Notes on perceptrons

Ernest Davis

November 21, 2018

A perceptron is a classifier for a binary category using numerical predictive attributes. That is, suppose that an instance $q$ is characterized in terms of a collection of $M$ numerical predictive attributes $\langle q_1 \ldots q_M \rangle$. The task is to determine whether $q$ is member of some category $C$.

Examples

Instance $q$ is a gray scale image with $M$ pixels. $q_i$ is the light intensity at the $i$th pixel. $C$ is the category of images of cats.

Instance $q$ is a color image with $M/3$ pixels. $q_{3i-2}$, $q_{3i-1}$, $q_{3i}$ are respectively the R,G,B intensity at the $i$th pixel. $C$ is the category of images of cats.

Instance $q$ is the text of a online review of a product. $M$ is the number of words in the language. $q_i$ is the number of times word $w_i$ occurs in text $q$. $C$ is the category of favorable reviews.

Instance $q$ is a patient. A collection of $M$ blood tests have been performed. $q_i$ is the level of the $i$th substance in the blood. $C$ is the category of patients with diabetes.

How the perceptron classifies an instance

The perceptron consists of a collection of weights $\langle w_1 \ldots w_M \rangle$ and a threshold $t$ and computes the following function. In summation notation, we can rewrite this

\[
\text{if } w_1q_1 + w_2q_2 + \ldots + w_Mq_m > t \quad \text{ output YES.}
\]

\[
\text{else} \quad \text{output NO.}
\]

In summation notation, we can rewrite this

\[
\text{if } \sum_{i=1}^{M} w_iq_i > t \quad \text{ output YES.}
\]

\[
\text{else} \quad \text{output NO.}
\]

To simplify the discussion of learning below, the following notation will be convenient:

Let $N = M + 1.$

Extend each vector of values $\vec{q} = \langle q_1 \ldots q_M \rangle$ by defining $q_N = -1$.

Define $w_N = t$, the threshold.

\[
\text{if } \sum_{i=1}^{N} w_iq_i \quad \text{ output YES.}
\]

\[
\text{else} \quad \text{output NO.}
\]
In vector notation, this can be written even more compactly:
\[
\text{if } \vec{w} \cdot \vec{q} > 0 \text{ output YES else output NO}
\]
where \( \vec{w} \) is the vector \( \langle w_1 \ldots w_N \rangle \), \( \vec{q} \) is the vector \( \langle q_1 \ldots q_N \rangle \), and \( \vec{w} \cdot \vec{q} \) is the dot product.

**Geometric example**

The instances are points in the plane; thus \( M = 2 \), \( q_1 \) is the \( x \)-coordinate of point \( q \) and \( q_2 \) is the \( y \)-coordinate of point \( q \). Points in the target category are green; points outside the category are blue. Points A-F are in the training set; their label is known. Points G-J are test points; their label is unknown.

The line shows the perceptron corresponding to the inequality \(-x + y > -2\). Thus for this perceptron \( w_1 = -1, w_2 = 1, w_3 = -2 \). This perceptron correctly classifies points A and E as green and B and C as blue. It incorrect classifies D as green and F as blue. It categories test points G and H as green and I as blue.

The corresponding training set is this:

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>1</td>
<td>YES</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>1</td>
<td>NO</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
<td>2</td>
<td>NO</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>3</td>
<td>NO</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>4</td>
<td>YES</td>
</tr>
<tr>
<td>F</td>
<td>6</td>
<td>3</td>
<td>YES</td>
</tr>
</tbody>
</table>
Gradient descent: Digression

Suppose that you have a real-valued function \( f(x_1 \ldots x_k) \) from \( \mathbb{R}^k \) to \( \mathbb{R} \). You wish to find a value of \( x_1 \ldots x_k \) that minimizes \( f \). Assume that \( f \) is continuous and piecewise differentiable. Then the method of gradient descent may give good results:

\[
\text{function } \text{gradientDescent}(f: \text{function from } \mathbb{R}^k \text{ to } \mathbb{R}) \text{ returns vector in } \mathbb{R}^k \{ \\
\quad \vec{x} \in \mathbb{R}^k = \text{some starting value (e.g. random)}; \\
\quad \delta = \text{some small positive real number}; \\
\quad \text{loop} \\
\quad \quad \vec{g} = \vec{\nabla}f|_{\vec{x}} \\
\quad \quad \vec{x} = \vec{x} - \delta \cdot \vec{g} \\
\quad \text{until (some termination condition is reached)} \\
\quad \text{return } \vec{x} 
\}
\]

Here \( \vec{\nabla}f|_{\vec{x}} \) is the gradient of function \( f \) evaluated at point \( \vec{x} \). The gradient points in the direction in \( \mathbb{R}^k \) in which \( f \) increases most rapidly. Its magnitude is the speed with which \( f \) increases if you go in that direction.

The gradient is computed as

\[
\nabla f = \frac{\partial f}{\partial x_1} \hat{x}_1 + \ldots + \frac{\partial f}{\partial x_k} \hat{x}_k
\]

So, intuitively, the current state is a vector \( \vec{x} \). The gradient is the direction in which \( f \) increases fastest. You take a step of size \( \delta \) in the opposite direction; so you are stepping in the direction in which \( f \) decreases. Iterate. Eventually, you (hopefully) find your way to the bottom.

There are a number of different possible termination conditions:

- The magnitude of the gradient \( \vec{g} \) is small.
- The change in \( f \) when you change the value of \( \vec{x} \) is small.
- You do this for a fixed number of iterations.

The choice of the value of \( \delta \) is an art in itself. Above, we’ve written it as if it is a constant, but actually you may change it in the course of the algorithm. All this kind of thing is studied in great depth in the field of numerical optimization.

Back to perceptrons

We are looking for the perceptron that does the best job of categorizing the training set \( T \). We proceed as follows: we define an error function \( E_T(\vec{p}) \) which measures how bad a job the perceptron \( \vec{p} \) does on the training set \( T \). We then use gradient descent to find a value of \( \vec{p} \) that (hopefully) minimizes \( E_T(\vec{p}) \).

The obvious error function would be the inaccuracy of the perceptron; the fraction of the points that it misclassifies. Unfortunately, that’s not a function where you can use gradient descent, because it’s a step function. The gradient of \( f \) at point \( \vec{x} \) describes how the value of \( f \) changes if you make
an infinitesimal change to $\vec{x}$. But most of the time, if you make a small change to the perceptron vector, that doesn’t change how it classifies any points in the training set, so the accuracy over the training set remains constant, so the gradient of the accuracy is zero. The accuracy of the training set only changes when the perceptron happens to move across one of the training points, and then it changes discontinuously.

So we need a different error function. What we are going to do is as follows:

- We change an error $E_{\vec{q}}(\vec{p})$ for each instance $\vec{q}$ in the training set $T$. We define the total error over the training set to the the sum of the errors for the individual instances: $E_T(\vec{p}) = \sum_{\vec{q} \in T} E_{\vec{q}}(\vec{p})$.
- If perceptron $\vec{p}$ correctly classifies point $\vec{q}$ then the associated error is 0: $E_{\vec{q}}(\vec{p}) = 0$.
- If perceptron $\vec{p}$ misclassifies point $\vec{q}$, then the associated error is the amount by which $\vec{q}$ is misclassified: $E_{\vec{q}}(\vec{p}) = |p_1 \cdot q_1 + \ldots + p_N \cdot q_N| = |\vec{p} \cdot \vec{q}|$.

Alternatively we can write this as follows: Take “YES” to be 1 and “NO” to be 0. Let $L(\vec{q})$ be the label of point $\vec{q}$ in the training set and let $O(\vec{q}, \vec{p})$ be the classification that perceptron $\vec{p}$ assigns to point $\vec{q}$.

Let $s(\vec{q}, \vec{p}) = L(\vec{q}) - O(\vec{q}, \vec{p})$.

Thus $s(\vec{q}, \vec{p}) =$

- 0 if perceptron correctly classifies the point;
- 1 if the point is labeled NO but classified YES; and
- $-1$ if the point is labelled YES but classified NO.

Therefore in all cases $E_{\vec{q}}(\vec{p}) = s(\vec{q}, \vec{p}) \cdot \vec{p} \cdot \vec{q}$.

Putting this altogether, we have $E_T(\vec{p}) = \sum_{\vec{q} \in T} E_{\vec{q}}(\vec{p}) = \sum_{\vec{q} \in T} s(\vec{q}, \vec{p}) \sum_{i=1}^{N} p_i q_i$.

In the above example,

$s(D, p) = 1$.

$s(F, p) = -1$.

The other points are correctly classified, and therefore have $s$ values of 0.

$\vec{p} \cdot D = (-1) \cdot 2 + 1 \cdot 3 + (-2) \cdot (-1) = 3$.

$\vec{p} \cdot F = (-1) \cdot 6 + 1 \cdot 3 + (-2) \cdot (-1) = -1$.

Therefor the total error $E_T(\vec{p}) = 1 \cdot 3 + -1 \cdot -1 = 4$.

Switching the order of summation we have $E_T(\vec{p}) = |\sum_{\vec{q} \in T} s(\vec{q}, \vec{p}) \cdot q_1| \cdot p_1 + \ldots |\sum_{\vec{q} \in T} s(\vec{q}, \vec{p}) \cdot q_N| \cdot p_N|$.

Since the values $s(\vec{q}, \vec{p})$ are constant under small changes to $\vec{p}$ (they only change when $\vec{p}$ changes its classification of $\vec{q}$), this is, locally, just a linear function of the $p_i$s. Taking the partial derivatives therefore couldn’t be easier:

$\frac{\partial E}{\partial p_i} = \sum_{\vec{q} \in T} s(\vec{q}, \vec{p}) \cdot q_i$.
Gradient descent for perceptrons

(Those who skipped the last section should come back.)

We can therefore write the overall perceptron learning algorithm as follows:

```plaintext
function s(Instance q, Perceptron p, int N) return Real }
    Real dp = 0;
    for (i = 1 .. N) dp += q.values[i] * p[i];
    if ((dp < 0) && q.label) return -1;
    if ((dp > 0) && !q.label) return 1;
    else return 0;
}

function grad(TrainingSet T, Perceptron p, int N) return N-dimensional-vector
    Vector g = N-dimensional 0 vector;
    for (Example q in T)
        g += s(q,dp) * q.values;
    return g;

function learnPerceptron(TrainingSet T) return Perceptron {
    M = number of predictive attributes in T;
    N = M+1;
    Perceptron p = starting value (an N dimensional vector);
    Real delta = small number;
    add to T an Nth column all -1;
    loop
        p = p - delta * grad(T,p,N)
        until (termination condition)
}
```

Note that we never have to compute the value of the error function, unless that is part of the termination condition.

One iteration from the above example

Let's take the above example, and do one iteration, with δ = 0.05. We have:

So the gradient \(\tilde{g} = s(D,p) * D + s(F) * F = \langle 2, 3, -1 \rangle - \langle 6, 3, -1 \rangle = \langle -4, 0, 0 \rangle\).

So the new perceptron is \((-1, 1, -2) - 0.05 \cdot \langle -4, 0, 0 \rangle = \langle -0.8, 1, -2 \rangle\), the line \(-0.8x + y > -2\).

Under the new perceptron, only D is incorrectly classified. The total error is 3.4.
Stochastic gradient descent

An alternative approach is at each stage to pick a misclassified training point $\vec{q}$ at random and adjust $\vec{p}$ by $-\delta \cdot \vec{q}$. This is harder to justify in terms of gradient descent but simpler to implement and often works better than the previous algorithm (called “batch” gradient descent. The algorithm is as follows:

```plaintext
function s(Instance q, Perceptron p, int N) return Real {
    Real dp = 0;
    for (i = 1 .. N) dp += q.values[i] * p[i];
    if ((dp < 0) && q.label) return -1;
    if ((dp > 0) && !q.label) return 1;
    else return 0;
}
```

```plaintext
function learnPerceptronStochastic(TrainingSet T) return Perceptron {
    M = number of predictive attributes in T;
    N = M+1;
    Perceptron p = starting value (an N dimensional vector);
    Real delta = small number;
    add to T an Nth column all -1;
    loop {
        Instance q = a random element of T;
        Real sg = s(q,p,N)
        p = p - sg*delta*q;
    }
    until (termination condition)
}
Convergence Theorem

Let $S = \langle q_1, q_2 \ldots \rangle$ be an infinite sequence of elements of $T$ (with repetitions) such that every element of $T$ appears infinitely often in $S$. For instance a round robin of $T$ has this property; a sequence of random choices has this property with probability 1.

If:

a. the category $C$ is linearly separable;

b. the procedure $\text{learnPerceptronStochastic}$ follows sequence $S$ in choosing the vectors $q$

then, for any $\delta > 0$, the perceptron $p$ will eventually be a perfect classifier for $C$. 