distribution close to each others in regions were the data is most likely to be found. The Kullback-Leibler divergence is always positive and gives an indication of the uniform discrepancy between two distributions. If we need to assess a relative shift toward one side of the spectrum , we have to use another measure. In the case of normalized probabilities over only three ordinal values $\{1;2;3\}$, the most straightforward signed version of the Kullback-Leibler divergence is

$$\tilde{K}(P,Q) = -q_1 \log\left(\frac{p_1}{q_1}\right) + q_3 \log\left(\frac{p_3}{q_3}\right).$$

This measure is negative if P is shifted towards 1 with respect to Q, and positive if P is more concentrated on 3. The term for $i = 2$ is not taken into account, but the fact that the distributions are normalized compensates for that.

3.4 Shifting the probability distribution

The most straightforward way to perform a project portfolio simulation would be to draw the a and δ from the distributions inferred from the whole sample. A little bit more elaborate, we can use the fact that different project groups (for illustration, here we assume three groups : $1,2,3$) have different distributions for a and δ , but without having any information about the group to which a particular project actually belongs. We then choose at random a group, and hence a specific distribution for a and δ, according to algorithm of section 3.2. In this case the $\{p_1, p_2, p_3\}$ are the so-called *a priori* probabilities, which are estimated uniformly for the whole sample as the frequency of each class. . This a priori probability distribution is a good reference point to assess the relevance of the information brought by the predictive model.

In this section, we assess the dependence of the Monte Carlo simulation on the class probability distribution, independently from the predictive model. Instead of using the genuine information output by the boosted stumps as we will do in next section, we vary artificially the probabilities. Based the a priori distribution $\{p_1, p_2, p_3\}$, we build shifted distributions uniform for all projects in the following way. Let x be a parameter lying in $[-0.9, 0.9]$. The shifted distribution is computed as

$$\frac{1}{Norm}\{p_1 + xp_1\ ,\ p_2\ ,\ p_3 - xp_3\}.$$

The factor $Norm$ insures a normalized probability distribution after the transformation. For negative x the distribution is shifted towards 1. Thus the signed Kullback-Leibler divergence $\tilde{K}$ with respect to the a priori distribution is negative. For different values of x, we compute the expected GP and GP%. and report them in figure (3-2) as a function of $\tilde{K}$.

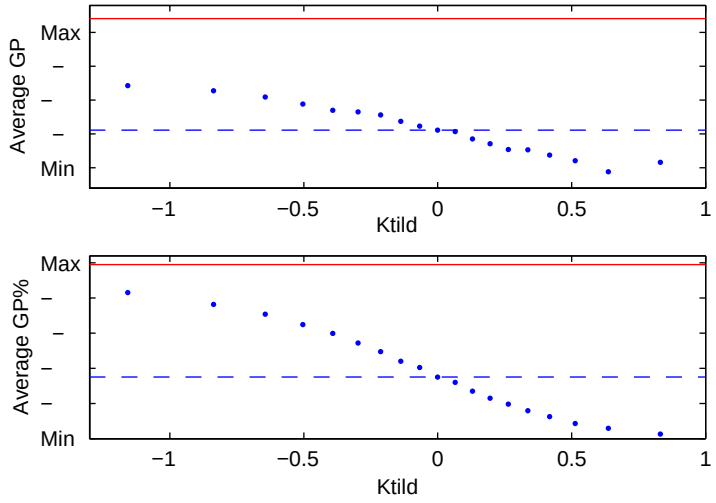

Figure 3-2: Influence of a shift in the group probabilities on the mean GP and GP%. The dashed line represents the prediction using the a priori probability distribution. The plain line shows the approved GP and GP%.

The upper graph shows the expected GP averaged on all projects from the database. The dashed line shows as a reference the prediction using the a priori probability distribution ($\tilde{K} = 0$). The plain line represents the approved GP and GP%, for comparison. For negative values of $\tilde{K}$ the probability distribution is shifted towards `group 1`. This bias results in higher values of both GP and GP%. The opposite is also true, which proves the consistency of the simulation.

3.5 Prior vs. predicted class distribution

Instead of varying the class probabilities uniformly and artificially, we now use the classifier model to associate to every project its own probability distribution over different groups depending on all its attributes. To find the probabilities, we run AdaBoost with stumps, and get class probabilities. Unlike what we did in section (2), we want to use only attributes available when the project is priced, and hence before it even starts. We focus on a priori attributes in the training set, excluding all actual financials and all appreciations made on ongoing or closed projects. Of course, the accuracy of the prediction suffers from this restriction, it is seen in figure (3-3).

In addition to plain stumps, we test the favoring stumps to see if a significant effect on the probabilities is observed. The favoring stumps exhibit strong oscillations in the first iterations as observed in (Cuendet and Labbi, 2001), but after 100, they reach a generalization error similar to the plain stumps. Additional tests not reported here showed that there is no significant improvement at further iterations.

The best classification error achieved here is lower than the one achieved when considering additional attributes, but again here the classification error is of lower importance, because we are interested in the overall information gained in the Monte-Carlo simulation.

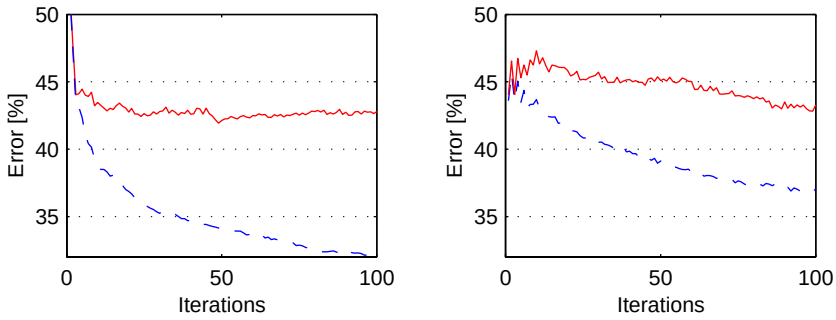

Figure 3-3 : `Group` classification error on 100 iterations of plain stumps (left) and favoring stumps (right). The attributes used here are only a priori.

For every project, one can measure the Kullback-Leibler divergence K and the signed Kullback-Leibler divergence $\tilde{K}$ between the predicted probabilities and the reference a priori probabilities. These measures are compared with the *GP_Shift* or the *GP%_shift* defined as

$$GP\_Shift = E\left[GP \mid \left\{p_j^{\text{model}}\right\}\right] - E\left[GP \mid \left\{p_j^{\text{a priori}}\right\}\right]$$

$$GP\%\_Shift = E\left[GP\% \mid \left\{p_j^{\text{model}}\right\}\right] - E\left[GP\% \mid \left\{p_j^{\text{a priori}}\right\}\right]$$

The plots on figure (3-4) show the variation of \$GP\$ and \$GP\\%\$ as functions of $\tilde{K}$. Let's recall that $\tilde{K}$ is negative when the tested distribution is shifted towards the low values with respect to the reference distribution.

In the upper graph, we show the relationship between the *GP_Shift* and $\tilde{K}$. A difficulty arises because of the very large range covered by *GP_Shift* both in the negative and positive values. To overcome this, we represent $\text{sign}(GP\_Shift) \cdot \log(|GP\_Shift| + 1)$. The drawback is that the region around zero is artificially depleted of points on the graph. But clearly, the negative values of $\tilde{K}$ induce a positive shift of GP. This is consistent with the intuition that a project with good health will generate better financial outcomes than average.

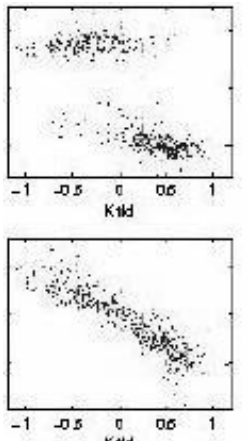

Figure 3-4 : Shifts in expected GP and GP% as functions of the signed Kullback-Leibler divergence $\tilde{K}$ between the predicted and a priori probabilities. The predictor is boosted plain stumps.

To conclude the analysis of figure (3-4), we focus on the bottom graph, which shows a strong correlation between $\tilde{K}$ and *GP%_Shift*. According to this, the information brought by the predictive model allows to fine-tune the estimation of the margins depending on the likelihood of project groups.

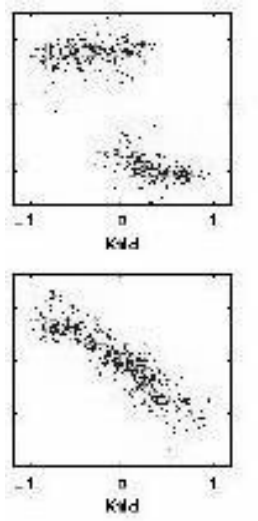

Figure 3-5 : Same plots as in figure 3-4 obtained with favoring stumps.

The results on figure 3-5 are obtained with favoring stumps. All graphs look essentially the same as figure 3-4. There seem to be more negative values of $\tilde{K}$ in the favored case. This means that class 1 is more strongly predicted. This is consistent with the goal of favoring boosting. Under these changes, the relationship between $\tilde{K}$ and the *GP%_Shift* (bottom graph) seem to be a bit tighter than with standard stumps.

4 Conclusion

As a conclusion we summarize the experimental results obtained using boosting predictors to estimate a priori class probabilities and use such distribution in the Monte Carlo simulation process in order to make a more precise estimate of project cost and margin.

On the experimental side, the derived boosted stumps are used to estimate and rank project risk factors according to their influence on different class likelihoods. The dependence of project class on the specific values of an attribute (or risk factor) of interest can be easily visualized.

Concerning Monte Carlo simulation for project cost and margin (pricing) estimation, we showed the feasibility of the simulation based on different project classes. First we proved that shifting the class probabilities of the projects changes the expected margin erosions (gains). Then we showed that information about the risk group is efficiently brought by the boosted decision trees. With this fine tuning of the simulation, the project pricer can take advantage of all the information at hand to get the best possible forecast of the financial outcomes of a project portfolio.

5 References

[1] Allwein E. L., Schapire R. E., Singer Y., *Reducing multiclass to binary: A unifying approach for margin classifiers*, Journal of Machine Learning Research, 1:113-141, 2000.

[2] Breiman L, Friedman J.H., Olshen R.A, and Stone, C.J.. *CART: Classification and Regression Trees*. Wadsworth: Belmont, CA, 1983.

[3] Cuendet M, Labbi, A. *Theoretical and Practical Perspectives on Boosting Weak Predicters*. IBM Research Tech. Report, RZ 5656566, November, 2001.

[4] Freund Y., Schapire R. E., *A short introduction to boosting*, Journal of Japanese Society for Artificial Intelligence, 14(5):771-780, 1999.

[5] Freund Y., Schapire R. E., *A brief introduction to boosting*, Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, 1999.

[6] Freund Y., Schapire R. E., *A decision-theoretic generalization of on-line learning and an application to boosting*, Journal of Computer and System Sciences, 55(1):119-139, 1997.

[7] Freund Y., Schapire R. E., *Experiments with a new boosting algorithm*, Machine Learning: Proceedings of the Thirteenth International Conference, pages 148-156, 1996.

[8] Freund Y., Schapire R. E., *Adaptive game playing using multiplicative weights, Games and Economic Behavior*, 29:79-103, 1999.

[9] Freund Y., Mason L., *The Alternating Decision Tree Learning Algorithm*, ICML-99, 1999.

[10] Friedman, J. H., Hastie T., Tibshirani R., *Additive Logistic Regression: a Statistical View of Boosting*, Technical report, Department of Statistics, Stanford University, 1998.

[11] Holte R. C., *Very Simple Classification Rules Perform Well on Most Commonly Used Datasets*, Machine Learning, 3 : 63-91, 1993.

[12] Iba W., Langley P., *Induction of one-level decision trees*, Proceedings of the Ninth International Machine Learning Conference. Aberdeen, Scotland: Morgan Kaufmann, 1992.

[13] Mitchell R. A., *Boosting Stumps from Positive Only Data*, Technical Report UNSW-CSE-TR-9907, 1999.

[14] Quinlan R., *MiniBoosting Decision Trees*, Journal of Artificial Intelligence Research, 1998.

[15] Schapire R. E., Singer Y., *Improved boosting algorithms using confidence-rated predictions*, Machine Learning, 37(3):297-336, 1999.

[16] Merz, C. J., Murphy, P. M., *UCI repository of machine learning databases*. http://www.ics.uci.edu/~mlearn/MLRepository.html.