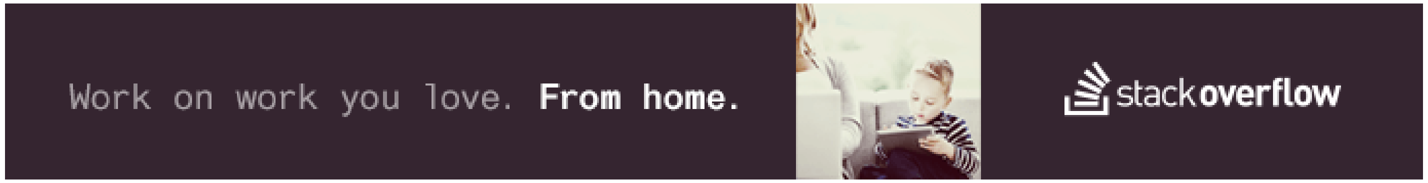
Right function for normalizing input of sklearn SVM



I found several questions related to this, but no one solved my doubts. In particular, the two answers to this question confused me even more.

I'm training a linear SVM on top of a set of features - Convolutional Neural Net features resulting from images. I have, for example, a 3500x4096 `X` matrix with examples on rows and features on columns, as usual.

I'm wondering how to properly standardize/normalize this matrix before feeding the SVM. I see two ways (using sklearn):

1. Standardizing features. It results in features with 0 mean and unitary std.

   ```
   X = sklearn.preprocessing.scale(X)
   ```

2. Normalizing features. It results in features with unitary norm.

   ```
   X = sklearn.preprocessing.normalize(X, axis=0)
   ```

My results are sensibly better with normalization (76% accuracy) than with standardiing (68% accuracy).

Is it a completely dataset-dependent choice? Or how can one choose between the two techniques?

python machine-learning statistics scikit-learn svm

asked Jun 18 '15 at 14:51
Iena Plissken
31 2

1 Are you using cross-validation to select optimal parameters for your Linear SVM? When you say your results are sensibly better: are you training and evaluating on all of your data, or running a cross-validation and reporting the average result across your validation sets? – Andreus Jun 19 '15 at 14:34

I am using a fixed train/validation split for this analysis. The point is interesting, I will try with different splits and see if the described situation holds. Regarding the hyperparameters, I set C=1. I did some grid search, but it seems like the default choice is the optimal one. – Iena Plissken Jun 24 '15 at 11:24

1 Answer

You should choose the scaling scheme based on what makes sense with your data. There are different ways of scaling, and which one you'll use depends on the data. Each scheme brings values of different features into comparable ranges, but each of them preserve different types of information (and distort others) as they do so. And even though there are rational explanations behind why some scaling schemes are better suited for a specific case, **there is nothing wrong with just trying these different ones (like you did with standard scaling and normalization) and using the one that works better** (as long as you cross-validate or otherwise make sure that your performance measurement is general and accurate).

StandardScaler
This is what `sklearn.preprocessing.scale(X)` uses. It assumes that your features are normally distributed (each feature with a different mean and standard deviation), and scales them such that each feature's Gaussian distribution is now centered around 0 and it's standard deviation is 1.

It does this by calculating the mean and stdev for each feature, then converts each actual value for the feature into a z-score: how many stdevs away from the mean is this value? z=(value-mean)/stdev

This quite often works well but if the normality assumption is completely wrong for your case, then this may not be the best scaling scheme for you. Practically, in many cases where the normality assumption does not hold, but the distributions are somewhat close, this scheme still works pretty well. However, if the data is completely far away from normality, for example highly skewed, fat-tailed distributions (like a power-law), this scheme will not give good results.

Normalizer

This is what `sklearn.preprocessing.normalize(X, axis=0)` uses. It looks at all the feature values for a given data point as a vector and normalizes that vector by dividing it by it's magnitude. For example, let's say you have 3 features. The values for a specific point are `[x1, x2, x3]` . If you're using the default `'l2'` normalization, you divide each value by `sqrt(x1^2 + x2^2 + x3^2)` . If you're using `'l1'` normalization, you divide each by `x1+x2+x3` . This makes sure that the values are in similar ranges for each feature, since each feature vector is a unit vector. If feature values for a point are large, so is the magnitude and you divide by a large number. If they are small, you divide them by a small number.

The reasoning is that you can think of your data as points in an n-dimensional space, where n is the number of features. Each feature is an axis. Normalization pulls each point back to the origin in a way that it is only 1 unit distant from the origin. Basically you collapse the space into the unit hypercube. The angles between the vectors for each point (from the origin to the data point) stay the same.

This is used a lot with text data, since it makes a lot of intuitive sense there: If each feature is the count of a different word, `'l1'` normalization basically converts those counts to frequencies (you're dividing by the total count of words). This makes sense. If you're using `'l2'` normalization, the angle between two vectors (this is called the cosine distance or similarity) will stay the same when you normalize both, and this distance is closer to a *meaning* distance since it corresponds to ratios of frequencies between words and is not affected by how long of a text each vector represents.

If conserving a cosine distance type of relationship between points is what makes more sense for your data, or if normalization corresponds to a natural scaling (like taking frequencies instead of counts), then this one is more suitable.

MinMaxScaler

You can use this one like `sklearn.preprocessing.MinMaxScaler().fit_transform(X)` . For each feature, this looks at the minimum and maximum value. This is the range of this feature. Then it shrinks or stretches this to the same range for each feature (the default is 0 to 1).

It does this by converting each value to `(value-feature_min)/(feature_max - feature_min)` . It is basically at what percentage of the range am I lying? Remember that the range is only determined by the min and max for the feature. For all this cares, all values might be hanging around 10, 11, or so, and there is a single outlier that's 900. Doesn't matter, your range is 10 to 900. You can see that in some cases that's desirable, and in others this will be problematic, depending on the specific problem and data.

This scheme works much better in certain cases where StandardScaler might not work well. For example, **if the standard deviations are very small for features**, StandardScaler is highly sensitive to tiny changes between standard deviations of different features, but **MinMaxScaler is very robust**. Also, **for features with highly skewed distributions, or sparse cases where each feature has a lot of zeros that moves the distribution away from a Gaussian, MinMaxScaler is a better choice.**

edited Jun 29 '15 at 15:37 answered Jun 26 '15 at 0:03

Irmak Sirer
126 5

Astonishingly complete answer, Irmak! Thank you a lot! Long story short, since my features are pretty sparse, I will definitely try out MinMaxScaler. – Iena Plissken Jun 27 '15 at 9:37