INSAID

Chapter 9

Why Does XGBoost Win "Every" Competition?

In this chapter, we will provide some informal arguments for why tree boosting, and especially XGBoost, performs so well in practice. Of course, for any specific data set, any method may dominate others. We will thus provide arguments as to why tree boosting seems to be such a versatile and adaptive approach, yielding good results for a wide array of problems, and not for why it is necessarily better than any other method for any specific problem.

Uncovering some possible reasons for why tree boosting is so effective are interesting for a number of reasons. First, it might improve understanding of the inner workings of tree boosting methods. Second, it can possibly aid in further development and improvement of the current tree boosting methodology. Third, by understanding the core principles of tree boosting which makes it so versatile, we might be able to construct whole new learning methods which incorporates the same core principles.

9.1 Boosting With Tree Stumps in One Dimension

Consider first the very simple case of boosting in one dimension, i.e. with only one predictor, using tree stumps, i.e. trees with only one split and two terminal nodes. We otherwise assume the model in unconstrained and unpenalized.

Does tree boosting have any particular advantages even in this simple problem? First of all, additive tree models have rich representational abilities. Although they are constrained to be piecewise constant functions, they can potentially approximate any function arbitrarily close given that enough trees are included.

The perhaps greatest benefit however is that tree boosting can be seen to adaptively determine the local neighbourhoods. We will now consider an example to clarify what we mean by this.

9.1.1 Adaptive Neighbourhoods

Let us consider a regression task and compare additive tree models with local linear regression and smoothing splines. All of these methods have regularization parameters which allows one to adjust the flexibility of the fit to the data. However, while methods such as local regression and smoothing splines can be seen to use the same amount of flexibility in the whole input space of $\mathcal{X}$, additive tree models can be seen to adjust the amount of flexibility locally in $\mathcal{X}$ to the amount which seems necessary. That is, additive tree models can be seen to adaptively determine the size of the local neighbourhoods.

Let us consider a concrete example. To make the point clear, we will design a problem where different amounts of flexibility is needed locally in $\mathcal{X}$. Let the data be generated according to

$$\begin{aligned} X &\sim \text{Uniform}(-6, 6) \\ [Y|X] &= \begin{cases} \sin(X) + \epsilon, & X \leq 0 \\ \epsilon, & 0 < X \leq \pi \\ \sin(4X) + \epsilon, & X > \pi \end{cases} \\ \epsilon &\sim N(0, 1/2). \end{aligned}$$

We will draw 500 samples from this. These samples together with the target function f^* is shown in Figure 9.1.

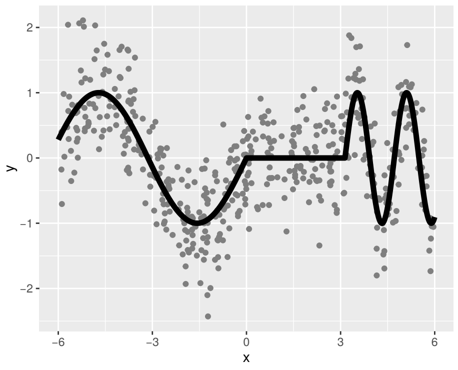


Figure 9.1: Simulated data in gray together with the target function f^* in black.

This data clearly requires a high degree of flexibility for higher values of x, less flexibility for low values of x and almost no flexibility at all for medium values of x. That is, for higher values of x, smaller neighbourhoods are needed to avoid a large bias. Keeping the neighbourhoods too small in the flat region of the target function, however, will unnecessarily increase variance.

Let us first fit local linear regression with two different degrees of flexibility to the data. The resulting fits are shown in blue in Figure 9.2. In Figure 9.2a a wider neighbourhood is used for the local regression. This seems to yield a satisfactory fit for lower values of x, but is not flexible enough to capture the more complex structure for the higher values of x. Vice versa, in Figure 9.2b, a smaller neighbourhood is used. This fit is flexible enough to capture the complex structure for the higher values of x, but seems too flexible for lower values of x. The same phenomenon is observed for smoothing splines. Figure 9.3 shows two smoothing spline fits of different flexibility.

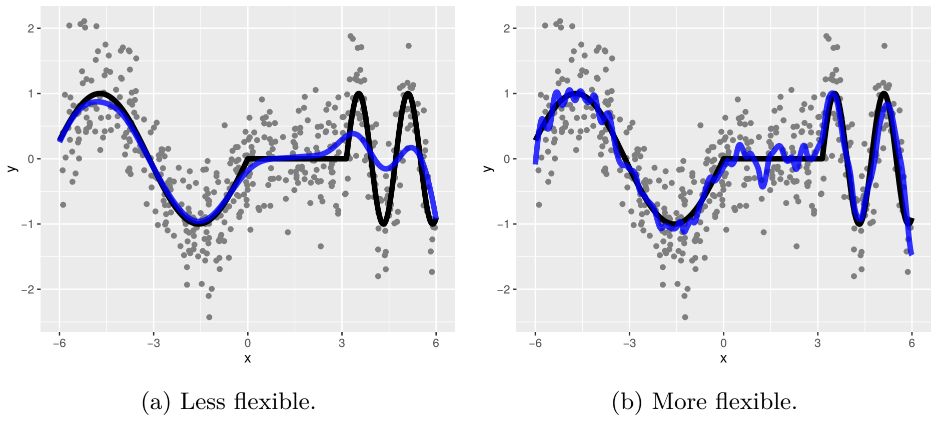


Figure 9.2: Local linear regression with two different degrees of flexibility fit to the simulated data.

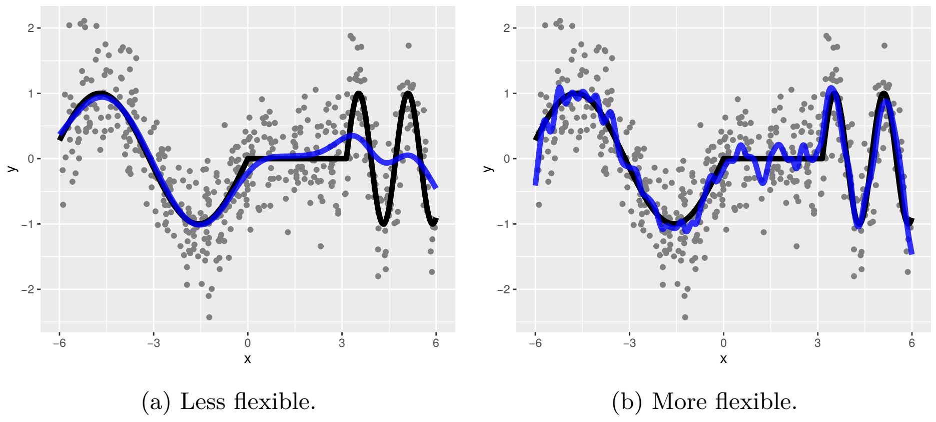


Figure 9.3: Smoothing splines with two different degrees of flexibility fit to the simulated data.

Finally, we fit an additive tree model (with $\eta = 0.02, M = 4000$) to the data using tree stumps. The result is displayed in Figure 9.4. We observe that in areas with simpler structure, the additive tree model has put less effort into fitting the data. In the region where the target function is constant, for example, the model

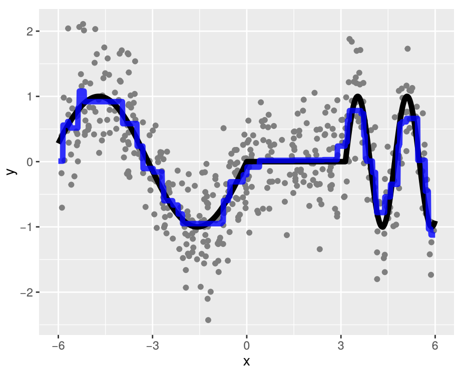


Figure 9.4: Boosted tree stumps fit to the simulated data.

has put very little effort into capturing any structure. In the region with more complex structure however, more effort has been put in to capture it. Tree boosting can thus be seen to use adaptive neighbourhood sizes depending on what seems necessary from the data. In areas where complex structure is apparent from the data, smaller neighbourhoods are used, whereas in areas where complex structure seems to be lacking, a wider neighbourhood is used.

To better understand the nature of how local neighbourhoods are determined, we will attempt make the notion of the neighbourhood more concrete. We will do this by considering the interpretation that many models, including additive tree models, local regression and smoothing splines, can be seen to make predictions using a weighted average of the training data.

9.1.2 The Weight Function Interpretation

Many models can be written in the form

$$\hat{f}(x) = \hat{w}(x)^T y,$$

where $\hat{w}(x)$ is a weight function, $\hat{w} : \mathcal{X} \to \mathbb{R}^n$. For each $x \in \mathcal{X}$, the weight function $\hat{w}(x)$ specifies the vector weights to use in the weighted average of the responses in the training data.

We can write that variance of the model as

$$\text{Var}[\hat{f}(x)] = \text{Var}[\sum_{i=1}^{n} \hat{w}_i(x) Y_i] = \sum_{i=1}^{n} \hat{w}_i(x)^2 \text{Var}[Y_i] = \sigma^2 \sum_{i=1}^{n} \hat{w}_i(x)^2.$$

From this, we can see that in order to keep the variance low, the weights should be spread out as evenly as possible, thus keeping the neighbourhood wide. The globally constant model keeps $\hat{w}_i(x) = 1/n, \forall x \in \mathcal{X}, i \in \{1, ..., n\}$ and thus keeps the variance as low as possible. If the target function is sufficiently complex however, this model will be severely biased. To decrease the bias, the weights have to be shifted such that points which are similar or close to x receives larger weight, while distant and dissimilar points receive lower weights.

Consider for example linear regression. Predictions are given by

$$\hat{f}(x) = x^T(X^TX)^{-1}X^Ty.$$

The weight can thus be written as

$$\hat{w}(x)^T = x^T(X^TX)^{-1}X^T.$$

For local linear regression, the weight function is a simple modification of this. It can be written

$$\hat{w}(x)^T = x^T(X^TW(x)X)^{-1}X^TW(x),$$

where $W(x)$ is a diagonal matrix where diagonal element i is $\kappa(x, x_i)$ (Hastie et al., 2009). The weight functions for local linear regression at three different points for two different degrees of flexibility are shown in Figure 9.5. The two different degrees of flexibility are the same as those used in Figure 9.2. We observe that the weight function has a particular shape, regardless of the position in $x \in \mathcal{X}$. Also, as expected, the less flexible the model is, the more spread out its weights are.

Another example is smoothing splines. The weight function can in this case be written

$$\hat{w}(x)^T = \phi(x)^T(\Phi^T\Phi + \lambda\Omega)^{-1}\Phi^T,$$

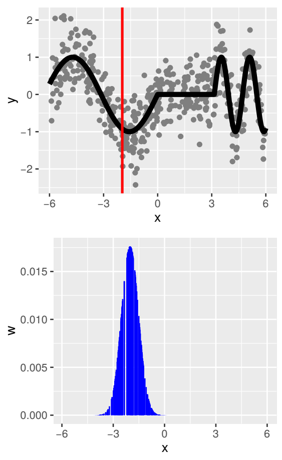
where $\Omega_{jk} = \int \phi_j^{''}(t)\phi_k^{''}(t)dt$ and λ is a regularization parameter which penalizes lack of smoothness (Hastie et al., 2009). In Figure 9.6, the weight functions for smoothing splines at three different points and for two different flexibilities are shown. We here observe the same phenomenon as for local linear regression, namely that the weight function takes the same form regardless of $x \in \mathcal{X}$.

For these models, and many others, $\hat{w}(x)$ is determined using only the location of the predictors x_i in the input space $\mathcal{X}$, without regard for the responses y_i. These models can thus be seen to have made up their mind about which points are similar beforehand. Similarity is often determined by some measure of closeness of points in the input space $\mathcal{X}$. Intuitively, most models will assign larger weights to data points which are determined to be closer to x.

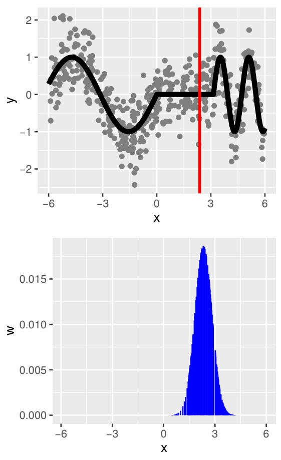
For additive tree models, on the other hand, the weight function can be seen to be determined adaptively. That is, while the other methods only take the predictors x_i in the training data into account when determining $\hat{w}(x)$, additive tree models also considers the responses in the training data. This is in fact a property of tree models, which adaptively determines neighbourhoods and fits a constant in each neighbourhood. Additive tree models inherit this from tree models and uses it to adaptively shift the weight functions at each iteration. At iteration m, boosting updates the model according to

$$\hat{f}^{(m)}(x) = \hat{f}^{(m-1)}(x) + \hat{f}_m(x)$$
$$\hat{w}^{(m)}(x)^Ty = \hat{w}^{(m-1)}(x)^Ty + \hat{w}_m(x)^Ty.$$

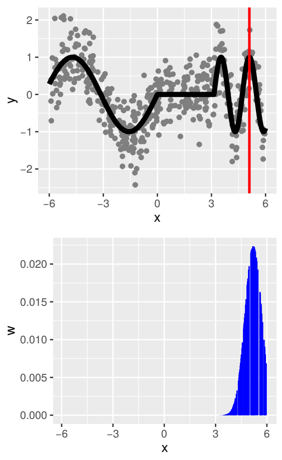
Tree boosting can thus be seen to update the weight functions at each iteration. At each iteration, the learning algorithm searches for splits that minimize the empirical



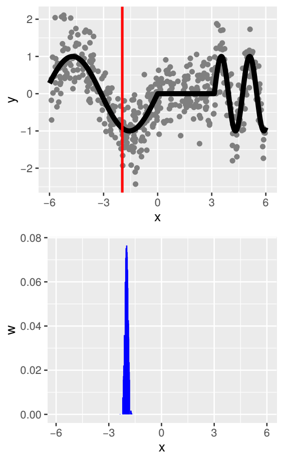
(a) Point 1, low flexibility.



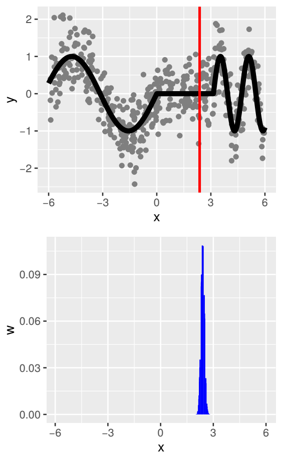
(b) Point 2, low flexibility.



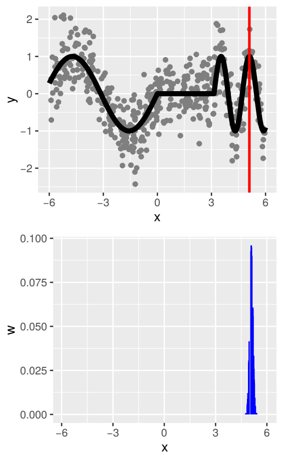
(c) Point 3, low flexibility.



(d) Point 1, high flexibility.

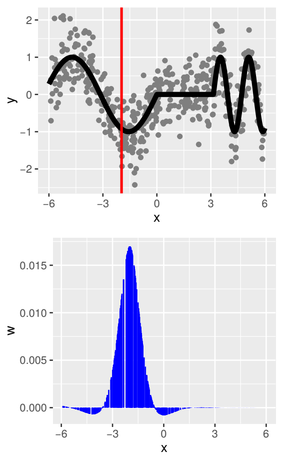


(e) Point 2, high flexibility.

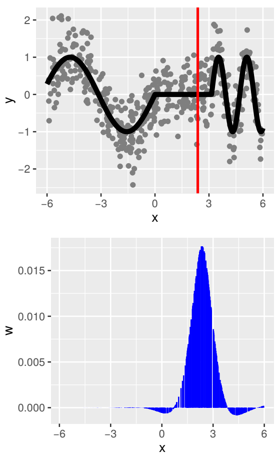


(f) Point 3, high flexibility.

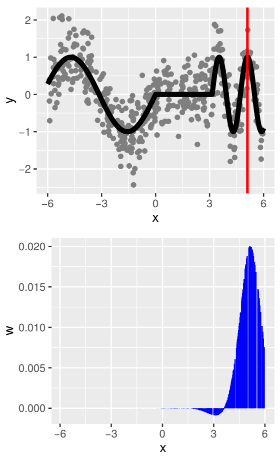
Figure 9.5: The weight function at 3 points for local linear regression with 2 different degrees of flexibility.



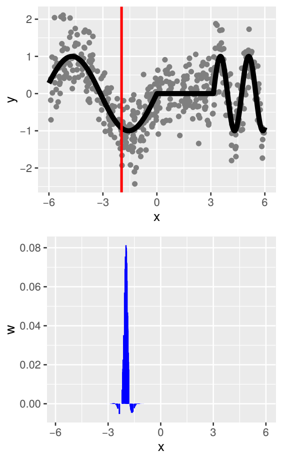
(a) Point 1, low flexibility.



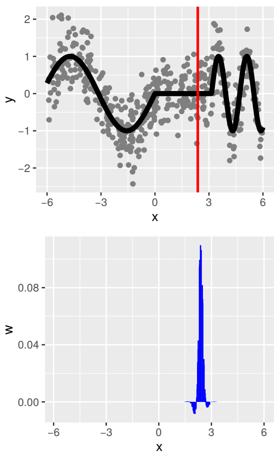
(b) Point 2, low flexibility.



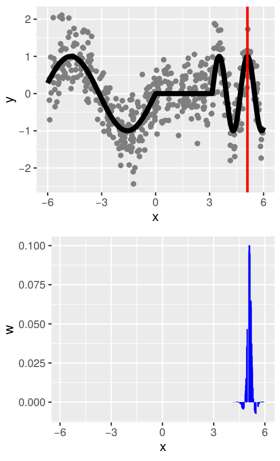
(c) Point 3, low flexibility.



(d) Point 1, high flexibility.



(e) Point 2, high flexibility.



(f) Point 3, high flexibility.

Figure 9.6: The weight function at 3 points for smoothing spline with 2 different degrees of flexibility.

risk. In this process, it can be seen to learn which points can be considered similar, thus adaptively adjusting the weight functions in order to reduce empirical risk. The additive tree model initially starts out as a global constant model with the weights spread out evenly, and thus has low variance. At each subsequent iteration, the weight functions are updated where it seems most necessary in order to reduce bias.

Tree boosting can thus be seen to directly take the bias-variance tradeoff into consideration during fitting. The neighbourhoods are kept as large as possible in order to avoid increasing variance unnecessarily, and only made smaller when complex structure seems apparent. Using smaller neighbourhoods in these areas can thus dramatically reduce bias, while only introducing some variance.

We will now show how the weight functions are adjusted by tree boosting at each iteration with the squared error loss. Consider the tree to be added at iteration m,

$$\hat{f}_m(x) = \sum_{j=1}^{T} \hat{\theta}_{jm} \mathrm{I}(x \in \hat{R}_{jm}).$$

The leaf weights for this tree is determined by

$$\hat{\theta}_{jm} = -\frac{G_{jm}}{n_{jm}} = \frac{\sum_{i \in \hat{I}_{jm}} [y_i - \hat{w}^{(m-1)}(x_i)^T y]}{n_{jm}}.$$

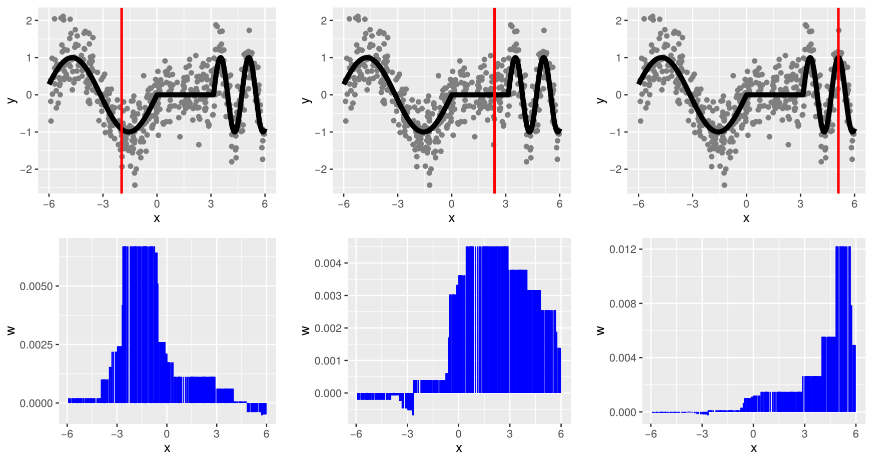
Manipulating this expression, we find that

$$\hat{\theta}_{jm} = \sum_{i=1}^{n} y_i \left[\frac{\mathrm{I}(x_i \in \hat{R}_{jm}) - \sum_{k \in \hat{I}_{jm}} \hat{w}_i^{(m-1)}(x_k)}{n_{jm}} \right].$$

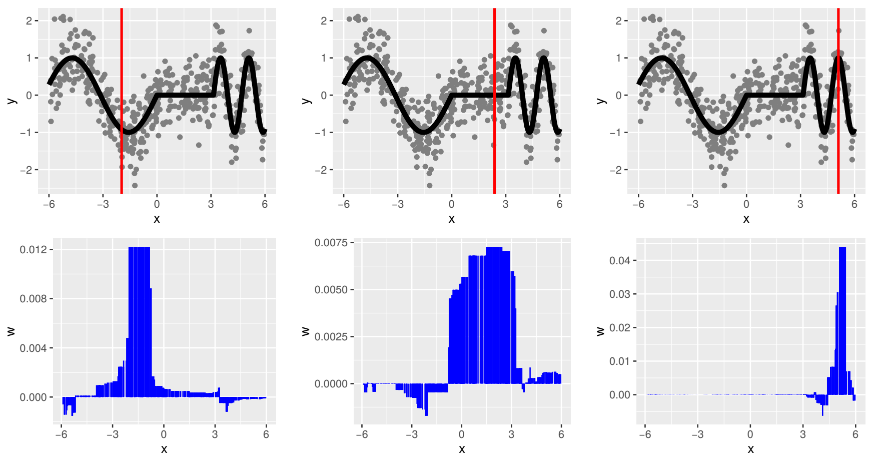
The update of element i of the weight function at x at iteration m is thus given by

$$\hat{w}_i^{(m)}(x) = \hat{w}_i^{(m-1)}(x) + \sum_{j=1}^{T} \mathrm{I}(x \in \hat{R}_{jm}) \left[\frac{\mathrm{I}(x_i \in \hat{R}_{jm}) - \sum_{k \in \hat{I}_{jm}} \hat{w}_i^{(m-1)}(x_k)}{n_{jm}} \right].$$

The weight functions for an additive tree model after 400 and 4000 iterations at the three different points are shown in Figure 9.7. The additive tree model shown in Figure 9.4 was for 4000 iterations. We see that the weight function is more spread out for lower iterations. The main point to observe however, is that the weight functions are different at different values of x. At the point in the flat region of the target function in Figure 9.7e, the weight function is spread out over the similar points nearby. This allows the model to calculate the prediction at this point with low variance, without introducing much bias. At the point in the region where the target function is most complex, shown in Figure 9.7f, the weight function is more peaked around x. This keeps bias low, which seems appropriate in this region of the input space. Finally, for the region where the target function is less complex, shown in Figure 9.7d, the peakedness of the weight function is somewhere between the two other. More interestingly however, the weight function is not centered around x, but seems to assign more weight to points at higher values of x. This also seems appropriate as these points are more similar, whereas in the direction of decreasing values of x, the target function changes more rapidly.



(a) Point 1, iteration 400. (b) Point 2, iteration 400. (c) Point 3, iteration 400.



(d) Point 1, iteration 4000. (e) Point 2, iteration 4000. (f) Point 3, iteration 4000.

Figure 9.7: The weight function at 3 points for boosted trees after 400 and 4000 iterations.

9.2 Boosting With Tree Stumps in Multiple Dimensions

In the last section, we considered tree boosting in one dimension. Some of the greatest benefits of tree boosting are however not apparent when considering one-dimensional problems, as tree boosting is particularly useful for high-dimensional problems.

When the dimensionality of the problem increases, many methods break down. This is, as discussed earlier, due to what is known as the curse of dimensionality. Many methods rely on some measure of similarity or closeness between data points, either implicitly or explicitly, in order to introduce locality in the models. Since distance measures become less useful in higher dimensions, these methods tend to not scale well with increasing dimensionality. Techniques such as feature selection and stronger regularization might be employed to combat this. However, good results still depends crucially on good transformations and relative scalings of the features and specification of appropriate amount of flexibility for each feature. For high-dimensional problems, this can be an almost impossible task.

Tree boosting "beats" the curse of dimensionality by not relying on any distance metric. Instead, the similarity between data points are learnt from the data through adaptive adjustment of neighbourhoods. Tree boosting initially keeps the neighbourhood global in all directions. The neighbourhoods are subsequently adjusted to be smaller where it seems most necessary. It thus starts out as a globally constant model and decreases bias by decreasing neighbourhood size. The property of adaptive neighbourhoods might not be needed as often for one-dimensional problems that we discussed in the last section. When the dimensionality is increased however, it is likely to beneficial. This allows the model to adaptively determine the amount of flexibility to use for each feature. In the extreme case, the method might use no flexibility at all for some features, i.e. keep the neighbourhood globally constant along them. The additive tree model will be unaffected by these features and can thus be seen to perform automatic feature selection.

It is thus the same property of adaptively determined neighbourhoods that we discussed in the previous section that makes it "immune" to the curse of dimensionality. By using adaptive neighbourhoods, the model also becomes invariant under monotone transformations of the inputs. This can thus potentially save a lot of work spent searching for appropriate transformations. Moreover, the relative scalings features are irrelevant for the model.

So far, we have considered tree boosting using only tree stumps. Consequently, our additive tree model would only be able to capture additive structure in the data, not any interactions.

9.3 Boosting With Deeper Trees

To capture interactions, we need deeper trees. The deeper the trees are allowed to be, the higher the orders of interactions we can capture. For an additive tree model where the maximum number of terminal nodes is $T_{\max}$, the highest order of interaction that can be captured is $\max(T_{\max} - 1, p)$. There are few other methods which are able to capture high order interactions from high-dimensional data without breaking down. This is one of the great benefits of additive tree models. Again, it is due to the property of adaptive neighbourhoods that it does not break down. That is, most interactions are not modeled at all, only interactions which seem beneficial to the model is included.

Although deeper trees allow us to capture higher order interactions, which is beneficial, they also give rise to some problems. With deeper trees, the number of observations falling in each terminal node will tend to decrease. Thus, the estimated leaf weights will tend to have higher variance. Stronger regularization might therefore be required when boosting with deeper trees. Another related problem is that the model may model interactions where they are not present. Consider for simplicity a two-dimensional problem where only additive structure is present. At early boosting iterations, a lot of the structure is still left in the data. Thus, after the first split, the second split may be taken along the other feature, thereby giving rise to an apparent interaction. It can thus confuse additive structure for interactions. This will unnecessarily increase variance since the neighbourhood is not kept as wide as it could have been. This might be an area where current boosting methods might be improved.

9.4 What XGBoost Brings to the Table

All the discussion so far have been general to tree boosting and is therefore relevant for both MART and XGBoost. In summary, tree boosting is so effective because it fits additive tree models, which have rich representational ability, using adaptively determined neighbourhoods. The property of adaptive neighbourhoods makes it able to use variable degrees of flexibility in different regions of the input space. Consequently, it will be able to perform automatic feature selection and capture high-order interactions without breaking down. It can thus be seen to be robust to the curse of dimensionality.

For MART, the number of terminal nodes is kept fixed for all trees. It is not hard to understand why this might be suboptimal. For example, for high-dimensional data sets, there might be some group of features which have a high order of interaction with each other, while other features only have lower order interactions, perhaps only additive structure. We would thus like to use deeper trees for some features than for the others. If the number of terminal nodes is fixed, the tree might be forced to do further splitting when there might not be a lot of evidence for it being necessary. The variance of the additive tree model might thus increase unnecessarily.

XGBoost uses clever penalization of the individual trees. The trees are consequently allowed to have varying number of terminal nodes. Moreover, while MART uses only shrinkage to reduce the leaf weights, XGBoost can also shrink them using penalization. The benefit of this is that the leaf weights are not all shrunk by the same factor, but leaf weights estimated using less evidence in the data will be shrunk more heavily. Again, we see the bias-variance tradeoff being taken into account during model fitting. XGBoost can thus be seen to be even more adaptive to the data than MART.

In addition to this, XGBoost employs Newton boosting rather than gradient boosting. By doing this, XGBoost is likely to learn better tree structures. Since the tree structure determines the neighbourhoods, XGBoost can be expected to learn better neighbourhoods.

Finally, XGBoost includes an extra randomization parameter. This can be used to decorrelate the individual trees even further, possibly resulting in reduced overall variance of the model. Ultimately, XGBoost can be seen to be able to learn better neighbourhoods by using a higher-order approximation of the optimization problem at each iteration than MART and by determining neighbourhoods even more adaptively than MART does. The bias-variance tradeoff can thus be seen to be taken into account in almost every aspect of the learning.

Chapter 10

Conclusion

Tree boosting methods have empirically proven to be a highly effective and versatile approach to predictive modeling. For many years, MART has been a popular tree boosting method. In more recent years, a new tree boosting method by the name XGBoost has gained popularity by winning numerous machine learning competitions. In this thesis, we compared these tree boosting methods and provided arguments for why XGBoost seems to win so many competitions.

We first showed that XGBoost employs a different form of boosting than MART. While MART employs a form of gradient boosting, which is well known for its interpretation as a gradient descent method in function space, we showed that the boosting algorithm employed by XGBoost can be interpreted as Newton's method in function space. We therefore termed it Newton boosting. Moreover, we compared the properties of these boosting algorithms. We found that gradient boosting is more generally applicable as it does not require the loss function to be strictly convex. When applicable however, Newton boosting is a powerful alternative as it uses a higher-order approximation to the optimization problem to be solved at each boosting iteration. It also avoids the need of a line search step, which can involve difficult calculations in many situations.

In addition to using different boosting algorithms, MART and XGBoost also offers different regularization parameters. In particular, XGBoost can be seen to offer additional parameters not found in MART. Most importantly, it offers penalization of the individual trees in the additive tree model. These parameters will affect both the tree structure and leaf weights in order to reduce the variance in each tree. Additionally, XGBoost provides an extra randomization parameter which can be used to decorrelate the individual trees, which in turn can result in reduction of the overall variance of the additive tree model.

After determining the different boosting algorithms and regularization techniques these methods utilize and exploring the effects of these, we turned to providing arguments for why XGBoost seems to win "every" competition. To provide possible answers to this question, we first gave reasons for why tree boosting in general can be an effective approach. We provided two main arguments for this. First off, additive tree models can be seen to have rich representational abilities. Provided that enough trees of sufficient depth is combined, they are capable of closely approximating complex functional relationships, including high-order interactions. The most important argument provided for the versatility of tree boosting however, was that tree boosting methods are adaptive. Determining neighbourhoods adaptively allows tree boosting methods to use varying degrees of flexibility in different parts of the input space. They will consequently also automatically perform feature selection. This also makes tree boosting methods robust to the curse of dimensionality. Tree boosting can thus be seen actively take the bias-variance tradeoff into account when fitting models. They start out with a low variance, high bias model and gradually reduce bias by decreasing the size of neighbourhoods where it seems most necessary.

Both MART and XGBoost have these properties in common. However, compared to MART, XGBoost uses a higher-order approximation at each iteration, and can thus be expected to learn “better” tree structures. Moreover, it provides clever penalization of individual trees. As discussed earlier, this can be seen to make the method even more adaptive. It will allow the method to adaptively determine the appropriate number of terminal nodes, which might vary among trees. It will further alter the learnt tree structures and leaf weights in order to reduce variance in estimation of the individual trees. Ultimately, this makes XGBoost a highly adaptive method which carefully takes the bias-variance tradeoff into account in nearly every aspect of the learning process.