13 | Ensemble Methods

> This is the central illusion in life: that randomness is a risk, that it is a bad thing. . .
> – Nassim Nicholas Taleb

Learning Objectives:

- Implement bagging and explain how it reduces variance in a predictor.
- Explain the difference between a weak learner and a strong learner.
- Derive the AdaBoost algorithm.
- Understand the relationship between boosting decision stumps and linear classification.

Dependencies:

Groups of people can often make better decisions than individuals, especially when group members each come in with their own biases. The same is true in machine learning. Ensemble methods are learning models that achieve performance by combining the opinions of multiple learners. In doing so, you can often get away with using much *simpler* learners and still achieve great performance. Moreover, ensembles are inherantly parallel, which can make them much more efficient at training and test time, if you have access to multiple processors.

In this chapter, you will learn about various ways of combining **base learners** into **ensembles**. One of the shocking results we will see is that you can take a learning model that only ever does slightly better than chance, and turn it into an arbitrarily good learning model, though a technique known as **boosting**. You will also learn how ensembles can decrease the variance of predictors as well as perform regularization.

13.1 Voting Multiple Classifiers

All of the learning algorithms you have seen so far are deterministic. If you train a decision tree multiple times on the same data set, you will always get the same tree back. In order to get an effect out of voting multiple classifiers, they need to differ. There are two primary ways to get variability. You can either change the learning algorithm or change the data set.

Building an emsemble by training different classifiers is the most straightforward approach. As in single-model learning, you are given a data set (say, for classification). Instead of learning a single classifier (e.g., a decision tree) on this data set, you learn multiple different classifiers. For instance, you might train a decision tree, a perceptron, a KNN, and multiple neural networks with different architectures. Call these classifiers $f_1, \ldots, f_M$. At test time, you can make a prediction by *voting*. On a test example $\hat{x}$, you compute $\hat{y}_1 = f_1(\hat{x}), \ldots,$

$\hat{y}_M = f_M(\hat{x})$. If there are more +1s in the list $\langle y_1, \ldots, y_M$ then you predict +1; otherwise you predict −1.

The main advantage of ensembles of different classifiers is that it is unlikely that all classifiers will make the same mistake. In fact, as long as every error is made by a minority of the classifiers, you will achieve optimal classification! Unfortunately, the inductive biases of different learning algorithms are highly correlated. This means that different algorithms are prone to similar types of errors. In particular, ensembles tend to reduce the **variance** of classifiers. So if you have a classification algorithm that tends to be very sensitive to small changes in the training data, ensembles are likely to be useful.

? Which of the classifiers you've learned about so far have high variance?

Note that the voting scheme naturally extends to multiclass classification. However, it does not make sense in the contexts of regression, ranking or collective classification. This is because you will rarely see the same exact output predicted twice by two different regression models (or ranking models or collective classification models). For regression, a simple solution is to take the *mean* or *median* prediction from the different models. For ranking and collective classification, different approaches are required.

Instead of training different types of classifiers on the same data set, you can train a single type of classifier (e.g., decision tree) on multiple data sets. The question is: where do these multiple data sets come from, since you're only given one at training time?

One option is to fragment your original data set. For instance, you could break it into 10 pieces and build decision trees on each of these pieces individually. Unfortunately, this means that each decision tree is trained on only a very small part of the entire data set and is likely to perform poorly.

A better solution is to use **bootstrap resampling**. This is a technique from the statistics literature based on the following observation. The data set we are given, D, is a sample drawn i.i.d. from an unknown distribution $\mathcal{D}$. If we draw a *new* data set $\tilde{D}$ by random sampling from D with replacement[1], then $\tilde{D}$ is *also* a sample from $\mathcal{D}$. Figure 13.1 shows the process of bootstrap resampling of ten objects.

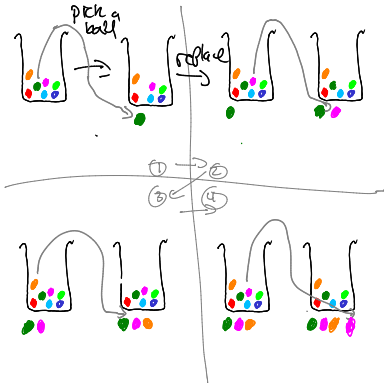

Figure 13.1: picture of sampling with replacement

[1] To sample with replacement, imagine putting all items from D in a hat. To draw a single sample, pick an element at random from that hat, write it down, and then *put it back*.

Applying this idea to ensemble methods yields a technique known as **bagging**. You start with a single data set D that contains N training examples. From this single data set, you create M-many "bootstrapped training sets" $\tilde{D}_1, \ldots \tilde{D}_M$. Each of these bootstrapped sets also contains N training examples, drawn randomly from D with replacement. You can then train a decision tree (or other model) seperately on each of these data sets to obtain classifiers $f_1, \ldots, f_M$. As before, you can use these classifiers to vote on new test points.

Note that the bootstrapped data sets will be similar. However, they will not be *too* similar. For example, if N is large then the number of

examples that are not present in any particular bootstrapped sample is relatively large. The probability that the first training example is not selected once is $(1 - 1/N)$. The probability that it is not selected at all is $(1 - 1/N)^N$. As $N \to \infty$, this tends to $1/e \approx 0.3679$. (Already for $N = 1000$ this is correct to four decimal points.) So only about 63% of the original training examples will be represented in any given bootstrapped set.

Since bagging tends to reduce *variance*, it provides an alternative approach to regularization. That is, even if each of the learned classifiers $f_1, \ldots, f_M$ are individually overfit, they are likely to be overfit to different things. Through voting, you are able to overcome a significant portion of this overfitting. Figure 13.2 shows this effect by comparing regularization via hyperparameters to regularization via bagging.

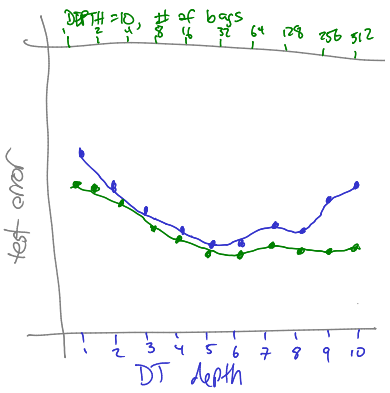

Figure 13.2: graph depicting overfitting using regularization versus bagging

13.2 Boosting Weak Learners

Boosting is the process of taking a crummy learning algorithm (technically called a **weak learner**) and turning it into a great learning algorithm (technically, a **strong learner**). Of all the ideas that originated in the theoretical machine learning community, boosting has had—perhaps—the greatest practical impact. The idea of boosting is reminiscent of what you (like me!) might have thought when you first learned about file compression. If I compress a file, and then re-compress it, and then re-compress it, eventually I'll end up with a final that's only one byte in size!

To be more formal, let's define a **strong learning algorithm** $\mathcal{L}$ as follows. When given a desired error rate ϵ, a failure probability δ and access to "enough" labeled examples from some distribution $\mathcal{D}$, then, with high probability (at least $1 - \delta$), $\mathcal{L}$ learns a classifier f that has error at most ϵ. This is precisely the definition of **PAC** learning that you learned about in Chapter 12. Building a strong learning algorithm might be difficult. We can as if, instead, it is possible to build a **weak learning algorithm** $\mathcal{W}$ that only has to achieve an error rate of 49%, rather than some arbitrary user-defined parameter ϵ. (49% is arbitrary: anything strictly less than 50% would be fine.)

Boosting is more of a "framework" than an algorithm. It's a framework for taking a weak learning algorithm $\mathcal{W}$ and turning it into a strong learning algorithm. The particular boosting algorithm discussed here is **AdaBoost**, short for "adaptive boosting algorithm." AdaBoost is famous because it was one of the first *practical* boosting algorithms: it runs in polynomial time and does not require you to define a large number of hyperparameters. It gets its name from the latter benefit: it automatically *adapts* to the data that you give it.

Algorithm 32 AdaBoost($\mathcal{W}$, $\mathcal{D}$, K)

$\boldsymbol{d}^{(0)} \leftarrow \langle \frac{1}{N}, \frac{1}{N}, \ldots, \frac{1}{N} \rangle$ // Initialize uniform importance to each example
for $k = 1 \ldots K$ **do**
  $f^{(k)} \leftarrow \mathcal{W}(\mathcal{D}, \boldsymbol{d}^{(k-1)})$ // Train kth classifier on weighted data
  $\hat{y}_n \leftarrow f^{(k)}(\boldsymbol{x}_n), \forall n$ // Make predictions on training data
  $\hat{\epsilon}^{(k)} \leftarrow \sum_n d_n^{(k-1)} [y_n \neq \hat{y}_n]$ // Compute weighted training error
  $\alpha^{(k)} \leftarrow \frac{1}{2} \log \left(\frac{1-\hat{\epsilon}^{(k)}}{\hat{\epsilon}^{(k)}} \right)$ // Compute "adaptive" parameter
  $d_n^{(k)} \leftarrow \frac{1}{Z} d_n^{(k-1)} \exp[-\alpha^{(k)} y_n \hat{y}_n], \forall n$ // Re-weight examples and normalize
end for
return $f(\hat{\boldsymbol{x}}) = \operatorname{sgn}\left[\sum_k \alpha^{(k)} f^{(k)}(\hat{\boldsymbol{x}})\right]$ // Return (weighted) voted classifier

The intuition behind AdaBoost is like studying for an exam by using a past exam. You take the past exam and grade yourself. The questions that you got right, you pay less attention to. Those that you got *wrong*, you study more. Then you take the exam again and repeat this process. You continually *down-weight* the importance of questions you routinely answer correctly and *up-weight* the importance of questions you routinely answer incorrectly. After going over the exam multiple times, you hope to have mastered everything.

The precise AdaBoost training algorithm is shown in Algorithm 13.2. The basic functioning of the algorithm is to maintain a *weight distribution* $\boldsymbol{d}$, over data points. A weak learner, $f^{(k)}$ is trained on this weighted data. (Note that we implicitly assume that our weak learner can accept weighted training data, a relatively mild assumption that is nearly always true.) The (weighted) error rate of $f^{(k)}$ is used to determine the *adaptive parameter* α, which controls how "important" $f^{(k)}$ is. As long as the weak learner does, indeed, achieve $< 50\%$ error, then α will be greater than zero. As the error drops to zero, α grows without bound.

? What happens if the weak learning assumption is violated and $\hat{\epsilon}$ is equal to 50%? What if it is worse than 50%? What does this mean, in practice?

After the adaptive parameter is computed, the weight distibution is updated for the next iteration. As desired, examples that are correctly classified (for which $y_n \hat{y}_n = +1$) have their weight *decreased* multiplicatively. Examples that are incorrectly classified ($y_n \hat{y}_n = -1$) have their weight *increased* multiplicatively. The Z term is a normralization constant to ensure that the sum of $\boldsymbol{d}$ is one (i.e., $\boldsymbol{d}$ can be interpreted as a distribution). The final classifier returned by AdaBoost is a weighted vote of the individual classifiers, with weights given by the adaptive parameters.

To better understand why α is defined as it is, suppose that our weak learner simply returns a *constant* function that returns the (weighted) majority class. So if the total weight of positive examples exceeds that of negative examples, $f(\boldsymbol{x}) = +1$ for all $\boldsymbol{x}$; otherwise $f(\boldsymbol{x}) = -1$ for all $\boldsymbol{x}$. To make the problem moderately interesting, suppose that in the original training set, there are 80 positive ex-

amples and 20 negative examples. In this case, $f^{(1)}(\boldsymbol{x}) = +1$. It's weighted error rate will be $\hat{\epsilon}^{(1)} = 0.2$ because it gets every negative example wrong. Computing, we get $\alpha^{(1)} = \frac{1}{2}\log 4$. Before normalization, we get the new weight for each positive (correct) example to be $1\exp[-\frac{1}{2}\log 4] = \frac{1}{2}$. The weight for each negative (incorrect) example becomes $1\exp[\frac{1}{2}\log 4] = 2$. We can compute $Z = 80{\times}\frac{1}{2} + 20{\times}2 = 80$. Therefore, after normalization, the weight distribution on any single positive example is $\frac{1}{160}$ and the weight on any negative example is $\frac{1}{40}$. However, since there are 80 positive examples and 20 negative examples, the *cumulative* weight on all positive examples is $80{\times}\frac{1}{160} = \frac{1}{2}$; the cumulative weight on all negative examples is $20{\times}\frac{1}{40} = \frac{1}{2}$. Thus, after a single boosting iteration, the data has become precisely evenly weighted. This guarantees that in the next iteration, our weak learner *must* do something more interesting than majority voting if it is to achieve an error rate *less than* 50%, as required.

? This example uses concrete numbers, but the same result holds no matter what the data distribution looks like nor how many examples there are. Write out the general case to see that you will still arrive at an even weighting after one iteration.

One of the major attractions of boosting is that it is perhaps easy to design computationally efficient weak learners. A very popular type of weak learner is a **shallow decision tree**: a decision tree with a small depth limit. Figure 13.3 shows test error rates for decision trees of different maximum depths (the different curves) run for differing numbers of boosting iterations (the x-axis). As you can see, if you are willing to boost for many iterations, very shallow trees are quite effective.

In fact, a very popular weak learner is a decision **decision stump**: a decision tree that can only ask *one* question. This may seem like a silly model (and, in fact, it is on it's own), but when combined with boosting, it becomes very effective. To understand why, suppose for a moment that our data consists only of binary features, so that any question that a decision tree might ask is of the form "is feature 5 on?" By concentrating on decision stumps, all weak functions must have the form $f(\boldsymbol{x}) = s(2x_d - 1)$, where $s \in \{\pm 1\}$ and d indexes *some* feature.

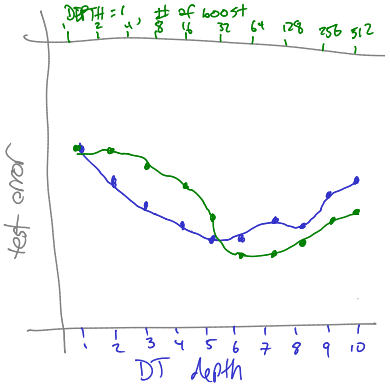

Figure 13.3: perf comparison of depth vs # boost

? Why do the functions have this form?

Now, consider the *final* form of a function learned by AdaBoost. We can expand it as follow, where we let f_k denote the single feature selected by the kth decision stump and let s_k denote its sign:

$$f(\boldsymbol{x}) = \operatorname{sgn}\left[\sum_k \alpha_k f^{(k)}(\boldsymbol{x})\right] \tag{13.1}$$

$$= \operatorname{sgn}\left[\sum_k \alpha_k s_k (2x_{f_k} - 1)\right] \tag{13.2}$$

$$= \operatorname{sgn}\left[\sum_k 2\alpha_k s_k x_{f_k} - \sum_k \alpha_k s_k\right] \tag{13.3}$$

$$= \operatorname{sgn}\left[\boldsymbol{w}\cdot\boldsymbol{x} + b\right] \tag{13.4}$$

Algorithm 33 RANDOMFORESTTRAIN($\mathcal{D}$, *depth*, K)

for $k = 1 \dots K$ **do**
  $t^{(k)} \leftarrow$ complete binary tree of depth *depth* with random feature splits
  $f^{(k)} \leftarrow$ the function computed by $t^{(k)}$, with leaves filled in by $\mathcal{D}$
end for
return $f(\hat{\boldsymbol{x}}) = \text{sgn}\left[\sum_k f^{(k)}(\hat{\boldsymbol{x}})\right]$ // Return voted classifier

$$\text{where } w_d = \sum_{k:f_k=d} 2\alpha_k s_k \quad \text{and} \quad b = -\sum_k \alpha_k s_k \tag{13.5}$$

Thus, when working with decision stumps, AdaBoost actually provides an algorithm for learning **linear classifiers**! In fact, this connection has recently been strengthened: you can show that AdaBoost provides an algorithm for optimizing **exponential loss**. (However, this connection is beyond the scope of this book.)

As a further example, consider the case of boosting a **linear classifier**. In this case, if we let the kth weak classifier be parameterized by $\boldsymbol{w}^{(k)}$ and $b^{(k)}$, the overall predictor will have the form:

$$f(\boldsymbol{x}) = \text{sgn}\left[\sum_k \alpha_k \text{sgn}\left(\boldsymbol{w}^{(k)} \cdot \boldsymbol{x} + b^{(k)}\right)\right] \tag{13.6}$$

You can notice that this is nothing but a two-layer **neural network**, with K-many hidden units! Of course it's not a classifically trained neural network (once you learn $\boldsymbol{w}^{(k)}$ you never go back and update it), but the structure is identical.

13.3 Random Ensembles

One of the most computationally expensive aspects of ensembles of decision trees is training the decision trees. This is very fast for decision stumps, but for deeper trees it can be prohibitively expensive. The expensive part is choosing the tree structure. Once the tree structure is chosen, it is very cheap to fill in the leaves (i.e., the predictions of the trees) using the training data.

An efficient and surprisingly effective alternative is to use trees with fixed structures and random features. Collections of trees are called forests, and so classifiers built like this are called **random forests**. The random forest training algorithm, shown in Algorithm 13.3 is quite short. It takes three arguments: the data, a desired depth of the decision trees, and a number K of total decision trees to build.

The algorithm generates each of the K trees independently, which makes it very easy to parallelize. For each trees, it constructs a full binary tree of depth *depth*. The features used at the branches of this

tree are selected randomly, typically *with replacement*, meaning that the same feature can appear multiple times, even in one branch. The leaves of this tree, where predictions are made, are filled in based on the training data. This last step is the *only* point at which the training data is used. The resulting classifier is then just a voting of the K-many random trees.

The most amazing thing about this approach is that it actually works remarkably well. It tends to work best when all of the features are at least marginally relevant, since the number of features selected for any given tree is small. An intuitive reason that it works well is the following. Some of the trees will query on useless features. These trees will essentially make random predictions. But some of the trees will happen to query on good features and will make good predictions (because the leaves are estimated based on the training data). If you have enough trees, the random ones will wash out as noise, and only the good trees will have an effect on the final classification.

13.4 Further Reading

TODO further reading