Tree-based Methods for Data Mining

Paula Iborra, Marcel Pons

March 28, 2021

1 Introduction

Among all the well known learning methods, decision trees come closest to satisfying the requirements of an off-the-shelf procedure for data mining. They are relatively fast to construct and they produce simple and useful results for interpretation. They naturally incorporate mixtures of numeric and categorical predictor variables and missing values, and they are invariant under transformations of individual predictors. Hence, scaling and other more general transformations are not a problem, and they are not affected by outliers in the predicted value. A basic aspect of decision trees is the performance of internal feature selection as an integral part of the process. Thereby, even if they are not completely immune, they are resistant to the inclusion of many irrelevant predictor variables. These properties of decision trees are largely the reason why they have become the most popular learning method for data mining.

However, decision trees typically are not competitive with the best supervised learning approaches in terms of prediction accuracy. Inaccuracy is then one condition that prevents them from being an ideal tool for predictive learning. They rarely provide the best prediction accuracy comparable to the best that can be achieved with the data at hand.

Hence we will introduce some methods that can greatly improve decision trees' accuracy. Each of these techniques involves generating multiple trees and then combining them to produce a single consensus prediction. As we will see, merging a large number of trees can often significantly improve prediction accuracy, but at the expense of some interpretability loss.

2 Decision Trees

Decision Trees are characterized by the easy interpretation and intuition of the decisions they make in order to do predictions. A Decision Tree is a data-structure consisting of a hierarchy of individual units called nodes, which are grown recursively depending on the state of its predecessor nodes.

The *root* is the node at which the tree starts growing, where the initial *if-else* question about an individual feature will be made in order to split the labeled data. The tree continues growing recursively in the same way by means of *decision* or *internal nodes*, which will form sub-trees, until all the data is split according to the response variable. The terminal nodes which have no parents are called *leaves* and will contain the final predictions.

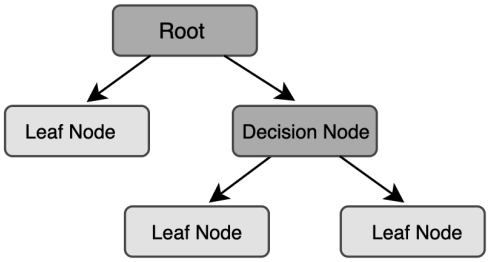

Figure 1: Decision Tree schematic representation

There are several algorithms used to build decision trees. Here the CART algorithm will be considered, since it is the most popular and widely used, both for machine learning models and also as weak learners for many ensemble methods.

2.1 Classification and Regression Trees

Split Criteria

On the root and on every internal node, the CART algorithm works by splitting the data into two subsets using a single feature k and a threshold t_k, which maximizes the information gain[1] after each split.

This information gain is measured differently between regression and classification trees.

Classification trees search for the splits that produce the minimum impurity, which is a measure of the homogeneity of the labels on the node. The purest nodes are the ones that contain only one class after the split. The impurity of a node can be calculated either by the *Gini* index (G_i) or by the *Information Entropy* (H_i):

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{1}$$

$$H_i = -\sum_{k=1}^{n} p_{i,k} log_2(p_{i,k}) \tag{2}$$

where $p_{i,k}$ is the ratio of class k instances among the observations in the i^{th} node. Both measures are weighted by the size of observations that are included in the node (n_m). Being M the total number of observations in the data sample, the algorithm searches for the pair of child nodes that produce the purest subsets:

[1] Decreased measurement when the data set gets split and there is a reduction in entropy (degree of randomness in the data set).

$$S(k, t_k) = \min_{k,t_k} \Big(\frac{m_{left}}{M} \cdot G_{left} + \frac{m_{right}}{M} \cdot G_{right} \Big) \tag{3}$$

For regression trees, instead of minimizing impurity, the algorithm searches for the split that best minimizes the Mean Squared Error (MSE) of the pair of child nodes:

$$S(k, t_k) = \min_{k,t_k} \Big(\frac{m_{left}}{M} \cdot MSE_{left} + \frac{m_{right}}{M} \cdot MSE_{right} \Big) \tag{4}$$

where $MSE_{node} = \sum_{i \in node} (\hat{y}_{node} - y^{(i)})^2$, being the predicted value $\hat{y}_{node}$ the average value of the instances falling in that node.

Tree Pruning

If a tree is left unconstrained, the tree structure will adapt to the data given as input, fitting it very closely and therefore increasing the variance of the model (i.e., being overfit).

For this reason, an optimal tree which can explain the unseen data as best as possible has to be found. Many implementation of the CART algorithm define a hyperparameter `max_depth` which restricts the tree from growing further than a specified depth. On the other hand, another way of building optimal trees is to use *Cost complexity pruning*, which selectively collapses certain parts of the tree in order to minimize a cost complexity function $C_\alpha(T)$.

$$C_\alpha(T) = R(T) + \alpha |T| \tag{5}$$

$C_\alpha(T)$ quantifies the capacity of a tree to predict accurately the data in terms of missclassification rate in classification and MSE in regression trees, expressed as $R(T)$. The more terminal nodes a tree has ($|T|$), the larger is the tree, and consequently the more it adapts to the data, therefore a complexity parameter α is taken into consideration in order to penalize large trees.

Either the optimal `max_depth` or α are found by cross-validation, obtaining a tree that adapts well to the training data without overfitting, thus being able to predict new data in a proper way.

3 Bagging and Boosting

3.1 Bagging

Decision trees discussed in the previous section are very sensitive to the data they are trained, changes in the training data may cause the tree structure to

be significantly different, and suffer from high variance. This means that if we randomly split the training data into two parts and fit the decision tree into both halves, the results we get may be very different. However, a procedure with low variance will yield similar results if applied repeatedly to distinct data sets.

Bagging (or bootstrap aggregation) is a general process to diminish the variance in statistical learning methods particularly useful and frequently used in the context of decision trees.

Recall that given a set of n independent observations $Z^1, ..., Z^n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by $\frac{\sigma^2}{n}$. That is to say, averaging a set of observations reduces variance. Therefore, the natural way to reduce variance and hence improve the prediction accuracy of statistical learning methods is to obtain many training sets from the population, use each training set to build a separate prediction model, and average the obtained predictions. Thereby, we can use B separate training sets to calculate $\hat{f}^1(x), \hat{f}^2(x), ..., \hat{f}^B(x)$, and average them to obtain a low-variance statistical learning model, given by

$$\hat{f}_{avg}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^b(x) \tag{6}$$

However, this is not practical since in general we do not have access to multiple training sets. Instead, we can *bootstrap*[2] by repeatedly sampling from a single training data set. Following this method, we generate B different bootstrap training data sets. Then, we train our model on the b^{th} bootstrapped training set to get $\hat{f}^{*b}(x)$, and finally average all predictions. The bagging estimate is defined by

$$\hat{f}_{bag}(x) = \frac{1}{B} \sum_{b=1}^{B} \hat{f}^{*b}(x) \tag{7}$$

When *bagging* is applied to the regression trees, we simply use B bootstrapped training sets to construct B regression trees, and average the obtained predictions. These trees are grown deep, and are not pruned. Therefore, each individual tree has a high variance, but a low bias.

Since each tree generated in bagging is identically distributed (i.d.), the expectation of an average of B such trees is the same as the expectation of any one

[2]Re-sampling technique that allows to emulate the process of obtaining new sample sets. Distinct data sets are obtained by repeatedly sampling observations from the original data set. Given a data set Z we randomly select n observations in order to produce a bootstrap data set, Z^{*1}. The sampling is performed with replacement, meaning that the same observation can occur more than once in Z^{*1}. The point of replacement is to make the re-sampling truly random. Otherwise, the samples drawn will be dependent on the previous ones. This procedure is repeated B times for some large value of B, in order to produce B different bootstrap data sets, $Z^{*1}, ..., Z^{*B}$.

of them. This means the bias of bagged trees is the same as that of the individual trees, and the only hope of improvement is through variance reduction. Averaging these B trees reduces the variance.

This technique can be extended to classification problems where the outcome is qualitative. In this case, there are several possible methods, but the simplest is as follows. For a given test observation, we can record the category predicted by each B-tree and take a majority vote: the overall prediction is the most common category among the B predictions.

Bagging has proved that, by combining hundreds or even thousands of trees into a single consensus prediction, accuracy can significantly be improved.

3.2 Boosting

Boosting is an ensemble method for improving the model predictions of any given learning algorithm that can be applied to many statistical learning methods for regression or classification. The term *boosting* refers to a family of algorithms which combines the outputs of many "weak"[3] learners to produce a powerful strong learner. The idea of boosting is to sequentially train weak learners, each trying to correct its predecessor, to repeatedly modified versions of the data, thereby producing a sequence of weak classifiers whose predictions are then combined to produce the final prediction.

Like *bagging*, *boosting* involves combining a large number of decision trees, where trees are grown sequentially in an adaptive way to remove bias, and hence are not i.d.; each tree is grown using information from previously grown trees. Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set.

One of the most popular boosting algorithm is called *AdaBoost.M1*, presented by Freund and Schapire (1997) [2].

Here we briefly describe the procedure of this algorithm. Considering a two-class problem, with the output variable coded as $Y \in \{-1, 1\}$, given a vector of predictor variables X and a classifier $G(X)$, *AdaBoost* sequentially apply the weak classification algorithm to repeatedly modified versions of the data. It produces a sequence of weak classifiers $G_m(x), m = 1, 2...M$. These weak learners in *AdaBoost* are decision trees with a single split, called decision *stumps*. Then, through a weighted majority vote, the prediction results are combined to arrive at the final prediction result:

$$G_m(x) = sign\left(\sum_{m=1}^{M} \alpha_m G_m(x)\right) \tag{8}$$

[3] A weak classifier is one whose error rate is only slightly better than random guessing.

Here $\alpha_1, \alpha_2 ... \alpha_m$ are computed by the boosting algorithm and weight the contribution of each respective $Gm(x)$. Their effect is to give higher influence to the more accurate classifiers in the sequence.

Data modifications at each boosting step consist of applying weights $w_1, w_2, ..., w_n$ to each of the training observations $(x_i, y_i), i = 1, 2, ..., N.$, where y_i is the label associated with instance x_i.

At the outset, all of the weights are initialized to $w_i = \frac{1}{N}$, so that the first step is simply to train the classifier on the data in the usual manner. For each successive iteration $m = 2, 3, ..., M$, the observation weights are individually modified and the classification algorithm is reapplied to the weighted observations. At step m, to correct the previous error, those observations that were misclassified at the prior iteration by the classifier $G_{m1}(x)$ have their weights increased, whereas those that were classified correctly have their weights decreased. Consequently, as we proceed in the iterations, observations that are difficult to classify correctly receive ever-increasing influence. Each successive classifier is thereby forced to concentrate on those training observations that are missed by previous ones in the sequence.

AdaBoost algorithms can be adapted and used for both classification and regression problems.

4 Random Forests

Random forest is a tree-based algorithm composed of many decision trees where their outputs are combined to enhance the performance of a given model. It uses bagging and feature randomness when building each individual tree to try to create an uncorrelated forest of trees whose prediction by committee is more accurate than that of any individual tree. Recall that an average of B i.i.d. (independent i.d.) random variables, each with variance σ^2, has variance $\frac{1}{B}\sigma^2$. If the variables are simply i.d. (but not necessarily independent) with positive pairwise correlation ρ, the variance of the average is:

$$\rho\sigma^2 + \frac{1-\rho}{B}\sigma^2 \tag{9}$$

Notice that as B increases the second term disappears, but the first remains, and hence the size of the correlation of pairs of bagged trees limits the benefits of averaging. The idea in random forests is to improve the variance reduction of bagging by reducing the correlation between the trees. This is achieved by randomly selecting input variables during the tree-growing process. Thereby, when growing a tree on bootstrapped training samples, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. A new set of m predictors is considered at each split, and usually we choose $m \approx \sqrt{p}$, that is, the number of predictors considered

at each split is approximately equal to the square root of the total number of predictors.

That is to say, at each split in the tree, the algorithm is not allowed to consider a majority of the available predictors. This is to avoid cases where a very strong predictor in the data set is used in the top split for most or all the trees leading to similar bagged trees. Hence the predictions from the bagged trees will be highly correlated. Unfortunately, averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities. As mention, random forests overcome this problem by forcing each split to consider only a subset of the predictors.

The main difference between bagging and random forests is the choice of predictor subset size m. For instance, if a random forest is built using $m = p$, then this is equivalent to bagging.

A pseudo-code implementation for the Random Forest algorithm proposed by Jerome Friedman *et al.* in *The Elements of Statistical Learning* [5] for both classification and regression is shown below.

---

1. For $b = 1$ to B:
   (a) Draw a bootstrap sample $\mathbf{Z}^*$ of size N from the training data.
   (b) Grow a random-forest tree T_b to the bootstrapped data, by recursively repeating the following steps for each terminal node of the tree, until the minimum node size n_{min} is reached.
      i. Select m variables at random from the p variables.
      ii. Pick the best variable/split-point among the m.
      iii. Split the node into two daughter nodes.
2. Output the ensemble of trees $\{T_b\}_1^B$.

To make a prediction at a new point x:

Regression: $\hat{f}_{\text{rf}}^B(x) = \frac{1}{B}\sum_{b=1}^{B} T_b(x)$.

Classification: Let $\hat{C}_b(x)$ be the class prediction of the bth random-forest tree. Then $\hat{C}_{\text{rf}}^B(x) = \textit{majority vote}\ \{\hat{C}_b(x)\}_1^B$.

---

5 Gradient Boosting

Gradient Boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak learners, which are typically trees. It builds the models in a stage-wise way and it generalizes them by allowing optimization of an arbitrary differentiable Loss Function ($L(y_i, \gamma)$).

The Gradient Boosting algorithm consists on iteratively assembling models that

are built from the residuals generated by the previous iteration instead of the values of the response variable. The way in which regression and classification work differs in the Loss Function $L(y_i, \gamma)$ used to obtain the residuals used to build the trees. For this reason, the algorithm proposed by Jerome Friedman *et al.* in *The Elements of Statistical Learning* [5] can be used to briefly explain the two applications of supervised learning, stating which are the Loss Functions that are mostly used.

---

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^{N} L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute
   $$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$
   (b) Fit a regression tree to the targets r_{im} giving terminal regions $R_{jm},\ j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute
   $$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$
   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

The **first step** of the algorithm is to initialize the model with a constant $f_o(x)$, which is just a single terminal node tree. This node (or leaf) takes as input the training dataset $\{(x_i, y_i)\}_{i=1}^{n}$ and finds an initial prediction for the response variable of all the samples (γ) that minimizes the Loss Function.

The **second step** of the algorithm consists on the process of iteratively building the trees (m_i), where the final model will have a total of M trees[4]. The construction of each trees is divided in four stages:

(a) Consists on computing the residuals $r_{i,m}$ of the predictions where i is the sample number and m is the tree that is being built. The computation consists on the derivative of the Loss Function with respect to the Predicted value (i.e., the gradient) of the previous tree $F(x) = F_{m-1}(x)$.

(b) Consists on fitting the decision tree to the $r_{i,m}$ values previously computed. The trees are built like a typical decision tree but instead of predicting y values they predict residuals. Besides, each leaf (terminal region) has assigned a label R_{jm}.

[4]The number of leaves the trees have is a parameter that has to be optimized. Usually Gradient Boosting models use trees with between 8 to 32 leaves.

(c) Consists on determining the output values for each leaf R_{jm} of the tree by finding the predicted value (γ) that minimizes the Loss Function for all the values that fall in the leaf ($x_i \in R_{ij}$). This stage is similar to the *Step 1* but taking into account the prediction made by the previous tree $F_{m-1}(x_i)$ when calculating the Loss Function.

(d) Consists on making new predictions, which are calculated by summing the prediction of the residuals $r_{i,m}$ for each sample to the predicted values of the previous tree $F_{m-1}(x_i)$. The predicted values of the residuals are usually penalized by a Learning Rate ν (where $0 < \nu \leq 1$) that reduces the effect each tree has on the final prediction, improving accuracy in the long run and avoiding overffiting. These new predictions will be used to calculate the residuals $r_{i,m}$ used to construct the next tree $(m + 1)$.

Once all M trees are built, the **third step** consists on making the final predictions $\hat{f}(x_i)$ for each value of the training sample.

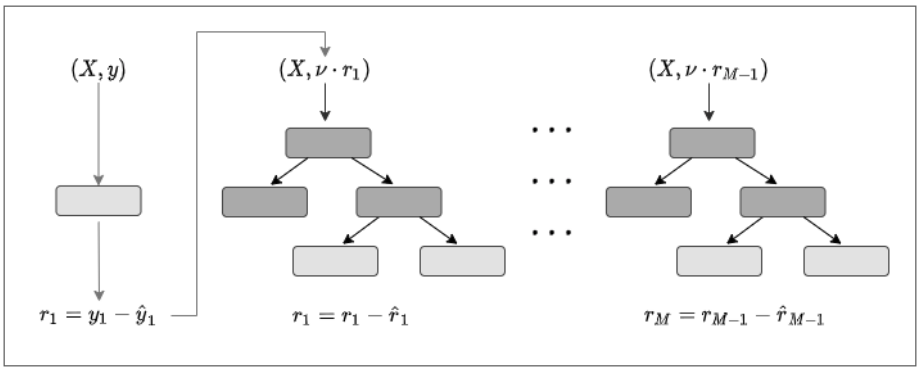

Figure 2: Visual representation of the Gradient Boosting algorithm, which is initialized by a single leaf and further extended by the construction of decision trees using the residuals of the previous iteration r_i.

5.1 Loss Functions

As mentioned previously, the election of Loss Function to calculate the residuals determines whether Gradient Boosting is used for regression or classification. For each technique, one Loss Function is usually used among the others.

For **continuous** response variables, a variant version of the Mean Square Error $\frac{1}{2}(y_i - \hat{y}_i)^2$ is mostly used because it eases the derivatives necessary to find γ that minimizes the error, therefore being computationally more efficient.

For **categorical** response variables, the Loss Function that is mostly used is the negative $log(likelihood)$.

5.2 Stochastic Gradient Boosting

In situations where the size of the input data is considerably large, incorporating randomization into the Gradient Boosting procedure can improve its performance and execution speed. Stochastic Gradient Boosting works by drawing a subsample (without replacement) at each iteration m from the full training data. This subsample is used to build the m tree and determine the output values for each leaf R_{jm}, consequently playing a role in the final predictions[5]. By means of this implementation, a higher bias is obtained at expanses of a lower variance.

6 Extreme Gradient Boosting (XGBoost)

Extreme Gradient Boosting is an optimized implementation of Gradient Boosting aimed to be extremely fast, perform upon huge amounts of data and having a high scalability in all scenarios.

In XGBoost, the construction of the tree ensamble model is done in a similar way as in Gradient Boosting, mainly obtaining the residuals from the previously predicted values by minimizing a Loss Function and using a penalty (called η) to smoothen the final prediction.

6.1 XGBoost Trees

One of the novelties of XGBoost is that the individual trees are built in a slightly different way, which involves a *Similarity score* computed after each split and an *information gain* based on these scores. The computation of the Similarity scores, as well as the output values of the final tree are derived from:

$$\left[\sum_{i=1}^{n} L(y_i, \hat{y}_i^{(t-1)} + f_t(x_i))\right] + \gamma|T| + \frac{1}{2}\lambda f_t(x_i)^2 \tag{10}$$

Where the Loss Function L gets the actual values y_i and the predicted values of the previous tree plus the new output values $f_t(x_i)$. $\gamma|T|$ is a penalty meant to encourage pruning and the third part of the equation consists in a regularization to shrinkage the prediction's sensitivity to individual observations.

By means of Taylor Series, gradients g and hessians h, the Similarity scores and output values are extracted from (8), where:

$$f_t(x) = \frac{-(g_1 + g_2 + \cdots + g_n)}{(h_1 + h_2 + \cdots + h_n + \lambda)}$$

[5]This approach also increases robustness against overcapacity of the weak learners.

$$Similarity\ Score = \frac{-(g_1 + g_2 + \cdots + g_n)^2}{(h_1 + h_2 + \cdots + h_n + \lambda)}$$

The *Information Gain* used to asses whether a split leads to better prediction performance is defined as:

$$Gain = Sim_{t_{left}} + Sim_{t_{right}} - Sim_t \tag{11}$$

The value of *Gain* is compared with the predefined[6] value of γ. If $(Gain - \gamma) < 0$ the leaves will be pruned. The regularization parameter λ makes the leaves easier to be pruned since it reduces the similarity scores and consequently the *Information Gain*.

6.2 Optimizations

Extreme Gradient Boosting includes several optimizations in order to speed up the computations made and make it characteristic for its scalability and capacity to handle large datasets.

Approximate Greedy Algorithm

When dealing with huge amounts of data, it is infeasible to search for the best split over all the possible thresholds on all the features. In general, all tree based models follow an *Exact Greedy* procedure, making a decision on the thresholds without looking ahead to see whether it is the absolute best choice in the long term. However, with lots of features and values the *Exact Greedy* procedure can become slow due to it has to assess every possible threshold (in every single feature). Therefore, in order to do it efficiently, the XGBoost algorithm proposes candidate splitting points according to quantiles of feature distribution.

For the purpose of approximately finding the best quantiles, the algorithm uses two additional optimizations: **Parallel Learning** and a **Weighted Quantile Sketch**. Essentially, these two optimizations consist on splitting the data so that multiple computers can work on it in parallel and then combining the values from each computer to make an histogram that is used to calculate approximate quantiles.

Sparsity-Aware Finding

By means of the Sparsity-Aware Finding optimization, XGBoost can build trees when the data is sparse and eventually deal with new observations with missing data. That is done by adding a default direction (learned from the data) in each tree node, which is followed in case a value is missing.

[6]The default value for γ is 0, which does not turn off pruning since there can be negative *Gain* values.

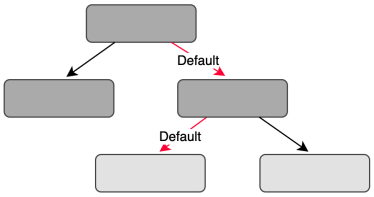

Figure 3: Default directions on a sparsity-aware implementation

System Design

System design optimizations consider the computer hardware and how the data is stored in memory.

- **Cache-Aware Access** consists on putting the gradients g and hessians h in the cache so that the algorithm can rapidly calculate Similarity Scores and Output values.
- **Blocks for Out-of-Core Computation** consists on compressing the data into *blocks* when the cache and main memory are not available and the hard drive is used. In this way, the algorithm minimizes the time spend reading and writing data from the hard drive.

7 References

[1] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[2] Yoav Freund and Robert E Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[3] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[4] Jerome H Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.

[5] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[6] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.