CS221 Lecture notes #7

Supervised learning summary

In the previous sets of notes on supervised learning, we discussed many specific algorithms for supervised learning. Now, we're going to take a step back and discuss some of the principles of how to use these learning algorithms to achieve good performance.

1 Multi-class classification

When discussing logistic regression and decision trees, we simplified our task by focusing on binary classification tasks, where there are only two categories to distinguish. However, many problems require us to distinguish more than two categories. Many binary classification algorithms can be extended to directly deal with multiple classes, but there is one general approach we can take even for algorithms which don't have straightforward multiclass extensions.

In **one-vs.-all** (also called **one-vs.-many** or **one-vs.-rest**), if we are trying to distinguish between N different classes, we train N different classifiers, each one of which tries to distinguish one class from all the rest. For instance, suppose we are given the three-class data shown in Figure 1 (a). We construct three different classification problems, each of which uses one of the three classes for the positive examples and the other two classes for the negative examples. The resulting classifiers are shown.

How do we combine these classifiers to get a prediction on a novel example x? Each of the classifiers outputs some sort of confidence score that it sees a positive example. For instance, with logistic regression, the confidence score is given by $h_\theta(x)$. For decision trees, it is the probability estimate associated with the corresponding leaf node. Our prediction on the new example x will

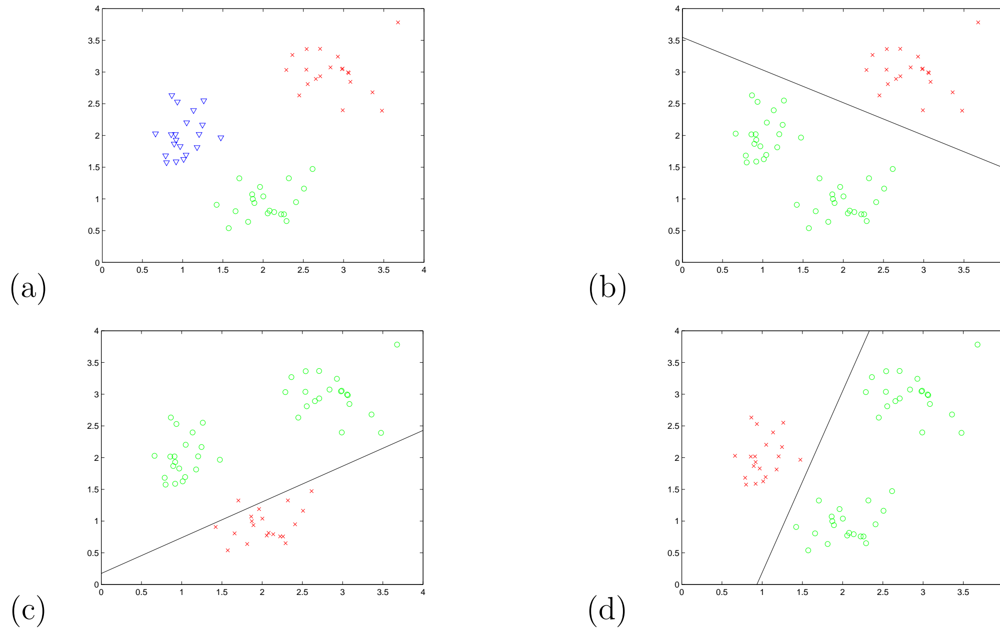

Figure 1: (a) A multiclass classification problem, with three categories. (b-d) Learned classifiers for each of the binary classification subproblems in one-vs.-all.

simply be the class for which the classifier returns the highest confidence score of the example being a member of that class.

2 Bias, variance, and generalization error

In the first machine learning lecture, we introduced the idea of overfitting or underfitting. Recall that we said a model *underfits* the training data if, like the first model in Figure 2 (b), it does not capture all of the structure available from the data. On the other hand, a model *overfits* if it captures too many of the ideosyncrasies of the training data, as in Figure 2 (d). In this section, we define more formally what we mean by overfitting and underfitting.

2.1 Regression

For the moment, let's focus on the regression problem. Suppose we have a training set $S_{\text{train}} = \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ sampled independently

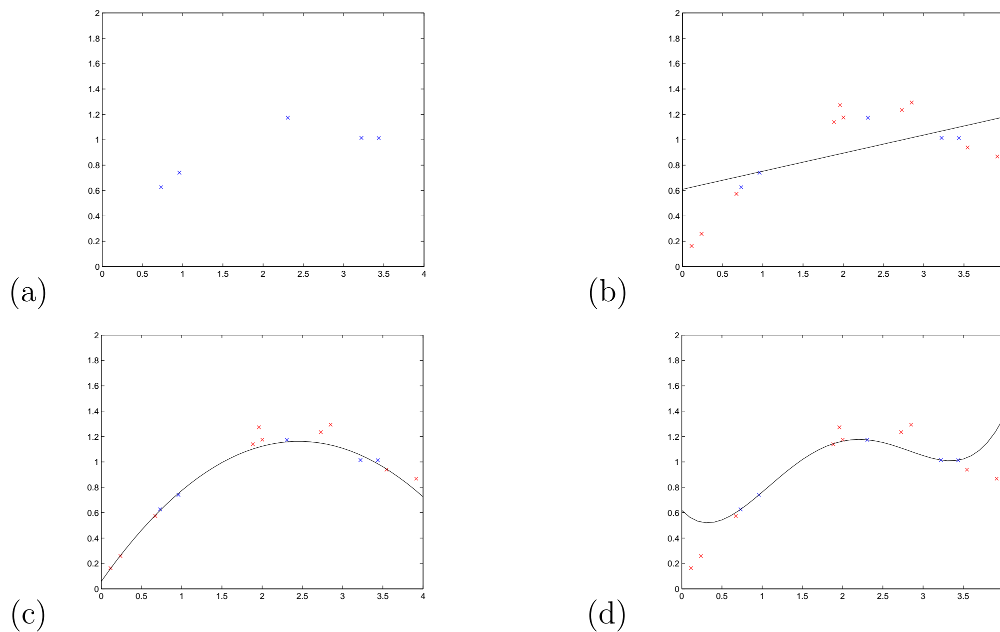

Figure 2: (a) Five data points to which we would like to fit a polynomial model. (b) The linear regression fit of a linear function to the five data points. New test examples are shown in red. (c) The linear regression fit of a quadratic polynomial. (d) The linear regression fit of a fourth-degree polynomial.

and identically distributed (IID) from some distribution $\mathcal{D}$. Define the **average training error** $\varepsilon_{s_{\text{train}}}(h_\theta)$ of a hypothesis h_θ to be:

$$\varepsilon_{S_{\text{train}}}(h_\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2.$$

But we're not actually interested in classifying training examples, since we have their labels already. What we care about is the **generalization error**, or the expected error on new examples (drawn from $\mathcal{D}$). This is written as as:

$$\varepsilon(h_\theta) = \mathrm{E}_{(x,y)\,\sim\,\mathcal{D}}[(h_\theta(x) - y)^2].$$

Now we can be more precise about why underfitting and overfitting are bad: they lead to high generalization error. Figure 2 shows what happens when we draw new examples from $\mathcal{D}$. The linear model shown in (b) has high generalization error because it is too simple a model to capture all of the structure in the data. The fourth-order polynomial in (d), however, has high generalization error because it varies too wildly. The best model, the quadratic polynomial in (c), has the lowest generalization error because it captures all the structure in the data, but no more.

We often describe models which underfit as having high **bias**, and models which overfit as having high **variance**. To make sense of these terms, imagine fitting the model parameters to a series of random data sets drawn from $\mathcal{D}$. Such a simulation is shown in Figure 3. The linear model will typically generate roughly the same parameters on each run, and so the "variance" from one trial to the next is small. However, all of these models will be systematically "biased" to underestimate the target values for points in the middle and overestimate the target values of points near the ends. On the other hand, the fourth-degree polynomial estimate varies wildly from trial to trial, and therefore it has high variance. But on average, it tends to guess correctly. This is shown on the right, where the parameters are averaged over many iterations.

We can't directly find out the generalization error, since we only have a finite set of data to work with. Instead, we estimate generalization error using a separate test set $\{(x^{(1)}_{\text{test}}, x^{(1)}_{\text{test}}), \ldots, (x^{(k)}_{\text{test}}, y^{(k)}_{\text{test}})\}$, which we do not include during the training phase. We define the **test error** $\varepsilon_{S_{\text{test}}}$ as follows:

$$\varepsilon_{S_{\text{test}}}(h_\theta) = \sum_{i=1}^{k}(h_\theta(x^{(i)}_{\text{test}}) - y^{(i)}_{\text{test}})^2.$$

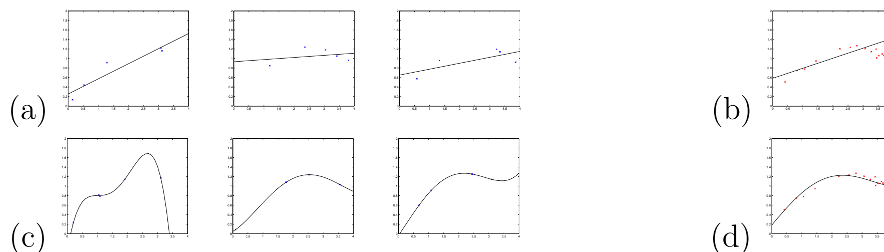

Figure 3: (a) The linear regression fits of linear functions to three different training sets, each uniformly sampled over the interval $[0, 4]$. Because most of these functions look fairly similar, we say the linear model has low variance. (b) The resulting linear model when the parameters are averaged over 50,000 trials. On average, the models systematically underestimate the function in the middle of the range and overestimate it near the ends, so we say the linear model has high bias. (c) The linear regression fits of a fourth-order polynomial to three random training sets. The fourth-order model has high variance. (d) The fourth-order fit averaged over 50,000 trials. The average of all of the parameter values achieves a good fit, so we say the model has low bias.

It can be shown that the test error is a good predictor of the generalization error. Training error is not a good predictor of generalization error; in fact, it will be overly optimistic, especially for more complex models. Recall from the first machine learning lecture our cartoon, shown in Figure 4, which show how training and test error vary as a function of model complexity. (Model complexity might include the degree of the polynomial, the size of the decision tree, or the number of features.)

2.2 Classification

What do overfitting and underfitting mean in the context of classification? Figure 5 shows a cartoon example. Our definitions for classification will be identical to those for regression, except that we use the **0/1 classification error**, the proportion of misclassified examples, rather than mean-squared error, as the penalty. Suppose for a moment that our classifier outputs a binary decision 0 or 1. (This might be achieved, for instance, by choosing a threshold confidence score in logistic regression.) A useful notation is the **indicator function** $1\{\cdot\}$, where $1\{\text{true}\} = 1$ and $1\{\text{false}\} = 0$. For instance, $1\{2 = 3\} = 0$ and $1\{1 + 1 = 2\} = 1$.

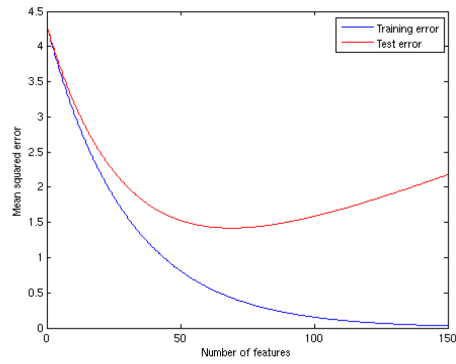

Figure 4: A cartoon picture of training and test error as a function of model complexity. Model complexity increases from left to right.

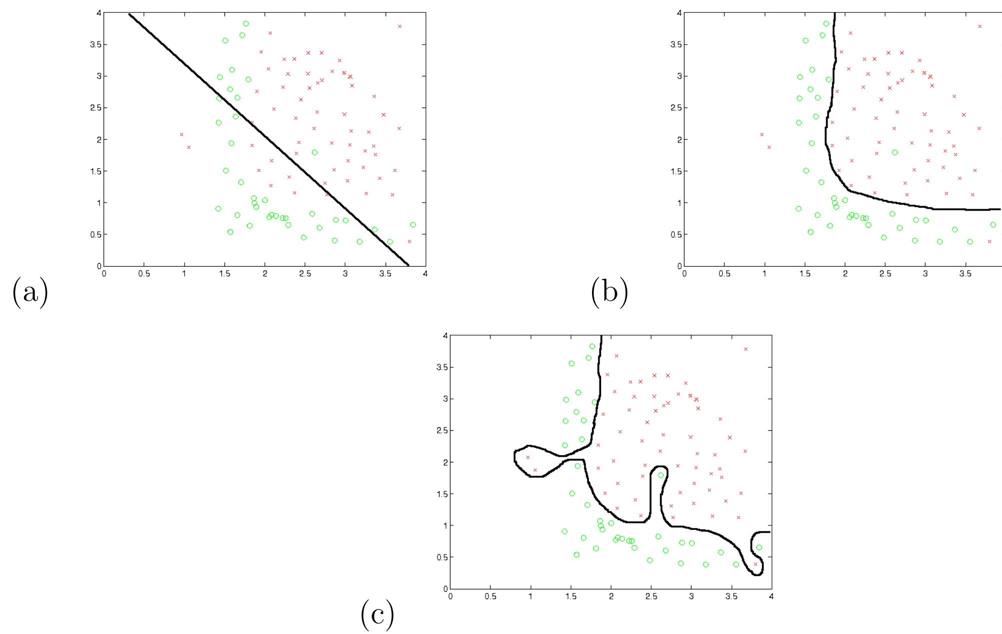

Figure 5: (a) A classifier which underfits the data. (b) A classifier which achieves a good fit. (c) A classifier which overfits.

We define the **training error** as the proportion of training examples which are misclassified:

$$\varepsilon_{S_{\text{train}}}(h_\theta) = \frac{1}{m}\sum_{i=1}^{m} 1\{h_\theta(x^{(i)}_{\text{train}}) \neq y^{(i)}_{\text{train}}\}$$

The **generalization error** is defined as the probability of a new example being misclassified:

$$\varepsilon(h_\theta) = P_{(x,y)\,\sim\,\mathcal{D}}(h_\theta(x) \neq y)$$

All of the properties we outlined above hold true for classification as well as for regression.

3 Bias and variance in practice

Now that we've introduced bias and variance, how do we minimize the problems associated with both of them? In other words, how do we choose a model which achieves a good tradeoff between bias and variance? Often, we might have a fixed set of models to choose from. For instance, we might be choosing between the polynomials of degree i, for $i \leq N$. Or we might be trying to choose the right tree depth for our decision tree classifier. In these cases, it makes sense to use **hold-out cross-validation**. Here, we leave aside part of our training data as a **hold-out cross-validation set** which we use to evaluate the performace of the different models. Specifically, the procedure is as follows:

- Split your data (not including test data) randomly into, say, 70% train and 30% cross-validation.
- For each of your N models, train a hypothesis from the training set. This will result in N different hypotheses.
- Evaluate each of these hypotheses on the cross-validation set. Choose the one with lowest error on the cross-validation set.
- Evaluate this model on the test data. This will give you the final number you report as your estimated generalization error.

Optionally, between the third and fourth step above, you may also retrain the selected model on the combination of the training and cross-validation sets, since training on more data almost always gives better performance.

But, as is the case with most of what we learn, real life is rarely this simple. Often, you won't have a fixed set of models you are trying to choose amongst. Suppose you've trained a learning algorithm and it gives poor generalization error. What do you do next? Use fewer features? More features? Get more data? Implement a different learning algorithm? Some of these options are clearly costly, so it would be nice if we had a rough idea of which ones are likely to work in which situations. Consider the following:

- If your model has high bias, it is too simple. It doesn't fit the structure already there in the data. In this case, you want to make your model more complex, perhaps by adding more features or using deeper decision trees.
- If your model has high variance, it is too complex. It fits the ideosyncratic properties of the data, and doesn't generalize well. In these cases, you want to simplify your hypothesis (e.g. by removing features) or get more data.

Keeping this advice in mind may well save you from wasting six months collecting more data when the problem is high bias, or six months designing new features when the problem is high variance.

How do you actually tell if your problem is bias or variance? You can do so by checking how different the training and test error are. The simplest way to do this is to simply compare the training and test errors given all of your data. If they are very different, your model is likely to be high variance, while if they are almost the same, your model is likely to be high bias. Alternatively, we can also plot the training and test error as a function of the number of training examples. In other words, we train our classifier only on the first 100 examples, then on only the first 200, and so on. Figure 6 shows a cartoon of what such plots might look like if your model is high bias or high variance. If the training and test errors appear to have asymptoted, your model is high bias, while if they are still steadily changing, your model is high variance. (Ask yourself: why does the training error increase as you add more examples?)

More generally, it is impossible to give a general recipe for applying a given machine learning algorithm to any arbitrary problem, since so much depends on particular properties of the problem, such as the amount of data available, the kind of structure in the data, and the noisiness of the data. There is no substitute for a careful analysis of the particular case to determine where the learning algorithm is having trouble.

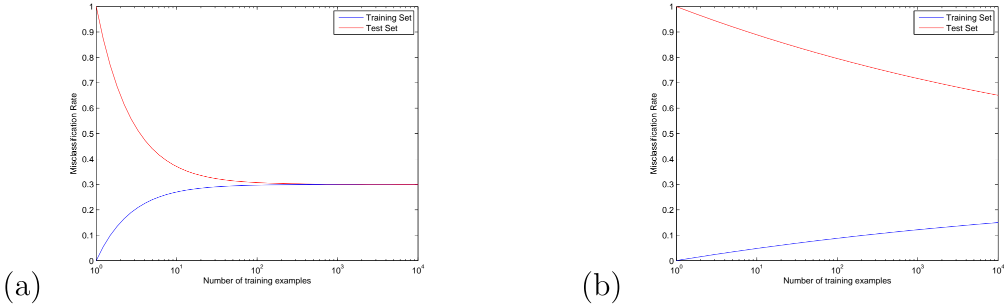

Figure 6: Cartoon examples showing how the training and test error might vary with the number of training examples. (a) A model with high bias. (b) A model with high variance.

Actually, this is a special case of a general rule of thumb for building complex AI systems in general. Broadly speaking, there are two ways to approach any problem:

1. Think for a long time about the best way to solve the problem, and then implement it.

2. Implement something quick and dirty to begin with, and then iteratively improve on it as you carefully analyze the deficiencies at each stage.

The second option is often preferable. It is not always a good way to do machine learning research, but it is a good way to get a real-world system working. You almost never know in advance what will go wrong, and unexpected turns can ruin the best-laid plans. You don't know if your model will turn out to be high bias or high variance. Therefore, you don't know in advance whether you should spend time collecting more data or implementing new and better features. Rather, you should build a prototype, check whether the problem is bias or variance, and fix it.

This is analogous to the idea of premature optimization in software engineering. It is hard to predict in advance which part of your code will be the bottleneck. If you try to guess, you will just waste time optimizing sections which make up only a small portion of the running time, and add useless clutter to your code.

Here are some more examples (which you may possibly encounter at some point in your lives) of situations where this rule applies:

- Object recognition. Is the problem that your classifier has a hard time recognizing clocks? Mugs? Or is the problem related to tracking objects,

rather than identifying them? Is running time a problem? It is hard to predict any of these factors in advance.

- Robot dog. Does it fall over when it moves its legs? Does its body collide with the terrain? Is the planning algorithm too slow? Is it unable to move its leg from one point to another without hitting a rock?

Build the system first — that will tell you which part of the task is the hard part, and you will know where to focus your attention.

4 Bagged decision trees

We said it's hard to predict in advance whether an algorithm will have a problem with bias or variance, but in fact, single decision tree classifiers famously have a problem with high variance. Why? Recall that models tend to have a problem with variance if their behavior varies wildly depending on the training set. It turns out that our information gain criterion for choosing decision tree split nodes tends to have a lot of "close calls," and because of this, differences in the training data can cause much different trees to be learned.

As we will see shortly, it is possible to reduce the variance of decision trees by training many models on a series of different training sets, and then somehow averaging these models together. In general, algorithms which try to improve performance by averaging different models together are known as **ensemble methods**.

As an intuition for why averaging models should reduce variance, recall the comparisons in Figure 3. We saw that, even though the individual fourth-degree polynomials showed high variance, when we averaged the hypotheses learned from 50,000 different training sets, we got an excellent fit. This is, of course, slightly misleading, because we will never have enough data to generate 50,000 different training sets IID. If we did, we could also improve performace (relative to the individual hypotheses) by combining the 50,000 training sets into one big training set and learning a single hypothesis on the combination. In fact, in the case of linear regression, it turns out we can't do better than simply using all of our training data as one big training set. But surprisingly, in the case of decision trees, we can do better.

A particularly good ensemble method for decision trees is called **bagging**. Suppose we have a training set S_{train} with m examples. We generate a series of B random training sets by sampling m elements from S_{train} *with*

relpacement (meaning an example may be chosen multiple times). More formally, we use the following procedure:

For $b = 1, \ldots, B$,

For $i = 1, \ldots, m$, choose $(\hat{x}^{(i)}, \hat{y}^{(i)}) \in \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ randomly.

Learn h_b using $\{(\hat{x}^{(1)}, \hat{y}^{(1)}), \ldots, (\hat{x}^{(m)}, \hat{y}^{(m)})\}$.

The final hypothesis $h(x)$ will be the average of the B learned hypotheses, $h(x) = \frac{1}{B} \sum_{b=1}^{B} h_b(x)$.

4.1 Boosting

We just showed how to reduce the variance of decision trees by averaging models learned on random training sets. But we can do even better by systematically choosing the series of training sets to include the "hardest" examples. More specifically, each decision tree is trained on a training set which emphasizes the examples the previous tree got wrong. This general strategy is known as **boosting**, and the particular algorithm we present is **AdaBoost**, or Adaptive Boost. AdaBoost is done as follows:

Initialize $D_1(1) = D_1(2) = \cdots = D_1(m) = \frac{1}{m}$.

For $b = 1, \ldots, B$,

For $i = 1, \ldots, m$,

Choose $(\hat{x}^{(i)}, \hat{y}^{(i)}) \in \{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ randomly, where the probability of choosing the ith instance is $D_b(i)$.

Learn h_b using $\{(\hat{x}^{(1)}, \hat{y}^{(1)}), \ldots, (\hat{x}^{(m)}, \hat{y}^{(m)})\}$.

Define $\varepsilon_b = \sum_{i=1}^{m} D_b(i) 1\{h_b(x^{(i)}) \neq y^{(i)}\}$.

Set $\alpha_b = \frac{1}{2} \log \frac{1-\varepsilon_b}{\varepsilon_b}$.

Set

$$D_{b+1}(i) = \begin{cases} \frac{D_b(i) e^{-\alpha_b}}{z} & \text{if } h_b(x^{(i)}) = y^{(i)} \\ \frac{D_b(i) e^{\alpha_b}}{z} & \text{if } h_b(x^{(i)}) \neq y^{(i)} \end{cases},$$

where z is a normalization constant chosen so $\sum_{i=1}^{m} D_{b+1}(i) = 1$.

The final hypothesis is given by $h(x) = \sum_{b=1}^{B} h_b(x) \frac{\alpha_b}{\sum_{t=1}^{B} \alpha_t}$.

Boosted decision trees are often regarded as one of the best supervised learning algorithms. They are often used with depth-limited decision trees[1], and the number of decision trees is often very large, on the order of tens of thousands.

There is a remarkable theoretical result about AdaBoost: if each tree is a "weak learner", in that it does slightly better than random chance on a given data set (say it achieves 51% accuracy), the algorithm as a whole will approach zero training error as B approaches ∞.

[1]Sometimes, the depth is limited to 1, so that the decision tree just tests a single feature of the root node, and then makes its prediction. These very small decision trees are also called **decision stumps**.