Week 6: Support Vector Machines

Instructor: Sergey Levine

1 Linear Classifiers without Probabilities

So far, we've seen a number of different methods for classification: decision trees, naïve Bayes, logistic regression, and neural networks. A common theme has been that linear classifiers (naïve Bayes and logistic regression) can be optimized globally, while nonlinear classifiers (decision trees and neural networks) are more powerful, but generally lack global optimization methods: decision trees are optimized heuristically, while neural networks converge to locally optimal solution (though these solutions in practice are typically quite good). In this lecture, we'll come back to linear classifiers, for which we can construct simple global optimization methods. We'll then explore how these linear classifiers can be made into powerful nonlinear classifiers using something called the kernel trick. We'll derive a type of classifier called a support vector machine (SVM), which does not attempt to model probabilities, but instead tries to directly separate positive and negative examples with the decision boundary.

First, let's review linear classifiers. When we discussed logistic regression, we talked about a linear classifier of the form

$$y = \begin{cases} -1 \text{ if } \mathbf{w} \cdot h(\mathbf{x}) < 0 \\ +1 \text{ if } \mathbf{w} \cdot h(\mathbf{x}) \geq 0 \end{cases}$$

This very simple classifier is determined entirely by the weights $\mathbf{w}$, just like logistic regression. Note that we use -1 as the negative label here for convenience (instead of 0), so that we can easily define our classifier as $y = \text{sign}(\mathbf{w} \cdot h(\mathbf{x}))$. What if we directly try to push the boundary $\mathbf{w}$ such that positive examples are all on one side and negative examples are all on the other? We can do this with an iterative algorithm that is reminiscent of stochastic gradient descent.

Algorithm 1 Perceptron training

```
w ← 0⃗
for t in 1, ..., T (T passes over the data) do
    for i in 1, ..., N (step over each datapoint) do
        if sign(w · h(x)^i) ≠ y^i then
            w ← w + y^i h(x)^i (push w toward h(x)^i if y^i = +1 or away otherwise)
        end if
    end for
end for
```

This algorithm can be interpreted as stochastic gradient descent (SGD) with a learning rate of 1 and a batch size $B = 1$. But what objective is this algorithm optimizing? Well, it's pretty clear that this algorithm will push the decision boundary so as to reduce the number of incorrect classifications, since each incorrect classification will push the decision boundary to make that classification more correct (it would be interesting to compare this update to the gradient in logistic regression!). We can check that the gradient in this case is the gradient of the following objective function (which we are trying to minimize):

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \max(-y^i(\mathbf{w} \cdot h(\mathbf{x})^i), 0).$$

That is, if $y^i(\mathbf{w} \cdot h(\mathbf{x})^i) < 0$, meaning that $\text{sign}(\mathbf{w} \cdot h(\mathbf{x})^i) \neq y^i$, we incur a cost equal to $|\mathbf{w} \cdot h(\mathbf{x})^i|$, otherwise we incur a cost of zero. This is referred to as a hinge loss, because the loss increases linearly as we get further away from the decision boundary in the "wrong" direction, but is zero for any correctly classified example.

We can also write down the objective for the perceptron as a constrained optimization problem, as following:

$$\begin{aligned}
&\min_{\mathbf{w}, s_1, \ldots, s_N} \sum_{i=1}^{N} s_i \text{ such that} \\
&y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) + s_i \geq 0 \quad \forall i \in \{1, \ldots, N\} \\
&s_i \geq 0 \quad \forall i \in \{1, \ldots, N\}
\end{aligned}$$

The constraints say first that we should have $y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) \geq 0$ whenever possible, but if we have $y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) < 0$, we can add s_i to it to make it greater than or equal to zero 0, and that incurs a cost. And of course each s_i must be positive (so we don't get a "bonus" for putting correct classifications further from the decision boundary). This type of optimization is called a linear program, and it can be solved exactly using standard linear program solvers.

2 Maximizing the margin

There are a few problems with perceptrons. One of the issues is that, if we classify all the points correctly, we still have potentially an infinite number of decision boundaries we can choose, and there is no pressure to make the decision boundary "far" from any of the points: a boundary that passes very very close to a point is just as good as one that gives it a wide margin. We could try to address this simply by removing the constraint that $s_i \geq 0$, but then we will push points that are already far from the boundary even further away. That is not really what we want. We want to push away those points that are *too* close to the boundary.

We can accomplish this by instead saying that the classifier should not only get each point right, but it should get it right by some margin γ, and then

maximize that margin. First, let's get rid of the slack variables s_i (we'll put them them back in later), and rewrite the perceptron as a constraint satisfaction problem:

$$\text{find } \mathbf{w} \text{ such that } y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) \geq 0 \quad \forall i \in \{1, \ldots, N\}.$$

We can satisfy this constraint if the data is linearly separable. To transform this into the max margin formulation, we have to figure out how far away each point is from the line (or plane) $\mathbf{w} \cdot h(\mathbf{x}^i) = 0$ and determine the margin (the minimum distance over all points), and then maximize this margin. We could write the margin like this

$$\hat{\gamma} = \min_i y^i(\mathbf{w} \cdot h(\mathbf{x}^i)).$$

However, this kind of margin is not very useful, because we could simply scale $\mathbf{w}$ by a constant and the boundary $\mathbf{w} \cdot h(\mathbf{x}^i) = 0$ wouldn't change, since $\mathbf{w} \cdot h(\mathbf{x}^i) = 10\mathbf{w} \cdot h(\mathbf{x}^i) = 0.0001\mathbf{w} \cdot h(\mathbf{x}^i)$, etc. Indeed, we can always choose a scale for $\mathbf{w}$ to make $\hat{\gamma}$ be equal to any positive constant: we simply rescale $\mathbf{w}$ by $\hat{\gamma}/||\mathbf{w}||$!

We could instead constrain $\mathbf{w}$ so that $||\mathbf{w}|| = 1$, but this constraint is difficult to enforce, since it's not linear. Instead, we can define a geometric margin according to $\gamma = \hat{\gamma}/||\mathbf{w}||$. This removes the extra scaling parameter, so that we can't cheat to get a bigger margin just by scaling all entries in $\mathbf{w}$. Using this formulation, we can write a max-margin optimization problem as follows:

$$\begin{aligned} &\max_{\mathbf{w},\gamma} \frac{\hat{\gamma}}{||\mathbf{w}||} \text{ such that} \\ &y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) \geq \hat{\gamma} \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

The constraint forces $\hat{\gamma} = \min_i y^i(\mathbf{w} \cdot h(\mathbf{x}^i))$, since we must have all $y^i(\mathbf{w} \cdot h(\mathbf{x}^i))$ to be at least as large as $\hat{\gamma}$, and because we want $\hat{\gamma}$ to be small, it must take on the value of the smallest margin.

Recall, however, that we can always rescale $\mathbf{w}$ such that $\hat{\gamma}$ can be equal to whatever we want, so this degree of freedom is *rundant*: if we change $\hat{\gamma}$ for example by multiplying by 10, we can get the same exact margin by multiplying each entry in $\mathbf{w}$ by 10. That means that we can fix $\hat{\gamma}$ to a constant without loss of generality. In particular, we can set $\hat{\gamma} = 1$, and then solve

$$\begin{aligned} &\max_{\mathbf{w}} \frac{1}{||\mathbf{w}||} \text{ such that} \\ &y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) \geq 1 \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

Maximizing $\frac{1}{||\mathbf{w}||}$ corresponds to minimizing $||\mathbf{w}||$, which in turn is the same as minimizing $||\mathbf{w}||^2 = \sum_k \mathbf{w}_k^2$, so we can rewrite the problem again as follows:

$$\begin{aligned} &\min_{\mathbf{w}} ||\mathbf{w}||^2 \text{ such that} \\ &y^i(\mathbf{w} \cdot h(\mathbf{x}^i)) \geq 1 \quad \forall i \in \{1, \ldots, N\} \end{aligned}$$

3 Support vector machines

Now we are ready to define the support vector machine (SVM) optimization problem. There are two small changes we need to make: first, with SVMs, we typically write out the bias term separately from the dot product, and do not minimize the bias (this turns out to the work fine: the magnitude of the bias never affects the size of the margin because it affects all datapoints equally). Second, we'll put back in the slack variables that we had in the perceptron, as otherwise the constraints may not be perfectly satisfiable: we can't always find a decision boundary that perfectly separates the positives from the negatives, so we need to penalize misclassifications using the hinge loss, and multiply the slack variables by some constant λ that determines how much we prefer correct classifications versus maximizing the margin. So the complete SVM optimization is given by

$$\begin{aligned}
&\min_{\mathbf{w},w_0,s_1,\ldots,s_N} ||\mathbf{w}||^2 + \lambda \sum_{i=1}^{N} s_i \text{ such that} \\
&y^i(\mathbf{w} \cdot h(\mathbf{x}^i) + w_0) + s_i \geq 1 \quad \forall i \in \{1,\ldots,N\} \\
&s_i \geq 0 \quad \forall i \in \{1,\ldots,N\}
\end{aligned}$$

This is simply a variant of the perceptron from before that also tries to maximize the size of the margin. The reason this method is called the support vector machine is because the boundary depends on only a small number of "support vectors" (datapoints) that are close to the boundary: either points that are classified incorrectly, or points that are classified correctly but for which the margin is exactly $\frac{1}{||\mathbf{w}||}$. Moving these points will change the decision boundary, but moving any other point will not.