Perceptron

CS 360 Machine Learning

Week 8, Day 1

March 19, 2024

Contents

1 Perceptron **1**

1.1 Perceptron model 2

1.1.1 Example 2

1.2 Perceptron algorithm 3

2 Support Vector Machines (SVMs) **4**

1 Perceptron

Perceptrons were inspired by the brain and are the genesis of neural networks. The basic idea is that as we receive inputs from our environment, a neuron either fires or not (binary output) based on these inputs. A neural network follows this same idea as well. The basic structure of a perceptron is that you have p inputs (with a fake constant 1), weights on various features, those weighted features are added together, and finally a step function determines the binary output. We'll talk about what this step function is in a bit.

The perceptron was invented in 1943 and first implemented in 1957 by Frank Rosenblatt. This was an early wave of the notion that AI will take over the world and solve everything. In 1969, a book called Perceptrons by Marvin Minsky and Seymour Papert identified limitations of the perceptron; patterns which the perceptron was unable to accurately model. These were seen by many incorrectly as irredeemable problems with the algorithm (and with AI itself). After the enthusiasm about the model and AI in general, the discovery of such limitations, and the gap between the expectations and the reality of the existing algorithms and implementations, led to what was known as an "AI winter," when there was less enthusiasm, funding, and research focus on AI.

So what is the perceptron? Suppose that we have points with positive and negative binary classification labels, with positive denoted by 1 and negative denoted by -1. In 2-dimensional space, we might want to draw a line separating the space containing positive examples from negative examples. There could be many such possible boundaries to classify these points, and in high dimensional space these separating boundaries are known as a *separating hyperplane*. We might imagine that there is some best separating hyperplane that nicely goes through the middle to divide the space. But for the perceptron algorithm we'll be happy with *any* separating hyperplane. Note that in our figure examples (see slides) it looks like a line – why do we call it a hyperplane? It's because in general we'll actually be working with very high dimensional data, we just have trouble imagining higher dimensional space, so when we imagine it on a plane it's just a line, and in 3-dimensions we can think of it as a plane, or a giant piece of paper, separating the points. In higher dimensional space, this separating boundary is known as a hyperplane; the goal of the perceptron algorithm is to find such a hyperplane that successfully has positive training data points on one side of the hyperplane and negative labeled training points on the other.

Does our training data always have to have a separating hyperplane that nicely divides the classes? No – data might not be naturally divided in this way. We also might find that various choices of the separating hyperplane could perform differently when given new (test) data. There are datasets that a perceptron cannot learn, for example the XOR function, since this has points in the first and third quadrants with the same classification, and points in the second and fourth quadrants with a different matching classification (see slides for visualization). This is an example of data that is not linearly separable, which has historically meant that people thought the algorithm wasn't powerful enough – but it turns out there are workarounds to handle this concern.

The perceptron algorithm is guaranteed to converge to a solution if a separating hyperplane exists, but it doesn't mean that it will find a *good* separating hyperplane. However if a separating hyperplane doesn't exist, like in the case of the XOR function, there are no guarantees about the behavior of the algorithm.

1.1 Perceptron model

Now let's introduce this perceptron model formally. Let the label be a binary classification $y \in \{-1, 1\}$ and let $\vec{x} \in \mathbb{R}^p$. The model is

$$h(\vec{x}_i) = sign(\vec{w} \cdot \vec{x}_i)$$

where *sign* is a step function such that all negative input values return -1, an input of 0 outputs 0, and positive input values return 1. In other words for $\vec{w} \cdot \vec{x} > 0$ we predict $\hat{y} = 1$ and for $\vec{w} \cdot \vec{x} \leq 0$ we predict $\hat{y} = -1$ (following the convention that zero values receive the negative classification).

Now consider the angle between $\vec{x}$ and $\vec{w}$ and denote it as θ. Note that $\vec{w}$ is perpendicular to the hyperplane. The norm of $\vec{w}$ is denoted and defined as:

$$||\vec{w}|| = \sqrt{w_1^2 + w_2^2 + ... + w_p^2}$$

and is a measure of the distance of $\vec{w}$ from the origin in the vector space. The norm of $\vec{x}$ is denoted and defined similarly. We can think of $\vec{w} \cdot \vec{x} = ||\vec{w}||\ ||\vec{x}|| \cos(\theta)$. We care about the sign of the cosine of the angle between $\vec{x}$ and $\vec{w}$; if $\cos(\theta) > 0$, $\vec{x}$ is on the same side of the hyperplane as $\vec{w}$ so we classify it as positive, if $\cos(\theta) < 0$, then $\vec{x}$ is on the opposite side so becomes negative.

Once trained, the resulting weights $\vec{w}$ represent the hyperplane separating the points. The first entry in the weight vector is known as the bias, and we can think about it as analogous to (though not the same as) the y-intercept of a line; though it's different, the bias also captures a shift away from the origin. The remaining entries of the weight vector represent the directionality of $\vec{w}$ perpendicular to the hyperplane. One of the nice things about a linear model like perceptron is that "for free" we get information about how important each feature is for the outcome; these are the per-feature weights.

1.1.1 Example

Suppose that

$$\vec{w} = \begin{bmatrix} -5 \\ 2 \\ 1 \end{bmatrix}$$

This means that the hyperplane equation is $-5 + 2x_1 + x_2 = 0$, which can be rewritten as $x_2 = 5 - 2x_1$ or the line with y-intercept of 5 and a slope of -2. We can think of

$$\begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

as the direction of $\vec{w}$ which is perpendicular to the hyperplane. The bias is -5 and the opposite of that is the y-intercept.

Consider the example point

$$\begin{bmatrix}1\\5\end{bmatrix}$$

we would add on the fake 1 and have the vector

$$\vec{x} = \begin{bmatrix}1\\1\\5\end{bmatrix}.$$

Then

$$\vec{w}\cdot\vec{x} = \begin{bmatrix}-1\\2\\1\end{bmatrix}\cdot\begin{bmatrix}1\\1\\5\end{bmatrix} = 2$$

so $\hat{y} = 1$. Note that the point $\vec{x}$ is on the side of the hyperplane pointing in the direction of $\vec{w}$, which is the positive classification side of the hyperplane.

Consider the example point $(-3, -2)$ which becomes

$$\vec{x} = \begin{bmatrix}1\\-3\\-2\end{bmatrix}.$$

Computing

$$\vec{w}\cdot\vec{x} = \begin{bmatrix}-5\\2\\1\end{bmatrix}\begin{bmatrix}1\\-3\\-2\end{bmatrix} = -13$$

so $\hat{y} = -1$. This point is on the opposite side from the $\vec{w}$ direction and so the classification is negative.

1.2 Perceptron algorithm

In order to identify a separating hyperplane, we can use the perceptron algorithm. We start by setting the weight vector equal to the zero vector: $\vec{w} = \vec{0}$. We repeat the following steps until all training data is correctly classified. Note that this assumes that some separating hyperplane exists.

Repeat until all training data is correctly classified:

1. select a random training example point $(\vec{x}_i, y_i)$
2. if $\vec{x}_i$ is correctly classified, do nothing.
   else: $\vec{w} \leftarrow \vec{w} + y_i\vec{x}_i$

Note that the update case is somewhat similar to a gradient descent, and is very simple!

The loss function we consider here is known as *surrogate loss* or *hinge loss*:

$$J(\vec{w}) = \sum_{i=1}^{n} max\Big(0, -y_i(\vec{w}\cdot\vec{x}_i)\Big)$$

If y_i and $\vec{w}\cdot\vec{x}_i$ have the same sign, then the prediction was correct and the resulting loss will be 0. If they have opposite signs, then cost is added to the cost function. That cost will be linearly increasing with the magnitude of $y_i(\vec{w} \cdot \vec{x}_i)$ and is essentially adding up the misclassified points, weighted by how far they are from the hyperplane. Per misclassified point, this update will pull the weight vector towards the misclassified point just enough to shift the hyperplane so that the point receives the correct classification.

This loss function is not differentiable because of the $x = 0$ point, however it is differentiable along the two segments separated by the y-axis. In the cases where the prediction is incorrect, we can take the derivative for that portion of the loss function:

$$\nabla J_{\vec{x}_i}(\vec{w}) = -y_i\vec{x}_i$$

Recall that in gradient descent we always had $\vec{w} \leftarrow \vec{w} - \alpha\nabla J_{\vec{x}_i}$ as the update step in our algorithm. If we plug in this derivative, we get:

$$\vec{w} \leftarrow \vec{w} + \alpha y_i\vec{x}_i$$

which is very close to our update step from above:

$$w \leftarrow \vec{w} + y_i\vec{x}_i$$

but missing the α. Typically for this algorithm we just set $\alpha = 1$, since α would only scale the resulting $\vec{w}$, which doesn't change whether a hyperplane is found by the algorithm, though may change the specific hyperplane found and the number of steps it takes to identify the hyperplane. The resulting hyperplane may also depend on the order of the given training example points.

Ideally, we would have a version of this algorithm that identifies a "good" hyperplane such that it's as far as possible from points close to the boundary. We'd also like something that will still provide a reasonable separating hyperplane if the points are not linearly separable. We also might find it useful to be able to create non-linear separators. SVMs will allow us all of these advantages.

2 Support Vector Machines (SVMs)

Before neural networks, SVMs were considered the best "off the shelf" binary classifier and were useful for a wide variety of datasets. They handle all of the wishlist ideas from above, and are still widely used in many fields. In 1963 the initial idea was developed, and there were critical improvements made to the idea in the early 1990s. SVMs will find the best hyperplane, i.e. the one with a large distance between the closest training data point of either label (the points on the "margin") and the hyperplane.

One main idea in SVMs are the support vectors. The *support vectors* are the vectors from the training data points to the separating hyperplane for the points that lie on the margin of the hyperplane.

The model is the same as the one from perceptron, i.e.:

$$h(\vec{x}) = sign(\vec{w} \cdot \vec{x} + b)$$

Note that in this notation we separate out the bias b and won't have fake 1s added to the terms. This is the same model, but expressed differently. Each of $\vec{x}$ and $\vec{w}$ are of length p.

The *functional margin* is defined as:

$$\hat{\gamma}_i = y_i(\vec{w} \cdot \vec{x}_i + b)$$

Note that this is very similar to our erlier loss function. The *geometric margin*, denoted γ_i, is the distance between the closest training point and the hyperplane. This is what we'd like to maximize, for the point with the minimum such γ_i. Next time, we'll see how to optimize for this.