Kernel Methods

Andrés Ponce

May 27, 2020

Kernel methods allow us to find patterns in data, useful for classification and regression. These methods allow us to quickly and efficiently work with high dimensional data. We break down a function that might not be linear in nature to linear functions of its components. $k(x, y) = (f(x), g(y))$, where f and g are the dimensional inputs.

Dual representations

Parametric vs. non-parametric models

Parametric methods refer to when we train a model that is a function of $y(x, w)$, where we map x to y. In this scenario, the training data are thrown away after training.

On the other hand, **non-parametric methods** use a nearest-neighbor classifier strategy to perform classification. However, in this scheme we keep the training data.

Kernel methods evaluate a **kernel function** based on the training data. The dual part of this representation comes from the kernel function itself. We define a basic kernel function as: [1]

$$k(x, x') = \phi(x)^T \phi(x')$$

[1] Notice this function is **symmetric**, since $k(x, x') = k(x', x)$

Based on this formulation, we can then define vector a, which we use instead of the weight vector w. What a allows us to rephrase the optimal solution as a linear combination of the entire training data set. We define a: [2]

$$a_n = -\frac{1}{\lambda}\{w^T \phi(x_n) - t_n\}$$

[2] a is the result of setting the derivative of $J(w)$ to zero, where $J(w)$ is a sum of squares formultion

Then, we next define the **Kernel Matrix** $K = \Phi\Phi^T$, which makes K a symmetric matrix with each cell in the matrix:

$$K_{nm} = \phi(x_n)^T \phi(x_m) = k(x_n, x_m)$$

Finally, the solution again is of the form $y(x) = w^T \phi(x)$, however with the new concepts we've introduced we can write the final solution as

$$k(x)^T (K + \lambda I_N)^{-1} t$$

Final notes on Dual representations

If we have M dimensions in our data, with N data points, we will usually have more data points than dimensions in our data. This

might make dual representations less efficient and might lead to more sparse matrices in the end. However, since we don't take into account the feature set of our data $\phi(x)$, we can handle data with potentially infinite features, making it more flexible.

Constructing Kernels

How do we build a kernel to lead to more accurate classification? Maybe we can first find a feature space for its inputs and then check if there is a space where the output is equal to the inner product of the data points (i.e. $y(x) = k(f(x), g(x'))$. So when we multiply the inner products in some lower dimensional space, the result exists in a higher space where we are wanting to find the kernel anyways.[3]

Once we find a candidate function, how canwe show it's a proper kernel function? We can know its a valid kernel function by calculating its eigenvalues. If all its eigenvalues are positive, then the matrix K is **positive** and **semi-definite**.

Some types of kernel functions:

- **Polynomial kernel**: $k(x, x') = (x^T x' + c)^M$

  Similar to our basic kernel function,this one is symmetric.

- **Gaussian kernel**: $k(x, x') = \exp(-\|x - x'\|^2)$

- **Sigmoidal kernel**: $k(x, x') = \tanh(\alpha x^T x' + b)$ where a and b are constants. The Gram matrix of this function is usually not positive or semi-definite, however it can be useful in practice.

[3] Maybe this is why it's used in facial recognition? Becuase each pixel in an image can correpsond to a dimension in the data, we want to break it down to a combination of lower dimensions.

Support Vector Machines for classification

Maximum Margin classifier

Suppose we have N training data points and their corresponding labels t. SVMs use a normal-enough looking formula like

$$y(x) = w^T \phi(x) + b$$

where $\phi(x)$ denotes the fixed feature-space transformation.

If we have linearly separable classes, we can have multiple decision boundaries that give adequate solutions. Is there an optimal one, though? If so, how do we calculate that?

We want to maximize the **margin**, since a greater distance would mean that the data is split more cleanly along the decision boundary. Then the perpendicular distance between the decision boundary and the data point is given by

$$\frac{t_n y(x_n)}{\|w\|} = \frac{t_n (w^T \phi(x_n) + b)}{\|w\|}$$

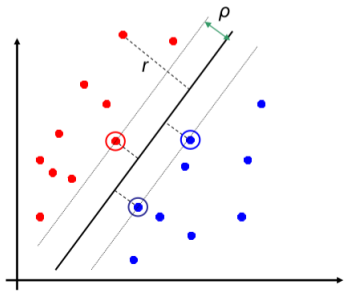

Figure 1: The margin is the smallest distance between the decision boundary and any of the training points.

which is the equation we want to maximize.

Here we can also use Lagrangian multipliers to find an optimization. The optimization can be done in $O(n^3)$.

According to the **KKT** conditions, we can have three conditions regarding SVMs:

1. $a_n \geq 0$
2. $t_n y(x_n) - 1 \geq 0$
3. $a_n\{t_n y(x_n) - 1\} = 0$

Due to the third contstraint above, either $a_n = 0$ or $t_n y(x_n) - 1 = 0$. If the former is true, then the point is of no importance when applying the decision function.

If the latter is true and $t_n(x_n) - 1 - 0$, then this point lies on the maximum margin and is called a **support vector**. We only keep the support vectors.

So far, what we have seen assumes the data are lienarly separable, that is, we can actually find a line that cleanly splts the dataset. But what if this is not the case?

Overlapping class distributions

To allow the possible misclassification of data, we can introduce a new variable called the **slack variable** ξ, such that for each data point x_n, $\xi \geq 0$. We then can modify one of the constraints mentioned above and write

$$t_n y(x_n) \geq 1 - \xi_n$$

This slack variable can tell us a couple things about the data point, for example:

1. $\xi_n = 0$: The point is on the correct side of the decision boundary (could be in the margin).
2. $0 < \xi_n < 1$: The point is in the margin, but still in the correct side of the boundary.
3. $\xi_n = 1$: The data point is on the decision boundary.
4. $\xi_n > 1$: The data point was classified incorrectly.

So ξ_n tells us how **misclassified** a certain data point is. We also use the KKT criteria mentioned above to calculate the parameters of the functions.

Summary of SVMs

Support Vector Machines are a linear model for classification or regression that use a form of sum of squares. We use a dual representation of the data, one usually described by a linear combination of the data. We use dual representation because the decision function $J(w)$ is calculated in terms of kernel functions.

SVMs try to maximize the margin between the decision boundary and the closest data point of either set. The function we try to maximize is of the form

$$\arg\min \frac{1}{2}\|w\|^2$$

To do this, we introduce a new slack variable ξ_n for each data point x_n. We use this variable whenever the dataset is not cleanly able to be partitioned in two. This slack variable tells us how misclassified a variable is. That is, we can know whether it is on the correct side of the decision boundary and by how much.

Multiclass SVMs

Since SVMs are fundamentally a two-class identifier, we need some extra mechanisms to adapt them to work with classes $k > 2$. We can have several different methods.

One-versus-the-rest

For the kth classifier, we make it so that it can distinguish between classes C_k and those points who are not in this class. We use data from C_k as positive data and the rest as data that does not belong in the class. One potential issue with this approach is that the symmetry of data points that belong in a class, i.e. the positive training data, might be lost. If we only have a small fraction of positive test cases, and an equal amount of all the other classes to use as negative examples, then the amount of data we will use as positive examples might not adequately serve the training data.

One-versus-one

With these kinds of classifiers, we compare each of them with another of the classes. This means we have $\frac{k(k-1)}{2}$ different SVM classifiers, which will make it much more computationally hard to classify the data. There is a classifier for classes i and j. This approach might remove the data inbalance issue with the previous classifier type, however the computational requirement might pose a downside.

DAGSVM

Alternatively, we can construct a DAG to act as a tree, again with $\frac{K(K-1)}{2}$ SVMs. Then, at each node we also make a decision, and as we follow the path down the tree to the leaf node, we at most evaluate $K-1$ nodes.

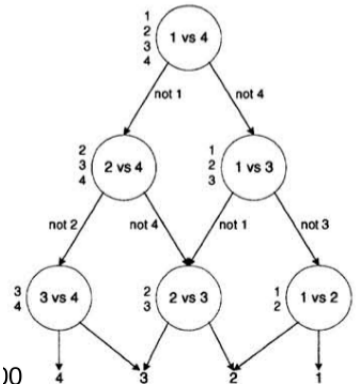

Figure 2: At each internal node we train a 1-v-1 SVM classifer, and as the point moves down the tree we eventually end up at a leaf node where the point is then classified.

ECOC

We can train an SVM again for the K classes, maybe using one-versus-one approach. Given a sutiable decoding scheme, we can then measure the robustness of errors and classifying should prove a bit easier.

Regression with SVMs

To use SVMs with regression, we first have to define an error function. Using ξ, we can define the error function as

$$E_\epsilon(y(x)-t) = \begin{cases} 0 & \text{if} |y(x)-t| < \xi \\ |y(x)-t| & \text{otherwise} \end{cases}$$

Similar to before, we can calculate the decision function for a regression version of SVMs as

$$y(x) = w^T \phi(x) + b$$

The next question then should be: how do we calculate w and b? For w, we can again use a dual representation such that

$$w = \sum_{n=1}^{N} (a_n - \hat{a}_n)\phi(x_n)$$

Then the prediction for a new point will be

$$y(x) = \sum_{n=1}^{N} (a_n - \hat{a}_n)k(x, x_n) + b$$