4_Gradient_Boosting

Nicholas Mitchell

August 12, 2016

Contents

1 Gradient Boosting

1.1 What is boosting?

There were several important developments that were necessary to arrive at the model that this study ultimately uses. They are each addressed in the following sections, and may be summarised as follows:

1. The first **boosting** algorithm was defined: AdaBoost [?], [?]
2. AdaBoost was re-formulated as **gradient descent**, taking a special loss function [?], [?]
3. AdaBoost was generalised to allow a multiplicity of loss functions: **gradient boosting** [?], [?]

The generalised final product, gradient boosting, can be used for regression, classification and ranking problems, making it a tool with a diverse range of applications, in many different contexts.

As listed above, the first notions of boosting stem from work by Freund and Schapire, in which they developed an algorithm named *AdaBoost* (full form: adaptive boosting[1]) that gradually improves model performance on a given data set over a number of iterations. Instead of defining one complex function, AdaBoost iteratively fits a simple model - a *base-learner* - to the data set. Before the first iteration, all data points are considered of equal importance, and so uniformly weighted. In the case of regression with square loss, the residual errors produced from each iteration (labelled the *shortcomings*) are subsequently used to re-weight the data points. Each falsely classified point receives more weight (proportional to the log-odds of the previous iteration's weighted error) and, in a similar fashion, the weights of the correctly classified data points are decreased. Finally, all weights are normalised to sum to unity before the next iteration[2]. This weight adjustment steps instructs the naive model how to prioritise its fit or classification in the following iteration - targeting areas where the model performs weakly. It is possible that many mistakes are made by the naive base-learner used at each stage throughout the process; however, the weights reflect how many times each point was classified falsely, and it is the sum of these weights that defines the final model.

The shortcomings are the residuals from each fit, which are used to iteratively re-fit a function. The model is updated after each iteration of re-modelling on the shortcomings, thus incrementally improving the model and decreasing the observed error on the data set. If this overall error is thought of as a function of the model to be minimised, then the residuals correspond to the negative gradient of that function. In order to generalise the AdaBoost methodology for other loss functions, (as is described in the following sections), it is more useful to retain this concept of gradients when referring to the shortcomings of a model. Re-expressing the boosting methodology of AdaBoost in this generalised way: we are making approximations to the gradient of a loss function in each step - this leads to the descriptive name *functional gradient descent* (FGD).

1.2 Building on the basics

The simple idea of linear regression ([?] ch.4) may be thought of as the fundamental idea behind much of the boosting methodology used in this study. It is what is often used as a starting point[3] when building a boosting-based model and so may be considered an elementary building block e.g. when defining in a generalised linear model (GLM).

We consider a function, $f : \mathbf{X} \to \mathbf{Y}$, that describes a data set by mapping an input, $\mathbf{X} = \{X_1, X_2, \ldots, X_n\}$, to a response variable $\mathbf{Y}$. Given the input and output variables, it is this function, which is to be approximated. In the classic example of linear regression, this means placing a straight line over or through the data set, which is as close as possible to as many of the points within that data set as possible. The *goodness of fit* of that line can be quantified and so assessed by some metric on the distance found between the line and each of the individual points. In the case of the ordinary least squares (OLS) methodology[4], this is achieved by summing the *squares* of the distances between each point and the line, and then adjusting the line so that defined sum becomes as small as possible; thus the problem is one of minimisation[5]. To help visualise this, the distances are highlighted by red arrows

[1]A qualitative description only is given here. For a more detailed illustration, please refer to the referenced literature.

[2]As each step is reliant on the previous, the models that are fitted are not independent from one another. Within the context of machine learning, this is often termed an *ensemble method*.

[3]Here we mean in terms of a base-learner and initial *go-to method* before more complicated models are considered.

[4]If errors are assumed to be normally distributed, we could instead refer to the maximum likelihood estimation (MLE)†.

[5]More generally speaking the problem is one of optimisation; minimisation when optimising over a loss function, however *maximisation* when optimising over a likelihood function

in Figure 1, which is a short excerpt of some sentiment analysis data (see Section `data-exploration` for more information regarding the data used in this study). It is the sum of these lengths squared that is to be minimised by moving the line. We notice that the lines are all vertical, which is because we are interested only in the distance (also called the error or the residual) between the outcome variable that our line would predict for a given input value on the x-axis, and the *true* outcome, i.e. the y-coordinate of the corresponding point from the data set.

`/Volumes/Mac OS Drive/Thesis/Source Code/Reporting/nwm_Report/images/basic_residuals.png`

Figure 1: Sentiment scores of term "dow jones" versus Dow Jones returns of following day. Ten consecutive days are randomly selected and displayed with their residual errors as red arrows to the regression line, which is produced via a linear fit over all 695 data points. Inset: the same plot without error highlighting, including all data points.

The line represents our function, an approximation to the data set, and describes a one-dimensional response variable Y with one explanatory variable X. As will be shown in the following sections, this is often the case in component-wise gradient boosting, where a single variable vector is regressed on the single response variable (or iteratively on the residuals of consecutive fits).

1.3 Gradient boosting

1.3.1 The objective

The goal of gradient boosting is the same as any other (in this case, forecasting) model: to fit a function to a data set that accurately describes the data and allows for new input data points to make predictions on the outcome variable, all while minimising error. Consider a sample containing n values for each of p predictor variables $\mathbf{X} = (\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_p})^\top$, along with a corresponding one-dimensional[6] *outcome variable*: $\mathbf{Y}$. The goal is then to find the optimal function $f^*(\mathbf{X})$ that describes the given data and allows predictions to be made for $\mathbf{Y}$. The method should not be a *black-box* [7], i.e. it must be transparent at all stages, and the final model f^* should permit the interpretation of results and interactions between features.

A classic (simple) approach might fit additive regression models using MLE, such as the the linear model example described in Section `linear-model,` applied to each predictor individually. The output of such a model would then take the form shown in Equation (1), which has the structure of an additive predictor provided by a general additive model (GAM) [?] and the functions $\hat{f}_1 + \ldots + \hat{f}_p$ each correspond to the linear functions (later defined as the base-learners). These functions depend only on the predictor variables that were used as inputs. We shall see that single variables are generally use as inputs for each base-learner; however, subsets of variables may be allocated to each base-learner.

$$f^*(\mathbf{X}) = \beta_0 + f_1(\mathbf{X_1}) + f_2(\mathbf{X_2}) + \ldots + f_p(\mathbf{X_p}) \tag{1}$$

There are several points to consider when using such a model, i.e. when selecting base-learners and the subsets of variables to be used as input. Two examples, both of which indeed present themselves in this study's data set, are (1) high levels of pairwise-correlation between predictors, and (2) cases in which we have *wide* data sets ($p > n$). The questions that arise, regard matters such as the explanatory power attainable in the face of m highly correlated variables being included in one model, and then which of those variables provides the most information. A model that can cope with such input data should ideally include an *in-built* method of variable selection that can deal with multicollinearity, and also may return a sparse model, i.e. not all predictors must be included in the final model. Gradient boosting provides a framework in which those concerns are addressed satisfactorily, particularly because each step of the modelling procedure is transparent, allowing errors in modelling assumptions to be identified.

The naive approach to variable selection by means of exhaustively fitting models for all possible subsets of predictors is not an option when the data sets are wide, i.e. with large p. More systematic methods of variable selection, especially the case of wide data sets, can be difficult to perform and do not guarantee the optimal solution. Examples of step-wise search techniques include *backward* and *forward selection*, which avoid exhaustive model fitting. While reducing the number of models fitted, these methods cannot guarantee optimality as the explanatory power of feature interactions is not necessarily considered (most easily demonstrated with forward selection - [?] ch.10).

1.3.2 General properties

Gradient boosting provides a *fitting method* that minimises an empirical risk (loss) function[8] with respect to a given prediction function f. The risk function component is modular, meaning it may take many forms, in each case describing a particular loss function, which is to be minimised[9]. Examples are the L_1 (Laplace), L_2 (Gaussian), Huber, exponential and negative log-likelihood loss functions[10]. In order to minimise such functions, they must be convex and continuously differentiable so that the first derivative may be solved over any interval. There are particular applications of loss function minimisation that make the even stronger assumption that the loss-function be Lipschitz continuous [?], which places an upper bound on the magnitude of the gradient of the function over an interval. Two exceptions to

[6]Vector notation is used here, where $\mathbf{X_1}$ represents the vector $(X_1, X_2, \ldots, X_n)$ for $p = 1$.

[7]Neural networks are the standard example of such a model. High predictive accuracy may be obtained, but the modelling process does not allow for much interpretability. The characteristic of the final model being *non-identifiable* increases difficulty of further analysis. Recent advancements with visualisations† into the understanding of the *hidden layers* have, however, been made.

[8]Empirical risk and loss function are used synonymously. Depending on the context, other names are also given to the same principle, for example: a cost function (neural networks) and the hinge loss (support vector machines), to name just a few.

[9]This terminology is commonly used in *data mining*† communities; however, the distinction between minimising *loss functions* in data mining and maximising *likelihood functions* in classical statistics is rather hazy.

[10]For more detail on unbiased estimators that minimise a risk, refer to Ch. 3 of [?]

continuous differentiability are the L_1 and hinge loss functions, which are non-differentiable at their inflection points. In software implementations this discontinuity is often set to zero as to prevent a computational error being thrown (the gradient being undefined at the inflection points), thus allowing stable usage[11]. The results of boosting are generally in the form of an *additive* predictive function, as depicted by Equation (1), and so are readily interpreted. The further benefits that are reaped by using component-wise gradient boosting[12] will become apparent as the methodology is explored in the following sections, but may first be summarised here by stating that component-wise boosting:

1. is applicable for use with a wide range of loss functions (as described above),
2. is able to perform variable selection during the fitting process,
3. can perform well in situations where $p >> n$, e.g. genetics
4. inherently addresses multicollinearity between predictors, shrinking effect estimates towards zero,
5. optimises prediction accuracy with respect to a given risk function, and
6. offers high transparency throughout the modelling iterations.

1.3.3 Naive functional gradient descent

We first define the basic approach to gradient boosting, describing how gradient descent is performed. At the same time, specific applications are introduced within the context of this study. The component-wise version that was ultimately used is presented in the sections that follow. We start with the initial model, slightly extending that given in Section 1.3.1, by again considering a one-dimensional response variable $\mathbf{Y}$, but with a p-dimensional set of predictors $\mathbf{X} = (\mathbf{X_1}, \mathbf{X_2}, \ldots, \mathbf{X_p})^\top$. A model that aims to fit a model that explains this relationship may be presented thusly:

$$f^* := \underset{f(\cdot)}{argmin}\ \mathbb{E}_{\mathbf{Y},\mathbf{X}}[\rho(\mathbf{Y}, f(\mathbf{X}))] \tag{2}$$

where ρ is a loss function with the properties described in Section 1.3.2. The optimal model $\hat{f}(\cdot) = f^*$ therefore minimises the expected loss, i.e. the error. This study looks at several loss functions[13] that are able to be inserted into Equation (2); two examples of which are the *absolute loss* - illustrated in Equation (3) - and the *squared loss* (used in Gaussian regression) - illustrated in Equation (4).

$$\underset{\ell_1}{\rho} = |\mathbf{Y} - f(\mathbf{X})| \tag{3}$$

$$\underset{\ell_2}{\rho} = (\mathbf{Y} - f(\mathbf{X}))^2 \tag{4}$$

Modelling is performed on a sample set with n realisations $X = (X_1, X_2, \ldots, X_n)$ and $Y = (Y_1, Y_2, \ldots, Y_n)$ of $\mathbf{X}$ and $\mathbf{Y}$, respectively, and so an exact expectation for Equation (2) is unknown. Instead, boosting algorithms minimise an empirical risk function (the *observed mean*) with respect to some approximation function, $f(\cdot)$, given as:

$$\mathcal{R} = \frac{1}{n}\sum_{i=1}^{n} \rho(Y_i, f(X_i)) \tag{5}$$

Referring once again to the principle of modularity, either of the two loss functions given in Equations (3) and (4) (plus *any* other defined loss functions that fulfils the requirement of convexity and continuous differentiability) may be inserted into the empirical risk function given by Equation (5). Equation (6) shows the case for the L_2 loss function.

[11] The L_1 loss function has been shown, theoretically as well as experimentally, to achieve superior performance to the alternatives in certain cases - especially in *sparse* models, where few coefficients are found to be non-zero (see [?] p.611-613)

[12] Component-wise gradient boosting is specifically addressed in Section 1.4.

[13] The Gaussian L_2 loss function is used in Section `results-gauss`.

$$\mathcal{R} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - f(X_i))^2 \tag{6}$$

The next step is to establish the iterative *gradient descent* method, which minimises $\mathcal{R} = \mathcal{R}(f_{(1)}, f_{(2)}, \ldots, f_{(n)})$ with respect to the approximation function, $f_{(\cdot)}$; where $f_{(1)} = f(X_1), f_{(2)} = f(X_2), \ldots, f_{(n)} = f(X_n)$. We notice that these each are merely output numbers, meaning we can simply treat $f(X_i)$ as parameters and take derivatives with respect to Equation (5), which for the case of an L_2 loss function, as in Equation (6), yields:

$$\frac{\partial \mathcal{R}}{\partial f(X_i)} = \frac{\partial}{\partial f(X_i)}\left(\sum_{i=1}^{n}\rho(Y_i, f(X_i))\right) = \frac{\partial}{\partial f(X_i)}\left(\rho(Y_i, f(X_i))\right) = f(X_i) - Y_i \tag{7}$$

which illustrates the relationship between the residuals and the negative gradient of the cost function, thereby demonstrating the reason why the former may be interpreted, in a more general manner, via the latter. By simple re-arrangement:

$$Y_i - f(X_i) = -\frac{\partial \mathcal{R}}{\partial f(X_i)} \tag{8}$$

We define initial conditions by first setting the approximation functions $f_{(\cdot)}$ to some offset values $\hat{f}^{[0]}_{(1)}, \ldots, \hat{f}^{[0]}_{(n)}$, where an iteration counter m is set to zero (shown in the superscript of $f_{(\cdot)}$). In each step, an approximate to the gradient of the loss function is computed and used to update the approximation functions as follows:

$$\begin{pmatrix} \hat{f}^{[m]}_{(1)} \\ \vdots \\ \hat{f}^{[m]}_{(n)} \end{pmatrix} = \begin{pmatrix} \hat{f}^{[m-1]}_{(1)} \\ \vdots \\ \hat{f}^{[m-1]}_{(n)} \end{pmatrix} + \nu \cdot \begin{pmatrix} -\frac{\partial}{\partial f_{(1)}}(\hat{f}^{[m-1]}_{(1)}) \\ \vdots \\ -\frac{\partial}{\partial f_{(n)}}(\hat{f}^{[m-1]}_{(n)}) \end{pmatrix} \tag{9}$$

where ν is the learning rate, or *step length factor* - see Section 1.4.2.1 for more discussion on this model parameter. With each iteration of the model, we notice the approximation functions (and so the coefficients), as shown in Equation (9), are **all** simultaneously incremented[14] in each iteration, by an amount proportional to the gradient of the loss function. This is explicitly analogous to the *shortcomings* of AdaBoost (the residuals themselves) being used to improve the model (Section 1.1) and embodies the principle of steepest descent: the coefficients take the shortest path to their final estimates that minimise the loss function. Because of this feature, the algorithm is also know as *greedy* [**?**], [**?**]. The process is illustrated in Figure 2, which represents a contour plot of a fictitious cost function. The process is repeated until the algorithm converges to the values $\hat{f}^{[m_{stop}]}_{(1)}, \ldots, \hat{f}^{[m_{stop}]}_{(n)}$, which correspond the best approximation to the loss function's minimum. Here, $\mathrm{m_{stop}}$ represents the number of iterations required to reach this minimum.

There are several weaknesses to this naive version of FGD, one of which is that any structural relationship between the approximation functions $\hat{f}^{[m_{stop}]}_{(1)}, \ldots, \hat{f}^{[m_{stop}]}_{(n)}$ that act upon the data set are ignored. Simple relationships are assumed: $\hat{f}^{[m]}_{(1)} \rightarrow Y_i, \ldots, \hat{f}^{[m]}_{(n)} \rightarrow Y_n$, which may fail to capture all the information held in the model variables. The algorithm defined in the following section addresses this weakness and improves further upon the progress this naive FGD method has made - describing the primary modelling tool used for this study.

[14]Because all values $f(X_i)$ are updated in each iteration, this procedure may be referred to as *batch* gradient descent. This highlights the difference to *component-wise* gradient descent, described in Section 1.4.

1.4 Component-wise functional gradient descent

1.4.1 Definition and properties

With the two concepts of *boosting* and *gradient descent* having been defined, as well as the method of (batch) gradient boosting [**?**], the final enhancement is to be introduced; defining *component-wise* gradient boosting, which adds the last features that were outlined in Section 1.3.2. Namely, that a form of variable selection be implemented[15] within the boosting process. In addition to variable selection, as will be shown, this also inherently provides a certain amount of assurance of performance in the face of multicollinearity. Algorithm (1) defines the iterative procedure in which component-wise boosting minimises the empirical risk $\mathcal{R}$ (given in Equation (5)) over the approximation function, f.

```
Algorithm 1: Component-wise functional gradient boosting

Input: loss function, $\rho$; base-learners; counter, $m$; learning rate, $\nu \in [0, 1)$
Output: optimal prediction function: $f^*$

Step 1. Initialise the n-dimensional function estimate $\hat{f}^{[0]}$ with offset values        /* e.g.  $:= \mathbf{0}$ */
Step 2. Specify a set of $P$ base-learners; initialise the counter, $m := 0$
Step 3. Increase $m$ by one
Step 4.   a. Compute the negative gradient of the loss function: $\mathbf{u}^{[m]} = -\frac{\partial \rho}{\partial f}$
          b. Evaluate the negative gradient at the previous iteration's model estimate
          c. Fit each of the $P$ base-learners to the resulting negative gradient
          d. Select the base-learner that best fits $\mathbf{u}^{[m]}$ by some criterion        /* e.g.  SSE */
          e. Set $\hat{\mathbf{u}}^{[m]}$ equal to the best fitting base-learner
          f. Update current estimate $\hat{f}^{[m]} := \hat{f}^{[m-1]} + \nu \cdot \hat{\mathbf{u}}^{[m]}$

repeat
    Steps 3 and 4
until $m = m_{stop}$;

return the prediction function that minimises $\rho$: $f^* = \hat{f}^{[m_{stop}]}$
```

Step 1 sets the initial function estimate set to a zero-vector. The P base-learners that are specified in **Step 2** are generally simple estimators that each take a pre-defined set of input variables and provide a univariate response. Each of them may take different subsets of the (entire) model's input variables; the subsets are usually relatively small, in order to make use of the model's features. The base-learners that are provided to a model for implementation within Algorithm (1) provide the tool that allows the modeller to stipulate structural assumptions regarding the model. Beyond simply grouping of variable subsets, several further options are available, such as methods to introduce categorical and ridge-penalised effects - refer to [**?**] for more information on these options.

In this study, the base-learners are least-squares estimators, with input of single predictor variables. Therefore, each base-learner fits a simple linear model against the negative gradient for each of the individual predictor variables. Coupled with shrinkage effects being applied to the best base-learner (due to $\nu < 1$ in **Step 4.f**), this individual predictor modelling highlights how component-wise gradient boosting is still able to perform, should there be relatively high levels of multicollinearity among the predictors. If the learning rate were selected to be $\nu = 1$, the resulting algorithm would be *greedy* (just as the original AdaBoost algorithm) and so would not cope so well with multicollinearity. There is, however, an element of uncertainty in which predictor will be selected over the many iterations. If the correlation between two predictors is extremely high (e.g. > 0.7), which predictor is selected at each step may not be consistent - therefore it must be mentioned that there are limits to this facet of the variable selection feature of component-wise gradient boosting. This is taken into consideration within the empirical segment of this study, as discussed in Section `pairwise-corr`.

[15]This is of great importance for this study, as the inclusion of lagged variables produces data sets where $p > n$.

In **Step 4**, the computed negative gradient estimate is evaluated at the vector estimate of the previous iteration's approximation function, $\hat{f}^{[m-1]}\left(\mathbf{X}_i^\top\right)$, yielding Equation (10):

$$\mathbf{u}^{[m]} = \left(u_i^{[m]}\right)_{i=1,...,n} := \left(-\frac{\partial}{\partial f}\rho\left(Y_i, \hat{f}^{[m-1]}\left(\mathbf{X}_\mathbf{i}^\top\right)\right)\right) \tag{10}$$

The criterion that is used for this study to select the best performing base-learner (as in Step **4.d.**) is the sum of squared errors (SSE), however this may be adapted to the model, for example in the case the base-learners should become more complicated and take forms other than linear models. L_1 absolute error might be a good alternative if the model should be more robust to outliers. As a consequence of this study using base-learners containing only individual variables, the variable selection process (carried out in **Step 4.d.**) inherently extends to the notion of *model selection*, in this particular case. As previously mentioned, the choice of base-leaner provides a means to specifying structural assumptions, and the efficacy of those choices can be seen in this step. The learning rate, ν, for use in **Step 4.f.** should be a real number lying on the interval $[0, 1)$. More discussion on this parameter can be found in Section 1.4.2.1. The last major point of interest within Algorithm (1) is the parameter m_{stop}. A method to approximate an optimal value for this important parameter is described in more detail in Section 1.4.2.2.

The model description given in Section 1.4 initialises the weights to zero. There are several reasons why this is a reasonable choice. Firstly, initialising the values to zero means that in the case of a particular variable never representing the closest approximation to the negative gradient of the loss function, **u** - by not once producing the base-learner fit with the lowest error over all variables - this variable is never selected and so its weight never incremented. Keeping in mind that these weights correspond to the coefficients of the variables in the final model, this then equates to the final value of this coefficient at completion of the gradient descent remaining untouched, equal to zero, ergo the variable is not selected for the final model. This is part of the intuition behind the inherent feature of variable selection presented by component-wise gradient boosting[16]. The second useful property of using zero as the initial weights is that, regardless of whether a coefficient evolves to be positive or negative at completion, the starting point was the same. To a moderate extent, this symmetry additionally facilitates the direct comparison of variable importance via their coefficients magnitudes, which tell us *how far* each base-learner progressed the model along the surface of the loss function as it approached the minimum. This is of course a function of the negative gradient and the learning rate.

This section has summarised the main methodology used within this study; however some preliminary testing was also completed using several variants of this model. Some extra information explaining their usage is explained in the following sections.

1.4.2 Parameter selection

1.4.2.1 Learning rate: ν The learning rate, ν, is commonly held constant throughout the boosting process, which has proven to be a simple, yet effective method. To see why this approach is indeed an elegant, we must inspect the magnitude of the increments to our approximation function during the gradient descent, not only the scalar learning rate itself. One might consider different learning rates and their effect on the speed of approach to the loss-function's minimum (given there being only one global minimum). Given a tiny learning rate, the speed of approach would be extremely slow; however, offering a very close approximation to the minimum as a by-product. Selecting a large learning rate would conversely allow for a rapid descent towards the minimum; however, offering relatively little precision. The truth, however, is that the increments that are added to our approximation function at each iteration *are* indeed adaptive - in terms of the negative gradient, which must decrease as the gradient descent approaches the minimum, by the definition of the loss function being convex. This can be seen in Figure 3, where a simple one-dimensional case is demonstrated.

A fictitious loss function is plotted for the one-dimensional case (the parabola: $y = x^2 + \frac{1}{2}$), where the colour gradient of the curve reflects the magnitude of the negative gradient, $u = \left(-\frac{\partial}{\partial f}\rho(Y, f)\right)$; dark blue indicates a steep gradient, which slowly lightens as the function levels out to its minimum. A value for ν of 0.1 is defined, giving the size of the red points reflects the magnitude, $\nu \cdot u$, by which the approximation function is incremented during gradient descent. It is clear that the gradient descent will take steps

[16]Any possibility of a base-learner being incremented more than one time, and providing a final value $f_i^{[m_{stop}]}$ equal to zero, is completely ruled out by the requirement of the loss-function being convex. The increments for one particular base-learner (and so the variables contained) can only be of one sign, positive **or** negative, meaning the summation may never converge to zero. (Stochastic gradient boosting may, however, exhibit a non-zero possibility of this phenomenon.)

of ever decreasing length as the minimum is approached, the decrease in step-site proportional to the reduction in the gradient. Therefore there is no obvious benefit gained by e.g. adaptively decreasing ν over the iteration process. The accuracy this method offers, via the self-adapting step-length, is assumed to provide sufficient precision when approximating the loss function's minimum, even though ν is held constant.

Of course the *constant* value of ν is still a model parameter to be optimised, if not for precise approximations, for the the gradient descent to be performed as efficiently as possible in terms of computational cost. For performance, it is shown by the authors [?] to suffice to use a small value, e.g. 0.1, as is demonstrated in Figure 3. It must lie on the half-open interval $(0, 1]$, The usage of this variable within this study is discussed within Section `param-grid`. Furthermore, to dynamically adapt the step-size factor to the iteration count of the negative gradient does not improve the estimates of f^*, and will only increase the computational cost of running a descent to convergence. It is worth noting, additionally, that large values may prevent convergence to the minimum, and could even lead to cases of divergence[17].

1.4.2.2 Stopping iteration: m_{stop}

As already discussed in previous sections, there are two main input parameters that have a large effect on the overall performance of the boosting algorithm: ν and m_{stop}. The learning rate, ν, must be assigned a sensible value (which depends upon the input data); however, has the lesser impact of the two parameters. The more important model parameter is the number of boosting iterations, i.e. when to end the gradient descent algorithm, which requires optimisation with regards to the data set at hand. Up until this point, this was merely labelled as the point when Algorithm (1) converges, m_{stop}; in practice however, the exact value is less well defined and must be optimised empirically. One must consider the reality of *overfitting* the model to the in-sample (*training*) data set. If the model trains too closely to the data, the resulting prediction function will likely perform badly in out-of-sample testing. It is therefore necessary to perform some manner of cross validation on the results obtained from the gradient descent procedure.

One could perform the cross-validation in a number of ways, for example, using (1) k-fold cross validation, (2) sub-sampling and (3) bootstrapping methods - all of which are implemented within the `mboost` package, via the function: `cvrisk()`. This study exclusively made use of bootstrapping methods, whereby the number of cross-validation replications used was 25. The output model objects (created by the `mboost_fit` function) do not simply contain the final approximation function with the coefficients of the selected variables, but rather the information from every iteration from the gradient descent. The cross-validation can then bootstrap the results at each boosting iteration and record the error. Executing 25 bootstrapped replications then allows the average (squared) error to be computed - the iteration that holds the minimum value from this set of results indicates the optimal value of iterations, m_{stop}.

Consider a model that was produced from component-wise boosting, running in total for 100 iterations. Figure 4 illustrates the cross-validation methodology on the outcome, illustrating how the optimal number of iterations, m_{stop}, is identified. The iteration number is selected, where the minimum error over the 25-bootstrapped samples is found - the process is labelled an *early stopping strategy*, which aims to optimise the final models prediction accuracy.

The m_{stop} parameter affects the level of relative complexity[18] that the chosen model exhibits. If the final approximation function is selected, which was produced at the hundredth iteration (in the example given), the model is likely to include many parameters, which were selected in an exhaustive search for the minimum of the loss function. Parameters that perhaps have little effect on the outcome variable may have been included, leading to overfitting. In comparison, the value of m_{stop} obtained from (bootstrap) cross-validation not only has a lower average squared error (by definition), but is also likely to return much simpler model. Less iterations would have been performed, meaning that less variables in a wide data set can have been selected, and those that were

`/Volumes/Mac OS Drive/Thesis/Source Code/Reporting/nwm_Report/images/cvrisk-example.png`

[17]To show this, one must simply use the argument presented in Figure 3, instead using a large value of ν to see that the minimum may easily be overshot.

[18]The term 'complexity' is naturally somewhat subjective, being model dependent.

Figure 4: Example of 25-fold bootstrap cross-validation for a model with 100 iterations. Each of the 25 light grey lines shows the error at each iteration for each bootstrap. The black line displays the average over all bootstrap results. The minimum of

selected are the influential variables. As can be seen from Figure 4, the error reaches a minimum quit early on, and plateaus out. This means the approximation function at iteration $m_{stop} = 33$ explains just as much variance in the data set as the model at iteration $m = 100$, so selecting the simpler model is good practice (by arguments of model parsimony, i.e. 'Ockham's Razor').

The `mboost` literature, [?], also discusses usage of alternative criteria in order to locate the m_{stop} value. The example given is that of the Akaike Information Criterion (AIC). It is suggested that this method, however, tends to produce larger values of m_{stop}, which overshoot the minimised squared error shown through cross-validation.

1.5 Stochastic gradient boosting

The introduction of a stochastic component to gradient boosting has proved to be a great tool in reducing the standard errors in a final prediction model by variance reduction, especially when the predictors show signs of correlation. The idea of variance reduction is already seen in older machine learning algorithms such as *bagging* (bootstrap aggregation) and random forests [?]. The former creates many subsets of the training data to fit many models, and taking the average produces results with smaller errors than would've been found by using the whole training data. The bootstrapping procedure is out-of-bag (OOB) with replacement, meaning some variables will be selected many times and other perhaps not at all, depending on the setup. Random forests enhances bagging further still by saying (in terms of classification trees): at each stage when a tree has been fitted to one of the bootstrapped samples, select a subsets m of the p variables from the terminal nodes ***at random***, then select only the best variable among those m variables for the next split point.

This insertion of a stochastic procedure - randomly selecting from the variables after fitting the tree - squeezes as much variance reduction as possible into the modelling and therefore reduces the final error as much as possible when averaging. It is this notion that is applied to gradient boosting, thus making it stochastic. The random selection can occur at one of two places, creating either a random subsets of the training data (à la bagging), or a random subset of the features found at each step (à la random forests). The randomness in the model fit and reduction in final estimate errors, coupled with a slower loss-function descent, may also hinder over-fitting and so improve the models ability to generalise to out-of-sample data.

Other than the inclusion of this simple step into the method that was set out in Section 1.3, no other changes are made to the iterative procedure. As one may expect, removing information at each iteration can mean that some of the other model hyper-parameters must be adjusted, i.e. differing optimal values are likely to be found. The tendency is for the model to require a larger number of iterations to descend along the surface of loss function to the minimum.

Additionally, of course, there is the introduction of a new hyper-parameter, namely the proportion of data or features that are to be selected at random. This parameters can also be optimised for; however, Friedman ([?] Page 5, Figure 1) found that values between 50 % and 80 % provided the lowest errors.

A small practical note: the R package `mboost` has the functionality of the bagging approach, sub-sampling the input data, but does not support the random forest enhancement. The functionality is, however, provided by the `gbm` package, which finds implementation within this study, in Section 1.5.

1.6 Families of distributions

Depending on the requirements of the model, a specific family must be selected. The `mboost` package in R supplies many families. They are listed, along with their properties in Table 4 of *Model-based Boosting in R* [?]. This section briefly outlines the practical aspects of several further families of regressors that are used within this study. A rudimentary outline is provided to explain in which situation each family may be used, indicating why the methods were required in several aspects of the empirical work.

1.6.1 Gaussian

This family was used extensively for the general linear models performed on all data sets in order to predict stock market movements, discussed further in Section `main-modelling`. The Gaussian family is used in order to provide the conditional mean of a continuous response. In this case, the assumption is that the conditional outcome distribution, $\mathbf{Y}|\mathbf{X}$, is normally distributed and that the loss function is the negative Gaussian log-likelihood, which is equivalent to the L_2 loss - given in Equation (11):

$$\rho(Y, f(X)) = \frac{1}{2} \cdot (Y - f(X))^2 \tag{11}$$

1.6.2 Binomial

This family is used in order to model a binomial *class* response[19]: $\{0, 1\}$. Just as the Gaussian family, the binomial family was used on all data sets in this study to predict the direction of the market, but without regard for the magnitude. Analogously to the Gaussian family, the probability parameters may be approximated through the minimisation of the negative *binomial* log-likelihood - given in Equation (12):

$$\rho(Y, f(X)) = -\left[Y \cdot log(\mathbb{P}(Y = 1 \mid X)) + (1 - Y) \cdot log(1 - \mathbb{P}(Y = 1 \mid X))\right] \tag{12}$$

1.6.3 Gamma

This family allows predictions to be made purely of the magnitude of stock market movements, with no regard for the direction. The gamma distribution, implemented as the `GammaReg` family within the `mboost` package, and provides a continuous non-negative response, required for such a model. This function uses the negative gamma-likelihood coupled with the logarithmic link function. For more information on this distribution and estimates of its parameters, refer to [?] and [?].

1.6.4 Inspection within R

Listing 1 illustrates how one may inspect a family contained within the `mboost` package, directly within the R console. On line 1 the `mboost` package is first loaded into the session. The details of the Gaussian family are called up on line 3 and the information about the negative gradient on line 10. Similar operations may be performed for many of the families that the `mboost` package contains.

```
1   R > library(mboost)
2
3   R > Gaussian()
4
5         Squared Error (Regression)
6
7         Loss function: (y - f)^2
8
9
10  R > slot(Gaussian(), "ngradient")
11
12        function (y, f, w = 1)
13
14        y - f
```

Listing 1: An example of how to investigate the properties of an implemented *family* within the `mboost` package - here the example of the Gaussian family.

[19] As a practical note, the `Binomial()` family within the `mboost` package returns values {-1, 1} in its binary response, for reasons of computational efficiency.

/Volumes/Mac OS Drive/Thesis/Source Code/Reporting/nwm_Report/images/gradient_descent_contour_plot.p

Figure 2: A contour plot representing the surface of a simple *fictitious* loss function. The approximation function follows the principle of steepest descent from its (blue) initiation point - crossing each contour line orthogonally - reaching the (red) global minimum at iteration number $\mathrm{m}_{\mathrm{stop}}$.

/Volumes/Mac OS Drive/Thesis/Source Code/Reporting/nwm_Report/images/learning_rate-neg_gradient.png

Figure 3: A loss function for the one-dimensional case; the size of the red points along the curve representing the incremental addition to approximation (prediction) function during gradient descent. Holding ν constant, it is clear that the nominal step-size, $\nu \cdot \left(-\frac{\partial}{\partial f}\rho(Y,f)\right)$, does indeed adapt in size at each iteration.