Using Support Vector Machines Effectively

Getting the best performance out of support vector machines (SVMs)

August 2012, updated March 2014

Introduction

Support Vector Machines (SVMs) are one of the most commonly used supervised learning techniques, in large part because they are very effective for lots of problems. However, they require a bit of knowledge to use most effectively which is often omitted from the documentation provided with implementing libraries and other tutorials.

Picking a library

The canonical first suggestion used to be libsvm, which works very well and is also frequently updated. Note: in many versions of Matlab, the version of libsvm installed is from the bioinformatics toolbox, which always gave me inferior results. It's better to install the latest version from the libsvm website. (Note that the order of arguments is different from the two versions, so if you switch, you'll have to fix your code accordingly.)

However, if you're writing Python, then the best library is now scikit-learn.

For large-scale linear problems, stochastic gradient descent (SGD)-based methods are much faster to train, and offer only slightly worse performance. Here's the implementation in scikit-learn.

Prescaling/normalization/whitening

SVMs assume that the data it works with is in a standard range, usually either 0 to 1, or -1 to 1 (roughly). So the normalization of feature vectors prior to feeding them to the SVM is very important. (This is often called whitening, although there are different types of whitening.) You want to make sure that for each dimension, the values are scaled to lie roughly within this range. Otherwise, if e.g. dimension 1 is from 0-1000 and dimension 2 is from 0-1.2, then dimension 1 becomes much more important than dimension 2, which will skew results.

In more detail, you have to normalize all of your feature vectors by dimension, not instance, prior to sending them to your svm library.

Some libraries recommend doing a 'hard' normalization, mapping the min and max values of a given dimension to 0 and 1. However, in our experience, we found that is better to do a 'soft' normalization which subtracts the mean of the values and divides by twice the standard deviation (again, by dimension). Thus, if your input has **d** dimensions, then you will have **d** means and **d** standard deviations, no matter how many training examples you have. Note that if you're implementing this yourself, standard deviations for some dimensions might be zero, so the division could give you a divide-by-zero error. So you should add a very small epsilon value to it to prevent this.

These normalized vectors are sent to your SVM library for training. Then during testing, it is important to construct the test feature vectors in exactly the same way, except that you use the means and standard deviations saved from the training data, rather than computing it from the test data. In other words, scale your test inputs using the saved means and standard deviations, prior to sending them to your SVM library for classification.

If you're using scikit-learn, then this soft scaling is provided under preprocessing.StandardScaler.

Parameters

SVMs can be quite sensitive to training parameters, but fortunately there are relatively few of them to tune:

1. First are C (the slack variable cost) and γ (the width of the Gaussian if using an RBF kernel) values. Generally these are searched in exponential factors: for C, something like 0.1, 1, 10, 100, 1000; for γ, something like 0.1, 0.01, 0.001, 0.0001, 0.00001). If possible, you should try all combinations of C and γ to find the ones that give you the best accuracy (using cross-validation). Also, if you're competing for a benchmark, it's important you evaluate this on a held out subset so you don't inadvertently overfit to the benchmark data.
2. If you have unbalanced input data (i.e., many more negative examples than positive), you should generally set the training weight factors for the positive and negative classes in inverse proportion, so that the trained model is not more sensitive to the class for which you have more examples. There are some circumstances where you want the training to be biased, but those are generally rare.

Getting probabilities from outputs

In many cases, you will want more than just a binary yes or no output -- some measure of confidence or a proper probability. Most SVM libraries let you access this decision value (sometimes called a score or distance), which is the actual output from the SVM evaluation function. Taken raw, these values are unbounded, even though they might typically fall in a range around [-1, 1]. If you're only concerned with ranking multiple instances, then you can use these raw values directly to rank by confidence.

However, if you're dealing with multiple classifiers, then these raw values are not directly comparable! Similarly, if you need a proper probability (i.e., between 0-1), then you cannot use these raw values either. You have to convert them to the proper range.

The standard way of doing this is Platt scaling, where a logistic regression model (usually a sigmoid curve) is fit to the outputs of some labeled inputs to obtain probabilities. If you have enough labeled data, this is probably the right thing to do, but in many cases, there isn't enough labeled data available to properly estimate the parameters of the sigmoid.

An easier approach is to use our work on Multi-Attribute Spaces to calibrate outputs using Extreme Value Theory. This does not require any labeled data to estimate and has a few other benefits as well: it doesn't assume balanced sampling or symmetric error models that are common in many Platt implementations. Code for this normalization is provided here.

Note: if you find inaccuracies here or have other suggestions, please email me.