Ensemble Methods

CS 360 Machine Learning
Week 4, Day 1

February 13, 2024

Contents

1 Ensemble methods

Recall that the bias-variance tradeoff is when increases to model complexity (or flexibility) decrease bias but increase variance. We want to be at the balance point, when measured on the test set, between bias and variance. Ideally, we want to be able to use complex models without increasing the variance. One option to help handle this tradeoff is to use ensemble methods that we'll discuss today.

Models with low bias are generally simpler models, so we're going to try to use multiple simple classifiers to also reduce the variance. It's a bit like cross validation in that we'll be making a prediction for each model and then vote or otherwise combine the multiple classifiers created. That's the basic idea of an ensemble: taking multiple simpler models and then combining the results. It'll be important to make sure that we don't have multiple models that are wrong in the same way, since then the combined model will also be wrong in the same way. We'll take advantage of the fact that if n observations each have variance s^2, then the mean of the observations has variance $\frac{s^2}{n}$; the variance is reduced by averaging!

Suppose that we have a hypothesis space H. When considering classifiers, there are three mains sources of limitations. One is statistical - the hypothesis space is too large relative to the size of the data so we can't explore the space well based on the data we have. Another limit is computational - H is too large and we face complexity limits to finding the best model. And the last is representational - H is not expressive enough to fully represent the phenomena of interest.

Ensemble methods can help us with all of these concerns; the average of unstable models has more stability, searching from multiple points is better than one starting point, and the sum of many models can represent more hypotheses than an individual model. Another way to think about this is that we can imagine that we're finding samples from around the space near the true model where if we average them we can get closer to the true model, even if the best model for a phenomena is not sampled directly, or even is outside the hypothesis space of sampled models. Ensemble models have been very successful and practically solve many real world problems.

1.1 Bagging

First let's talk about the bagging algorithm approach to ensemble methods; this stands for *bootstrap aggregation* which is a basic technique using a *bootstrap* sample (randomly sampling with replacement) from the original data to create many different training sets. Many ensemble methods use bootstrapping as part of the technique. The bootstrap sample is generally the same size as the original sample; the training data set of n is replaced with a new training set of size n sampled from the original data. What happens if we get the same data set back? Suppose we ran a new learning algorithm on this same set – we wouldn't gain anything over learning on the original training data.

So what does the bootstrap sample gain us? Suppose we have some original training data with n examples. We can use bootstrapping to create artificial training datasets for each of multiple separate algorithms we'll run, where we'll use datasets of the same size but where some of the examples from the original set are included in these new training sets, some examples are included multiple times, and some examples are not included at all. Each of these new training sets would also have n examples. Then we can run the learning algorithm on each of these new training sets separately. These would each create separate hypothesis models $h_1, h_2, ...$ This will give us an ensemble of hypothesis models. Note that these models will all be trained based on the same algorithm and hyperparameters.

It might be concerning that we could get back essentially the same dataset each time. As n becomes large, this is effectively not the case and becomes less of a problem. Why don't we need to worry about this? What is the probability that I *don't* choose a specific example n_i from the dataset on the first choice of an example for a training dataset? $\frac{n-1}{n}$. Now what if I did that n times to compose a full sampled training set? Consider what the probability is overall that an example isn't included. (See handout question 1.) Since this is sampling with replacement, this total probability is

$$\left(\frac{n-1}{n}\right)^n$$

As the training set size n grows large (i.e., $\lim_{n\to\infty}$) we find that this probability that an example is *not* included in the training set converges to:

$$\left(1-\frac{1}{n}\right)^n \approx e^{-1} \approx 0.37$$

So we see that about 37% of examples will not be included in your bootstrap sample. This is a fairly high percentage of examples not included in a single training set! But recall that we're creating multiple training samples. This gives us confidence that we're getting diverse subsets of our data for each training set, which will lead to different hypothesis models, which will hopefully reduce the variance when we combine these hypothesis models for our ensemble model.

One way to think about the multiple sampling of specific training examples is weighting those examples more heavily — we'll talk about this approach more when we talk about boosting.

So what do ensemble methods actually look like? We'll consider an analysis for binary classification. We think about T as the number of models or classifiers in the ensemble (indexed by t). T should be fairly large. $X^{(t)}$ denotes the bootstrap training set t, $h^{(t)}$ denotes the hypothesis model created from $X^{(t)}$, and $h^{(t)}(\vec{x})$ denotes the hypothesis about x from that model. We'll analyze these two probabilities: r, the probability of error of an individual model and R, the number of votes for the wrong class.

A bagging (bootstrap aggregation) training phase operates as follows:

```
for t in T:
    create bootstrap dataset X^(t)
    train on X^(t) to get h^(t)
```

The testing phase then operates as follows:

```
for x in test data:
    h(x) is our overall prediction
    h(x) = argmax_{y∈{0,1}} Σ_{t=1}^{T} 𝟙 (h^(t)(x) = y)
```

where $\mathbb{1}$ is our indicator function. This essentially just says that we're going to vote based on the 0 or 1 labels, more votes for 0 leads to a classification of 0 (and the same for more votes for 1). (If you had more than two classes you could still do a majority vote, but we'll focus on two classes here since right / wrong is more straightforward and easy to think about, but it's essentially the same idea.)

Let's see how this could improve our results. Recall that r is the probability that the base classifier is wrong and R is the number of votes for the wrong class. I can think about what the probabilities are of these two values. Consider $P(R = k)$, i.e., the probability that I have k votes for the wrong class. That's

$$P(R = k) \;=\; (k^T)r^k(1-r)^{T-k}$$

where r^k gives the probability of being wrong k times while $(1-k)^{t-k}$ is the probability of the classifier being right $T-k$ times overall. This follows a binomial distribution (giving the k^T term).

But what I care about is not a specific k, but rather the overall amount I'm wrong:

$$P(\text{overall wrong}) = P\left(R > \frac{T}{2}\right)$$

$$= \sum_{k=\frac{T+1}{2}}^{T} \binom{T}{k} r^k(1-r)^{T-k}$$

Suppose that I had three classifiers ($T = 3$), $h^{(1)}, h^{(2)}, h^{(3)}$, and the true label is 1. Suppose that $h^{(1)}$ predicted 1, $h^{(2)}$ predicted 0, and $h^{(3)}$ predicted 1 — what would the vote (and final prediction) be in this case? It would be 1, which is correct. What if instead I had a situation where two of the models predicted 0 — then the final predicted label would be wrong. In that case I would have $k = 2$, i.e., two incorrect votes from the underlying hypothesis models. Using the probabilities that each of these underlying hypothesis models are incorrect we can calculate the probability the overall ensemble model is wrong. (See an example in question 2 of the handout.) Let's try to interpret the result from the handout exercise. We can compare it to the original r which was 0.25 in that example and notice that the final ensemble error is 0.16. Look - the error went down! That's a large improvement, even with three models as part of the ensemble. And in a real-world example, we would likely use more models, driving the error down further.

1.2 Random forests

The basic idea of random forests model is to use bootstrap sampling but add in further diversity in our models to help reduce the correlation between the models that make up the ensemble. We can do this by choosing a different subset of features for each classifiers (i.e., not just different training examples, but also different features). We also use decision stumps (trees with a depth of 1). In practice, we often choose $\sqrt{p}$ features without replacement for each model. We still use information gain or our other usual methods for selecting the features at each level. Overall, the idea of random forests is to make two key modifications to bagging algorithms: using decision stumps and restricting the number of features we use at each level to give model diversity. We'll come back and see how random forests can be used in the context of AdaBoost and ensemble methods later.

1.3 Ada Boost (adaptive boosting)

This is one of my favorite algorithms. What's the idea of boosting? Our goal is to boost the weight of certain examples in order to get them right. We choose these examples based on which examples we got wrong in a previous iteration so that we'll focus more on them, and hopefully get them right, in the next iteration. This focuses our attention on our errors. And then we average over multiple iterations. Let's make that idea more precise.

When training, we begin with equal weights on all examples. We'll do T iterations (these will be our T classifiers for the ensemble method). For each of these iterations, we'll learn a classifier on the weighted examples from the training dataset. (Note that these are not bootstrap sampled examples; here we're still using the full training set but weighting the examples differently.) We'll change example weights based on the *training* error. Finally, we test and get predictions from each of the T classifiers (for a total of T predictions). Our final classification will weight each of the T models in a vote based on how well it performed during training; each classifier receives a score. Since this is an iterative training algorithm where each round is based on the previous round, it allows us to be *adaptive* if we found that a previous round of the training didn't perform well.

Let's see how this works in practice. We start by setting the weights of each example in our first training iteration to be $\frac{1}{n}$ so that each example starts with an equal weight. At each iteration, the sum of the weights should be 1. Note that we'll need to find a way for each of our classifiers (decision trees) to cope with weighted examples. We'll fit the classifier to the weighted training set.

1.3.1 AdaBoost Algorithm Overview

Input

A set of n training examples, each with p features (represented by matrix $\boldsymbol{X}$), and a vector of labels $\boldsymbol{y}$ where each $y \in \{-1, 1\}$ (binary classification). We also need to choose a family of classifiers (i.e. decision stumps) that will work with *weighted* training examples, and a number of iterations T.

Initialization

- Assign uniform weights to all training data points: $w_i^{(1)} = \frac{1}{n}$ for $i = 1, 2, \cdots, n$. Note that we require the weights to sum to 1.

Adaptive Procedure

For $t = 1, 2, \cdots T$, use the following procedure to find a new classifier and update the weights on the training examples:

(a) Fit a classifier to the weighted training set. We will call this classifier $h^{(t)}(\boldsymbol{x})$.

(b) Compute weighted classification *error* on the training set:

$$\varepsilon = \sum_{i=1}^{n} w_i^{(t)} \mathbb{1}\left(y_i \neq h^{(t)}(\boldsymbol{x}_i)\right)$$

Note that since the weights sum to 1, $0 \leq \varepsilon_t \leq 1$. However, since we are in a binary classification scenario, we should never have an error greater than 0.5.

(c) Compute the *score* of the classifier:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

The score is 0 when $\varepsilon_t = \frac{1}{2}$ (random guessing). As $\varepsilon_t \to 0$, $\alpha_t \to \infty$ (i.e. a very good classifier).

This provides the basic setup for the algorithm. We'll finish the algorithm, computing the new weights and discussing testing, next class.