Chapter 28

Bagging and Random Forest

Random Forest is one of the most popular and most powerful machine learning algorithms. It is a type of ensemble machine learning algorithm called Bootstrap Aggregation or bagging. In this chapter you will discover the Bagging ensemble algorithm and the Random Forest algorithm for predictive modeling. After reading this chapter you will know about:

- The bootstrap method for estimating statistical quantities from samples.
- The Bootstrap Aggregation algorithm for creating multiple different models from a single training dataset.
- The Random Forest algorithm that makes a small tweak to Bagging and results in a very powerful classifier.

Let's get started.

28.1 Bootstrap Method

Before we get to Bagging, let's take a quick look at an important foundation technique called the bootstrap. The bootstrap is a powerful statistical method for estimating a quantity from a data sample. This is easiest to understand if the quantity is a descriptive statistic such as a mean or a standard deviation. Let's assume we have a sample of 100 values (x) and we'd like to get an estimate of the mean of the sample. We can calculate the mean directly from the sample as:

$$mean(x) = \frac{1}{100} \times \sum_{i=1}^{100} x_i \tag{28.1}$$

We know that our sample is small and that our mean has error in it. We can improve the estimate of our mean using the bootstrap procedure:

1. Create many (e.g. 1000) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-sample.

3. Calculate the average of all of our collected means and use that as our estimated mean for the data.

For example, let's say we used 3 resamples and got the mean values 2.3, 4.5 and 3.3. Taking the average of these we could take the estimated mean of the data to be 3.367. This process can be used to estimate other quantities like the standard deviation and even quantities used in machine learning algorithms, like learned coefficients.

28.2 Bootstrap Aggregation (Bagging)

Bootstrap Aggregation (or Bagging for short), is a simple and very powerful ensemble method. An ensemble method is a technique that combines the predictions from multiple machine learning algorithms together to make more accurate predictions than any individual model. Bootstrap Aggregation is a general procedure that can be used to reduce the variance for those algorithms that have high variance. An algorithm that has high variance are decision trees, like classification and regression trees (CART).

Decision trees are sensitive to the specific data on which they are trained. If the training data is changed (e.g. a tree is trained on a subset of the training data) the resulting decision tree can be quite different and in turn the predictions can be quite different. Bagging is the application of the Bootstrap procedure to a high-variance machine learning algorithm, typically decision trees. Let's assume we have a dataset of 1000 instances and we are using the CART algorithm. Bagging of the CART algorithm would work as follows.

1. Create many (e.g. 100) random sub-samples of our dataset with replacement.
2. Train a CART model on each sample.
3. Given a new dataset, calculate the average prediction from each model.

For example, if we had 5 bagged decision trees that made the following class predictions for an input instance: `blue`, `blue`, `red`, `blue` and `red`, we would take the most frequent class and predict `blue`. When bagging with decision trees, we are less concerned about individual trees overfitting the training data. For this reason and for efficiency, the individual decision trees are grown deep (e.g. few training samples at each leaf-node of the tree) and the trees are not pruned. These trees will have both high variance and low bias. These are important characteristics of sub-models when combining predictions using bagging.

The only parameters when bagging decision trees is the number of trees to create. This can be chosen by increasing the number of trees on run after run until the accuracy begins to stop showing improvement (e.g. on a cross validation test harness). Creating large numbers of decision trees may take a long time, but will not overfit the training data. Just like the decision trees themselves, Bagging can be used for classification and regression problems.

28.3 Random Forest

Random Forests are an improvement over bagged decision trees. A problem with decision trees like CART is that they are greedy. They choose which variable to split on using a

greedy algorithm that minimizes error. As such, even with Bagging, the decision trees can have a lot of structural similarities and in turn result in high correlation in their predictions. Combining predictions from multiple models in ensembles works better if the predictions from the sub-models are uncorrelated or at best weakly correlated.

Random forest changes the algorithm for the way that the sub-trees are learned so that the resulting predictions from all of the subtrees have less correlation. It is a simple tweak. In CART, when selecting a split point, the learning algorithm is allowed to look through all variables and all variable values in order to select the most optimal split-point. The random forest algorithm changes this procedure so that the learning algorithm is limited to a random sample of features of which to search. The number of features that can be searched at each split point (m) must be specified as a parameter to the algorithm. You can try different values and tune it using cross validation.

- For classification a good default is: $m = \sqrt{p}$.
- For regression a good default is: $m = \frac{p}{3}$.

Where m is the number of randomly selected features that can be searched at a split point and p is the number of input variables. For example, if a dataset had 25 input variables for a classification problem, then:

$$\begin{aligned} m &= \sqrt{25} \\ m &= 5 \end{aligned} \tag{28.2}$$

28.4 Estimated Performance

For each bootstrap sample taken from the training data, there will be samples left behind that were not included. These samples are called Out-Of-Bag samples or OOB. The performance of each model on its left out samples when averaged can provide an estimated accuracy of the bagged models. This estimated performance is often called the OOB estimate. These performance measures are a reliable estimate of test error and correlate well with cross validation estimates of error.

28.5 Variable Importance

As the Bagged decision trees are constructed, we can calculate how much the error function drops for a variable at each split point. In regression problems this may be the drop in sum squared error and in classification this might be the Gini score. These drops in error can be averaged across all decision trees and output to provide an estimate of the importance of each input variable. The greater the drop when the variable was chosen, the greater the importance.

These outputs can help identify subsets of input variables that may be most or least relevant to the problem and suggest at possible feature selection experiments you could perform where some features are removed from the dataset.

28.6 Preparing Data For Bagged CART

Bagged CART does not require any special data preparation other than a good representation of the problem.

28.7 Summary

In this chapter you discovered the Bagging ensemble machine learning algorithm and the popular variation called Random Forest. You learned:

- How to estimate statistical quantities from a data sample.
- How to combine the predictions from multiple high-variance models using bagging.
- How to tweak the construction of decision trees when bagging to de-correlate their predictions, a technique called Random Forests.

You now know about the bagging ensemble algorithm, bagged decision trees and random forests. In the next chapter you will discover how to implement bagged decision trees from scratch.

Chapter 30

Boosting and AdaBoost

Boosting is an ensemble technique that attempts to create a strong classifier from a number of weak classifiers. In this chapter you will discover the AdaBoost Ensemble method for machine learning. After reading this chapter, you will know:

- What the boosting ensemble method is and generally how it works.
- How to learn to boost decision trees using the AdaBoost algorithm.
- How to make predictions using the learned AdaBoost model.
- How to best prepare your data for use with the AdaBoost algorithm.

Let's get started.

30.1 Boosting Ensemble Method

Boosting is a general ensemble method that creates a strong classifier from a number of weak classifiers. This is done by building a model from the training data, then creating a second model that attempts to correct the errors from the first model. Models are added until the training set is predicted perfectly or a maximum number of models are added. AdaBoost was the first really successful boosting algorithm developed for binary classification. It is the best starting point for understanding boosting. Modern boosting methods build on AdaBoost, most notably stochastic gradient boosting machines.

30.2 Learning An AdaBoost Model From Data

AdaBoost is best used to boost the performance of decision trees on binary classification problems. AdaBoost was originally called AdaBoost.M1 by the developers of the technique. More recently it may be referred to as discrete AdaBoost because it is used for classification rather than regression. AdaBoost can be used to boost the performance of any machine learning algorithm. It is best used with weak learners.

These are models that achieve accuracy just above random chance on a classification problem. The most suited and therefore most common algorithm used with AdaBoost are decision trees with one level. Because these trees are so short and only contain one decision for classification,

they are often called decision stumps. Each instance in the training dataset is weighted. The initial weight is set to:

$$weight(x_i) = \frac{1}{n} \tag{30.1}$$

Where x_i is the i'th training instance and n is the number of training instances.

30.3 How To Train One Model

A weak classifier (decision stump) is prepared on the training data using the weighted samples. Only binary (two-class) classification problems are supported, so each decision stump makes one decision on one input variable and outputs a +1.0 or -1.0 value for the first or second class value. The misclassification rate is calculated for the trained model. Traditionally, this is calculated as:

$$error = \frac{correct - N}{N} \tag{30.2}$$

Where *error* is the misclassification rate, correct are the number of training instance predicted correctly by the model and N is the total number of training instances. For example, if the model predicted 78 of 100 training instances correctly the error or misclassification rate would be $\frac{78-100}{100}$ or 0.22. This is modified to use the weighting of the training instances:

$$error = \frac{\sum_{i=1}^{n}(w_i \times perror_i)}{\sum_{i=1}^{n} w} \tag{30.3}$$

Which is the weighted sum of the misclassification rate, where w is the weight for training instance i and *perror* is the prediction error for training instance i which is 1 if misclassified and 0 if correctly classified. For example, if we had 3 training instances with the weights 0.01, 0.5 and 0.2. The predicted values were -1, -1 and -1, and the actual output variables in the instances were -1, 1 and -1, then the *perror* values would be 0, 1, and 0. The misclassification rate would be calculated as:

$$\begin{aligned} error &= \frac{0.01 \times 0 + 0.5 \times 1 + 0.2 \times 0}{0.01 + 0.5 + 0.2} \\ error &= 0.704 \end{aligned} \tag{30.4}$$

A stage value is calculated for the trained model which provides a weighting for any predictions that the model makes. The stage value for a trained model is calculated as follows:

$$stage = ln(\frac{1 - error}{error}) \tag{30.5}$$

Where stage is the stage value used to weight predictions from the model, $ln()$ is the natural logarithm and error is the misclassification error for the model. The effect of the stage weight is that more accurate models have more weight or contribution to the final prediction. The training weights are updated giving more weight to incorrectly predicted instances, and less weight to correctly predicted instances. For example, the weight of one training instance (w) is updated using:

$$w = w \times e^{stage \times perror} \tag{30.6}$$

Where w is the weight for a specific training instance, e is the numerical constant Euler's number raised to a power, stage is the misclassification rate for the weak classifier and *perror* is the error the weak classifier made predicting the output variable for the training instance, evaluated as:

- perror = 0 **IF** y == p
- perror = 1 **IF** y != p

Where y is the output variable for the training instance and p is the prediction from the weak learner. This has the effect of not changing the weight if the training instance was classified correctly and making the weight slightly larger if the weak learner misclassified the instance.

30.4 AdaBoost Ensemble

Weak models are added sequentially, trained using the weighted training data. The process continues until a pre-set number of weak learners have been created (a user parameter) or no further improvement can be made on the training dataset. Once completed, you are left with a pool of weak learners each with a stage value.

30.5 Making Predictions with AdaBoost

Predictions are made by calculating the weighted average of the weak classifiers. For a new input instance, each weak learner calculates a predicted value as either +1.0 or -1.0. The predicted values are weighted by each weak learners stage value. The prediction for the ensemble model is taken as a the sum of the weighted predictions. If the sum is positive, then the first class is predicted, if negative the second class is predicted.

For example, 5 weak classifiers may predict the values 1.0, 1.0, -1.0, 1.0, -1.0. From a majority vote, it looks like the model will predict a value of 1.0 or the first class. These same 5 weak classifiers may have the stage values 0.2, 0.5, 0.8, 0.2 and 0.9 respectively. Calculating the weighted sum of these predictions results in an output of -0.8, which would be an ensemble prediction of -1.0 or the second class.

30.6 Preparing Data For AdaBoost

This section lists some heuristics for best preparing your data for AdaBoost.

- **Quality Data**: Because the ensemble method continues to attempt to correct misclassification's in the training data, you need to be careful that the training data is of a high-quality.
- **Outliers**: Outliers will force the ensemble down the rabbit hole of working hard to correct for cases that are unrealistic. These could be removed from the training dataset.
- **Noisy Data**: Noisy data, specifically noise in the output variable can be problematic. If possible, attempt to isolate and clean these from your training dataset.

30.7 Summary

In this chapter you discovered the Boosting ensemble method for machine learning. You learned about:

- Boosting and how it is a general technique that keeps adding weak learners to correct classification errors.
- AdaBoost as the first successful boosting algorithm for binary classification problems.
- Learning the AdaBoost model by weighting training instances and the weak learners themselves.
- Predicting with AdaBoost by weighting predictions from weak learners.
- Where to look for more theoretical background on the AdaBoost algorithm.

You now know about the boosting ensemble method and the AdaBoost machine learning algorithm. In the next chapter you will discover how to implement the AdaBoost algorithm from scratch.