CS 337, Fall 2023
The Perceptron Classifier

Scribes: Atharva Tambat, Balaji Siddardha, Nikhil Biradar, Anand Narasimhan, B Prabandh, Adithya Gautam
(***Equal contribution by all scribes***)
Edited by: Dhiraj Kumar Sah

September 4, 2023

Recap

We aim to estimate decision boundaries (hyper-planes) that best separate the training data if they are linearly separable. We have already seen one log-linear model - Logistic Regression. We will look at Perceptron now. Although it isn't used in practice directly, it is the building block for SVM classifiers (with non-linearity introduced) and Artificial Neural Networks (although with different update rules).

1 Introduction

Perceptron: A linear model of classification that estimates a hyperplane that 'best' separates the training data.

Hyperplanes are represented by the equation $\mathbf{w}^T\mathbf{x} = 0$, where $\mathbf{w}$ is a column vector of weights (including the bias) and $\mathbf{x}$ being a column vector of inputs (with 1 prepended to the $\mathbf{x}_i$s).

Recall: Equation of hyperplane:

$$\mathbf{w}^T\mathbf{x} = 0$$

where, $\mathbf{w}$ is the orthogonal vector to the plane.
The Perceptron classifier, at test time, predicts a test instance to have the label $h(x)$

$$h(x) = sign(\mathbf{w}^T\mathbf{x})$$

where,

$$sign(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \tag{1}$$

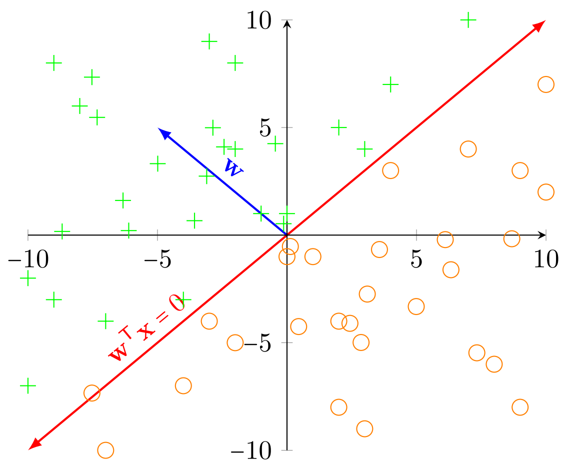

Figure 1: Linear model of classification

2 Properties of the Perceptron Model

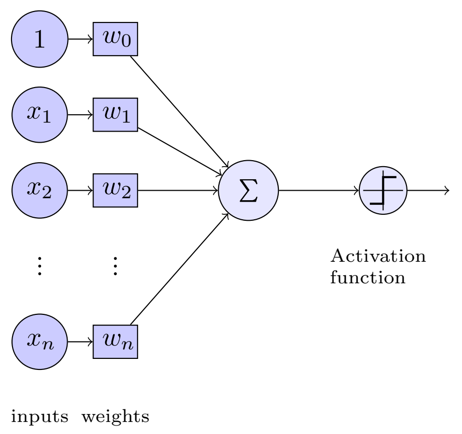

Figure 2: The Perceptron Model

2.1 Neuron-Like

A perceptron is a basic unit of the **Neural model of Learning**. It's very loosely based on how a neuron works. The junctions between the dendrite and axons consist of multiple electrical signals being modulated differently. This is quite similar to the inputs receiving different weights to be multiplied. This can be evolved into a basic feed-forward artificial neural network by adding many more such units and deciding on appropriate (mostly non-linear) activation functions.

A **multi-layered perceptron** (MLP) is a special case of the above but is often used interchangeably. The slight difference arises in the type of loss function used, which in the case of MLPs is the **Heavside** step function, instead of normal non-linear activation functions.
Also, this is mentioned because these generalise the notion of a perceptron to other types of data as well, not just linearly separable.

2.2 Online Algorithm

The perceptron algorithm uses its inputs (training examples) one at a time, using them to update weights. It does this in the same order in which those examples are supplied and does not revisit them again.

These types of algorithms are called **online** algorithms and are of interest to computer scientists from an algorithmic standpoint.

2.3 Error-driven Algorithm

The algorithm only takes action if one of the training examples is classified incorrectly by the hyperplane specified by the current weight vector. In other words, if the training example is classified correctly, change nothing. If not, update the weight vector. We'll see the updation rule shortly.

3 The Perceptron Update Algorithm

Weight update for perceptron model

```
Inputs: the set of training examples;
Algorithm hyperparameters: number of iterations T;
Initialize weight $\mathbf{w}$ to some $\mathbf{w_0}$
foreach $iteration \in \{1, 2...T\}$ do
    Pick a random training instance $(\mathbf{x}, y)$ where $\mathbf{x} \in \mathbb{R}^d$ and $y \in \{-1, 1\}$;
    Predict $\hat{y} = sign(\mathbf{w}^T\mathbf{x})$;
    if $\hat{y} \neq y$ then
        $\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + y\mathbf{x}$
    end if
end foreach
return $\mathbf{w_T}$
```

The update equation can be made more compact by saying:

```
Let $(\mathbf{x'}, y')$ be one training instance at iteration $t$, where weight is $\mathbf{w_t}$
if $y' \cdot \mathbf{w_t}^T\mathbf{x'} \leq 0$ then
    $\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + y'\mathbf{x'}$;
else
end if
```

The above modification implies that if the signs of y' (label) and $\mathbf{w_t}^T\mathbf{x}$ (scaled predicted label) are not aligned, then the weights should be updated (Note that weights should also be updated on $y' \cdot \mathbf{w_t}^T\mathbf{x'} = 0$ because the weights may be initialized to 0, so they should definitely be updated in the first iteration).

- To ensure that all instances are gone over at least once without repetition, the training set is shuffled and sequentially gone through - avoiding picking the same data repeatedly.
- It is common to take an average of weights $\mathbf{w}$ over training iterations because updates in the latter part of the training may have changed the weight to miss-classify some samples in the initial training.

4 Intuition of the weight update rule

Weight update rule (for in instance $(x, y) \in \mathcal{D}_{train}$) of the perceptron:

$$\mathbf{w_{t+1}} \leftarrow \begin{cases} \mathbf{w_t} + \mathbf{x}, & \text{if } y = 1 \\ \mathbf{w_t} - \mathbf{x}, & \text{if } y = -1 \end{cases} \tag{2}$$

The intuition behind the update rule is that the weight vector ($\mathbf{w_{t+1}}$) is moved towards or away from the current weight ($\mathbf{w_t}$), depending on whether its label is positive or negative respectively. As shown below, the train instance $\mathbf{x}$ will be correctly classified (shifted to the correct side of the hyperplane) after the update.

Also, the condition can be replaced by checking whether $y_i \mathbf{w}_t^T x_i \leq 0$ and performing the update rule if it is. This quantity will also be used later as a metric to show algorithm improvement.

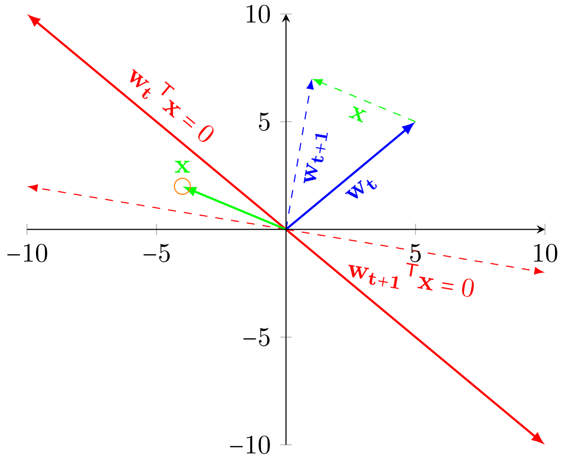

Figure 3: Visualization of the perceptron weight update rule

5 Does the perceptron weight update rule improve performance?

Is the perceptron update algorithm **guaranteed** to improve after a weight update on an erroneous training sample? **No**, it is **not guaranteed** to become correct on the erroneous example immediately after the update. It might work correctly at some later point but, it isn't assured. Let us consider a misclassified example $(\mathbf{x}, y) \in \mathcal{D}_{train}$ i.e. $sign(\mathbf{w_{old}}^T\mathbf{x}) \neq y$. According to the perceptron weight update rule,

$$\mathbf{w_{new}} \leftarrow \mathbf{w_{old}} + \mathbf{x}y$$

$$\begin{aligned} \Rightarrow \quad & y \cdot \mathbf{w^T_{new}}\mathbf{x} = y \cdot (\mathbf{w_{old}} + y\mathbf{x})^T\mathbf{x} \\ \Rightarrow \quad & y\mathbf{w_{new}}^T\mathbf{x} = y(\mathbf{w_{old}}^T\mathbf{x}) + y^2\|\mathbf{x}\|_2^2 > y\mathbf{w_{old}}^T\mathbf{x} \end{aligned} \tag{3}$$

We know that $y\mathbf{w_{old}}^T\mathbf{x} \leq 0$, and $y\mathbf{w_{new}}^T\mathbf{x} > y\mathbf{w_{old}}^T\mathbf{x}$. Our goal is to make $y\mathbf{w_{new}}^T\mathbf{x}$ positive (which implies the label is correctly predicted), or at least closer to that (less negative). As shown above, the weight update rule does exactly that. Note that the update does not guarantee that the label for that example will be correctly predicted. The new value of $y\mathbf{w_{new}}^T\mathbf{x}$ might still be negative (but lesser in magnitude, hence closer to being positive).

So by this, we would assume that the order of training examples is critical since there is no convergence guarantee.

It turns out though, that the perceptron is guaranteed to converge if the data is **linearly separable!**

Perhaps even more surprisingly, the number of iterations does not depend on things like n or d, it just depends on an upper bound of the norm of the points, call it D, and a possible margin of separation γ. The order of iterations that must be made is $\mathcal{O}(D^2/\gamma^2)$

A note on γ: If there exists a hyperplane that correctly classifies the training points such that for both the set of positive and negative examples separately, the minimum distance between the points and the hyperplane is $\geq \gamma$, then the hyperplane is said to have margin of separation γ.

Intuition says that during an update, it could also end up misclassifying previously correctly classified examples, but that's the beauty of the perceptron. The algorithm will find an appropriate classifier if the points are linearly separable.

Aside: In-Class Discussions

- During the update, the bias in $\mathbf{x}$ will only be updated by ± 1 with the above update rule, which is undesirable. To avoid this, the update is modified by a factor α, i.e.

$$\mathbf{w_{new}} \leftarrow \mathbf{w_{old}} + \alpha \mathbf{x} y$$

- While training on the training set, the perceptron can misclassify some previously correctly classified data due to updates while training on the remaining subset of training data. So, how are we guaranteeing that it is improving? Wait for the next class where convergence guarantees will be discussed.
- If one sees that the error is high after training, then the training data is reshuffled and trained over again. The same training sample is not to be iterated over and over again. One common scheme used in practice is to take an average of weights over all iterations.

6 What is the perceptron algorithm minimizing?

A natural objective is to **minimize** in classification is the **number of miss-classified examples**, or equivalently, maximizing $y\mathbf{w^T x}$ for all $(\mathbf{x}, y) \in \mathcal{D}_{train}$, with the intuition of making this quantity as positive as possible for as many test instances as possible. Let $\mathcal{D}'$ be the set of miss classified instances $(\mathbf{x}, y) \in \mathcal{D}_{train}$

$$min \sum_{(\mathbf{x},y)\in\mathcal{D}'} -yw^T x$$

The perceptron loss for an instance $(\mathbf{x}, y)$ (whether or not it is miss-classified or not):

$$max\{0, -y\mathbf{w}^T\mathbf{x}\}$$

So, the loss for all training data is:

$$L_{perc}(\mathbf{w}, \mathcal{D}_{train}) = \sum_{(\mathbf{x},y)\in\mathcal{D}_{train}} max\{0, -y\mathbf{w}^T\mathbf{x}\}$$

Figure 4 shows the perceptron loss specified above:

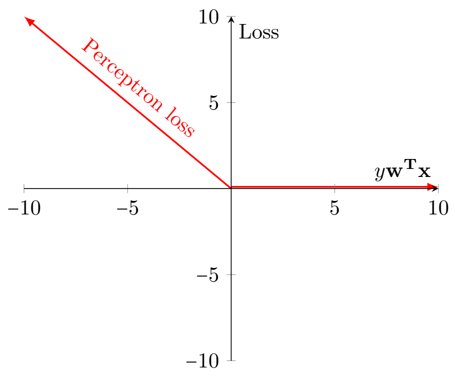

Figure 4: Graph of the perceptron loss function for a single training instance

6.1 Optimizing the loss function

The loss is convex hence, Stochastic Gradient Descent should work for optimizing this loss function. Except for where $y\mathbf{w}^\mathbf{T}\mathbf{x} = 0$, where the loss function is non-differentiable. At these points, subgradients are calculated instead of gradients.

Aside: Subgradients

A subgradient of a convex function $f(\mathbf{w_0})$ is all the vectors $\mathbf{g}$ s.t. for any other point $\mathbf{w}$, we have $f(\mathbf{w}) - f(\mathbf{w_0}) \geq \mathbf{g}^T(\mathbf{w} - \mathbf{w_0})$. Subgradient reduces to gradient when the function is differentiable.

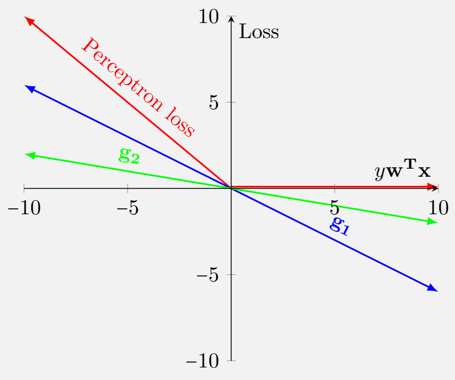

Figure 5: Graph of sub-gradients of the perceptron loss at $y\mathbf{w}^\mathbf{T}\mathbf{x} = 0$

It can be proved that if f is convex and is differentiable at w_0, then its gradient is its only subgradient. The converse can also be proved. Using these subgradients instead of normal gradients, we can run SGD to minimise the loss and obtain a good linear classifier for the data.