CS260: Machine Learning Theory
Lecture 17: Finishing up Kernels; Course Wrap-Up

November 21, 2011

Lecturer: Jennifer Wortman Vaughan

1 The Kernel Trick

At the end of the last lecture, we considered what would happen if, instead of using our original data points $(\vec{x}_1, y_1), \cdots, (\vec{x}_m, y_m)$ as input to the SVM algorithm, we wanted to apply the SVM algorithm to transformed data points $(\phi(\vec{x}_1), y_1), \cdots, (\phi(\vec{x}_m), y_m)$ for some mapping ϕ. We saw that the dual optimization problem would depend on the input points only through dot products of the form $\phi(\vec{x}_i) \cdot \phi(\vec{x}_j)$. Additionally, the label of a new point $\vec{x}$ could be computed as

$$\text{sign}\left(\sum_{i=1}^{m} \alpha_i^* y_i \phi(\vec{x}_i) \cdot \phi(\vec{x}) + b^*\right) ,$$

where

$$b^* = y_\ell - \sum_{i=1}^{m} \alpha_i^* y_i \phi(\vec{x}_i) \cdot \phi(\vec{x}_\ell)$$

for some ℓ such that $\alpha_\ell^* > 0$ (i.e., some ℓ such that $\phi(\vec{x}_\ell)$ is a support vector). There are two things to note about this. First, there is no need to compute the weights $\vec{w}^*$ directly. Second, this also depends on the input points only through dot products of transformed points.

It turns out that these observations are very powerful, and allow us to work with very large or even infinite vectors of transformed features. Given a mapping ϕ, we define the corresponding **Kernel function** K on data points $\vec{x}$ and $\vec{z}$ as

$$K(\vec{x}, \vec{z}) = \phi(\vec{x}) \cdot \phi(\vec{z}) .$$

Since both the SVM optimization problem and the resulting classifier only handle data through dot products of data points, even if computing $\phi(\vec{x})$ is inefficient (or even impossible), the SVM algorithm can still be run efficiently as long as we can compute the kernel function efficiently.

1.1 Examples of Kernels

Consider first the mapping function mapping n-dimensional vectors to n^2-dimensional vectors, with

$$\phi(\vec{x}) = \begin{bmatrix} x_1 x_1 \\ x_1 x_2 \\ \vdots \\ x_n x_{n-1} \\ x_n x_n \end{bmatrix} .$$

All CS260 lecture notes build on the scribes' notes written by UCLA students in the Fall 2010 offering of this course. Although they have been carefully reviewed, it is entirely possible that some of them contain errors. If you spot an error, please email Jenn.

We can derive the kernel function. We have that

$$\begin{aligned}
K(\vec{x},\vec{z}) &= \phi(\vec{x})\cdot\phi(\vec{z}) \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j \\
&= \left(\sum_{i=1}^{n} x_i z_i\right)\left(\sum_{j=1}^{n} x_j z_j\right) \\
&= (\vec{x}\cdot\vec{z})^2 \, .
\end{aligned}$$

We can calculate the value of this function in time linear in the dimension of $\vec{x}$ instead of quadratic, giving us an efficiency boost.

Suppose that we modified this kernel function slightly and defined $K(\vec{x},\vec{z}) = (\vec{x}\cdot\vec{z}+c)^2$. We have

$$\begin{aligned}
K(\vec{x},\vec{z}) &= (\vec{x}\cdot\vec{z}+c)^2 \\
&= \sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j z_i z_j + \sum_{i=1}^{n}(\sqrt{2c}x_i)(\sqrt{2c}z_i) + c^2.
\end{aligned}$$

This corresponds to the following feature mapping, which includes linear terms in addition to quadratic terms, and is an extension of the feature mapping we described that allowed us to learn circles:

$$\phi(\vec{x}) = \begin{bmatrix} x_1x_1 \\ x_1x_2 \\ \vdots \\ x_nx_{n-1} \\ x_nx_n \\ \sqrt{2c}x_1 \\ \vdots \\ \sqrt{2c}x_n \\ c \end{bmatrix} .$$

We again get a savings in terms of computation. Instead of working directly with the transformations $\phi(\vec{x})$ which would take $O(n^2)$ time, we can compute the kernel functions, which takes only $O(n)$ time.

Generalizing further, the Kernel $K(\vec{x},\vec{z}) = (\vec{x}\cdot\vec{z}+c)^d$ maps an n-dimensional vector to a feature space of size $O(n^d)$. However, despite working in an $O(n^d)$ dimensional space, computing the Kernel only takes $O(n)$ time.

There are kernels that correspond to infinite dimensional feature vectors that we can still work with efficiently, such as the popular radial basis function kernel. Running the SVM algorithm directly on these feature vectors would obviously be impossible since the vectors are infinite, but we can still do it efficiently using the kernel trick.

1.2 A Note on Generalization Error

We might expect it to be a bad idea to use what are effectively infinite-dimensional feature vectors since the class of functions we are learning will become very complex, but in fact, this works well in practice. It can be shown that only the margin and number of support vectors matter.

1.3 Kernelizing the Perceptron

The kernel trick can be used with other algorithms too. Recall the basic Perceptron algorithm that was introduced in lecture 8.

```
PERCEPTRON ALGORITHM
Initialize $\vec{w}_1 = \vec{0}$
At each round $t \in \{1, 2, \cdots\}$
    • Receive input $\vec{x}_t$
    • If $\vec{w}_t \cdot \vec{x}_t \geq 0$, predict $+1$, else predict $-1$
    • If there is a mistake (i.e., if $y_t(\vec{w}_t \cdot \vec{x}_t) < 0$), set $\vec{w}_{t+1} \leftarrow \vec{w}_t + y_t \cdot \vec{x}_t$, else set $\vec{w}_{t+1} \leftarrow \vec{w}_t$.
```

Suppose we wanted to replace each input $\vec{x}_t$ with a transformation $\phi(\vec{x}_t)$. Let $k(t)$ denote the number of mistakes that have been made up to and including time t, and let $m(i)$ denote the round on which the ith mistake was made. We can rewrite the weight vector as

$$\vec{w}_{t+1} = \sum_{i=1}^{k(t)} y_{m(i)} \vec{x}_{m(i)} \ .$$

In our kernelized version, this would be

$$\vec{w}_{t+1} = \sum_{i=1}^{k(t)} y_{m(i)} \phi(\vec{x}_{m(i)}) \ .$$

To determine the label to predict for the next input point, we would compute

$$\text{sign}\left(\sum_{i=1}^{k(t)} y_{m(i)} \phi(\vec{x}_{m(i)}) \cdot \phi(\vec{x}_{t+1}) \right) = \text{sign}\left(\sum_{i=1}^{k(t)} y_{m(i)} K(\vec{x}_{m(i)}, \vec{x}_{t+1}) \right) \ .$$

Note that unlike the original Perceptron algorithm, this version requires us to store all of the previous examples on which a mistake has been made.

The original Perceptron mistake bound can be applied immediately here. In particular, if the data has a margin of at least γ in "ϕ-space", then the number of mistakes will be bounded by $O(1/\gamma^2)$. This holds even if the original data points were not linearly separable!

2 Course Wrap-Up

We will spend the second half of class discussing ways in which you can apply the ideas and themes of this course to your own research.

2.1 How to Approach a Research Problem

The key ideas described here are important to keep in mind if you want to derive some learning theoretical results of your own, but apply to people working in other research areas too.

1. **Clearly define a model.** The model needs to come first. It should specify things like what it is the learner is trying to learn, what kind of data is available, how the data is presented to the learner, what type of feedback the learner receives, and the goal of the learning process. The model should capture all of the critical aspects of the particular learning problem that you have in mind without being overly complex; see point 2.

   Defining a model is important in empirical work too. A frequent problem with applications-oriented research is that the researchers don't take the time to precisely define the problem that they are attempting to solve. Failing to precisely define a problem can lead to flawed solutions.

2. **Always start simple.** In the context of learning theory, starting simple might mean assuming noise-free data or thinking about simple, well-studied function classes like intervals or two-dimensional thresholds. It is much easier to catch big problems in a model if you start with the most stripped-down, basic version first. If the simple version works, you can add the bells and whistles later.

   Working through some simple examples can also be extremely helpful for building intuition.

3. **Be prepared to change.** Nobody gets all of the details right on their first try. If you do catch problems in your model, think about small variations or simplifications you could make to work around them.

4. **Learn the tricks of the trade.** Many of the proof techniques we've seen in this class are really clever, and you might find yourself wondering how people ever came up with these ideas. However, once you have worked in the area for a while, you realize that a lot of the same tricks are used again and again. We've seen several examples of tricks that are commonly used in learning theory:

   - To get rid of annoying logarithmic terms or make numbers easier to multiply, it often helps to pull out the old $1 + x \leq e^x$ trick at least when the x you have in mind is very small. This comes from the Taylor expansion for e^x. Sometimes other Taylor expansions can be used to bound tricky terms.
   - Use the union bound to easily bound disjunctions of (relatively unlikely) events.
   - It is usually safe to assume that the average of a large number of random variables will be "close to" its expectation. You can use Hoeffding's inequality (or other Chernoff bounds) to make this precise. This is especially useful for moving from bounds on empirical (training) error to bounds on generalization error.
   - Especially when analyzing online algorithms, it can be useful to define a potential function to relate quantities of interest (e.g., error) to other quantities (e.g., weights, as we saw in the analysis of Winnow).
   - If you have an idea for an algorithm that might work in a particular setting, define a vague outline of your algorithm first and fill in the details as you need to. For example, remember how the learning rate η of RWM and the α_t parameter in AdaBoost were both optimized after the fact, based on the analysis.

   The tricks of the trade will be different if you're working in another area, but the same idea applies.

5. **Verify that your results make sense.** If you derive a bound, check that the dependence on various quantities of interest is sensible. For example, it would be unusual to have an error bound that increases in the number of samples. You can check how reasonable your bound is empirically too. Try

generating some synthetic data that meets your assumptions and see how your error bound compares with error of your algorithm on this data.

Performing simple "sanity checks" is important in any area of research to help you catch mistakes.

2.2 Additional Tips for the Practitioners

Even if you don't plan on deriving theoretical results of your own, the ideas we discussed in this class can give you a better understanding of empirical issues that arise and how to deal with them. For example, remember the learning bound we derived for the agnostic setting in Lecture 4:

$$\mathrm{err}(\hat{h}) \leq \min_{h \in \mathcal{H}} (\mathrm{err}[h]) + \sqrt{\frac{2 \ln |\mathcal{H}| + 2 \ln (2/\delta)}{m}}.$$

Even if you forget the details, the intuition captured in this bound is extremely important! The first term (referred to as the *bias* term) in this equation tells us that we want a large hypothesis class. However, the second term (the *variance* term) tells us that a smaller hypothesis class is better because of the Occam's razor principle. This bias-variance trade-off doesn't just arise in theory. It is commonly seen in practice too. Being aware of this trade-off can help you diagnose and solve problems that come up in practice that lead to high error.

For example, suppose you run a learning algorithm and it gives you a function that has low error on your training data, but high error on a held-out test set. The problem here can be diagnosed as high variance; you are overfitting the data. As the bound suggests, there are several ways to remedy this. One way is to get more data (i.e., make m bigger.) The other is to remove some features or use a more simple function class (i.e., make $|\mathcal{H}|$ smaller).

On the other hand, if you run a learning algorithm and it gives you high error on both your training data and your held-out test set, you probably have a bias problem. In this case, you may want to add *more* features or use a *more complex* function class.

Several of the other results we derived have clear take-away messages that practitioners can use. For example,

- Regularization can be used to add stability to solutions, which avoids overfitting.
- Very simple decision rules can be combined to form a low-error hypothesis.
- Large margins lead to low generalization error. As we saw today, this holds for "transformed" feature spaces too.

Hopefully you will find intuitions from this class helpful when designing new algorithms of your own.