Week 7: Model Ensembles

Instructor: Sergey Levine

1 Ensembles

So far, every time we've faced a classification problem where the decision boundary is complex, our solution has been to introduce more expressive learning methods: we can add features to a linear classifier, use decision trees, use SVMs with kernels, or try neural networks. The trouble is that the more expressive classifiers have a tendency to overfit more, and introduce considerable complexity.

In this unit, we'll discuss an alternative approach, where instead of designing a better classifier, we can instead ensemble together multiple classifiers to get a better solution. Imagine that we need to fit a decision tree to a 2D dataset where the positive examples are all inside a circle, and the negative examples are all outside (see lecture slides). A decision tree that always splits along one of the two axes will always produce a jagged and inaccurate fit, until it has so many leaves that it starts to overfit.

What if we instead fit many small decision trees and have them vote? We need each decision tree to be responsible for different parts of the data (or else they'll all vote exactly the same way), but if we can get them to each specialize to some part of the space, we can simply take the fraction of classifiers that vote affirmatively as an estimate of the probability that a given point is positive ($y = +1$) or negative ($y = -1$). We'll cover two ensemble methods: bagging and boosting. Both have the same structure: they incrementally add additional classifiers to the ensemble, and then have them vote to determine the label for a new datapoint. Here is an example structure of an ensemble method:

Algorithm 1 Ensemble method

```
1: for $t$ in $1, \ldots, T$ (to create ensemble with $T$ classifiers) do
2:     Create dataset $\mathcal{D}_t$ by reweighting the full training set $\mathcal{D}$
3:     Train hypothesis $h_t$ by minimizing error or maximizing likelihood of $\mathcal{D}_t$
4:     Put a weight $\alpha_t$ on $h_t$
5: end for
6: Final classifier is given by $H(\mathbf{x}) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}))$
```

2 Bagging

Bagging stands for "bootstrap aggregation," and corresponds to a very simple ensemble method. In bagging, in order to get the different classifiers to learn slightly different parts of the data, we simply resample the dataset randomly with replacement, and train each weak classifier on a different resampling of the data. Randomly resampling the dataset simply puts integer weights on each datapoint. Specifically, in bagging, we resample with replacement, which means that for each $i \in \{1, 2, \dots, N\}$, we choose a random index $j \in \{1, 2, \dots, N\}$ and set the i^{th} datapoint in $\mathcal{D}_t$ to be the j^{th} datapoint in $\mathcal{D}$. Some datapoints will be picked more than once, and some might never be picked, so each datapoint gets an integer weight.

Once we've trained K classifiers on these random resamplings of the data, we can produce a single answer simply by averaging together their votes (so $\alpha_t = 1/T$). We can also conveniently use the fraction of classifiers that vote for one class as a probability score for that class, providing a natural quantification of uncertainty. Regions where all classifiers agree are high-confidence classifications, while regions where many disagree end up fuzzier. So the bagging algorithm looks like this:

Algorithm 2 Bagging

```
1: for $t$ in $1, \dots, T$ (to create ensemble with $T$ classifiers) do
2:     Create dataset $\mathcal{D}_t$ by sampling $N$ points from $\mathcal{D}$ with replacement
3:     Train hypothesis $h_t$ by minimizing error or maximizing likelihood of $\mathcal{D}_t$
4:     Put a weight $\alpha_t = 1/K$ on $h_t$
5: end for
6: Final classifier is given by $H(\mathbf{x}) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}))$
```

3 Sidenote: random forests

One very effective variant of bagging combines bagging with decision trees. This is called random forests. Random forests are bagged decision trees, typically with a limit on their maximum depth for simplicity. However, even with bagging, the different trees might still end up too similar. A simple heuristic is to also randomly sample the features at each split of the tree, to prevent all trees from splitting on the same highly informative feature. So instead of choosing the optimal split, we randomly sample a subset of the features, and choose the optimal split among those (a good choice for the number of features is $\sqrt{K}$, where K is the number of features). An even more extreme example is called extremely randomized trees: these simply sample 1 of the K features, instead of using a criterion like information gain. Because many trees get to vote (typically hundreds or thousands), even these very weak learners tend to perform well.

4 Boosting

Before describing boosting, let's step back and revisit the idea of a weak classifier (or "weak learner" as it is more commonly called in the context of ensemble methods). A weak learner is a learner with a lot of bias: a simple learner like logistic regression with only linear features, naïve Bayes, or a very shallow decision tree (or even a "decision stump" – a tree with just one decision). We like weak learners because they usually don't overfit and are often very quick and easy to implement. But we also don't want to use them all the time, because they can't learn very complex functions. So the purpose of an ensemble method is to allow weak learners to learn complex functions by working together. Before we go further, we need a more formal definition of a weak learner. For the purpose of this discussion, we'll define a weak learner as a classifier $h(\mathbf{x})$ that outputs $+1$ if it predicts the label is positive and -1 if it predicts the label is negative. The weak learner must be a learner: that is, it must be able to learn at least better than random guessing. This means that we must have:

$$\sum_{i=1}^{N} y^i h(\mathbf{x}^i) > 0.$$

A weak learner that satisfies these conditions can be used for boosting, which we'll describe next.

Boosting is a more sophisticated ensemble method where, instead of resampling the data randomly for each classifier, we'll instead reweight the data based on the mistakes of past classifiers: that is, for the k^{th} classifier, we'll ask it to prioritize getting the right answer for exactly those datapoints that previous classifiers got wrong. We'll then weight the classifier by its performance, so that classifiers that do better get more "votes." The specific method we'll cover is called AdaBoost. See the lecture notes for a visual illustrate of multiple rounds of boosting.

AdaBoost has the same structure as bagging, but a different choice for the datapoint weights and classifier weights:

Algorithm 3 AdaBoost

```
1:  for $t$ in $1, \ldots, T$ (to create ensemble with $T$ classifiers) do
2:    if $t = 1$ then
3:      Initialize weights to $D_{1,i} = 1/N$
4:    else
5:      Set weights $D_{t,i} \propto D_{t-1,i} \exp(-\alpha_{t-1} y^i h_{t-1}(\mathbf{x}^i))$
6:    end if
7:    Train hypothesis $h_t$ by minimizing error $\mathcal{D}$ weighted by $D_t$
8:    Evaluate weighted error $\epsilon_t = \sum_{i=1}^{N} D_{t,i} \delta(h_t(\mathbf{x}^i) \neq y^i)$
9:    Put a weight $\alpha_t = \frac{1}{2} \ln\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$ on $h_t$
10: end for
11: Final classifier is given by $H(\mathbf{x}) = \text{sign}(\sum_{t=1}^{T} \alpha_t h_t(\mathbf{x}))$
```

There are two decisions for AdaBoost that we need to analyze: the choice of weight update for weights D_t and the choice of classifier weight α_t. But first, let's briefly go over what it means to train a classifier on a weighted dataset. In the discussion of bagging, we saw how weighting datapoints by integer values corresponds simply to training a classifier on a new dataset where some points occur more than once. In general, the counts on each datapoint do not have to be integers. How weights enter into different learning algorithms varies, but it's typically straightforward: in naïve Bayes, instead of evaluating counts for estimating conditional distributions, we evaluate weighted counts. In logistic regression, we weight the gradient of each point by its weight. In decision trees, we weight the information gain by the weight on each datapoint.

5 Boosting: intuition

First, let's examine the AdaBoost weight update:

$$D_{t+1,i} \propto D_{t,i} \exp(-\alpha_t y^i h_t(\mathbf{x}^i)).$$

That means that

$$D_{t+1,i} = \frac{D_{t,i} \exp(-\alpha_t y^i h_t(\mathbf{x}^i))}{\sum_{i'=1}^N D_{t,i'} \exp(-\alpha_t y^{i'} h_t(\mathbf{x}^{i'}))} = \frac{1}{Z_t} D_{t,i} \exp(-\alpha_t y^i h_t(\mathbf{x}^i)),$$

where we've defined $Z_t = \sum_{i'=1}^N D_{t,i'} \exp(-\alpha_t y^{i'} h_t(\mathbf{x}^{i'}))$. Note that we have $y^i h_t(\mathbf{x}^i) = 1$ if h_t classifies $\mathbf{x}^i$ correctly, and -1 otherwise. Since we negate this quantity, the weight update increases the weight on incorrectly classified datapoints, and decreases the weight on correctly classified datapoints. The amount by which the weights are increased or decreased is determined by α_t: high-weight classifiers will change the weights more than low-weight classifiers.

Now let's look at the weight α_t. First, we compute the weighted error of our classifier:

$$\epsilon_t = \sum_{i=1}^N D_{t,i} \delta(h_t(\mathbf{x}^i) \neq y^i)$$

If the classifier is perfect, we'll have $\epsilon = 0$, and if it's perfectly wrong, we get $\epsilon = 1$. Realistically, the worst classifier we should get is a random one, so that $\epsilon = 0.5$ – if it's any worse, we can always do better just by flipping a coin! The classifier weight is

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

As the error decreases, α_t grows, approaching infinity as $\epsilon_t \to 0$. This makes sense: if we're lucky enough to get a perfect classifier, we should trust it a lot.