CS 446: Machine Learning
Lecture 4, Part 2: On-Line Learning

0.1 Linear Functions

So far, we have been looking at Linear Functions as a class of functions which can separate some data and not others. $f(x) = \begin{cases} 1 & if\ W_1X_1 + W_2X_2 + \ldots + W_nX_n \geq \Theta \\ 0 & Otherwise \end{cases}$

Linear functions can be used for:
Disjunction:

$$y = X_1 \vee X_3 \vee X_5$$
$$y = (1 \cdot X_1 + 1 \cdot X_3 + 1 \cdot X_5 \geq 1)$$

At least m of n:

$$y = at\ least\ 2\ of\ \{X_1 \vee X_3 \vee X_5\}$$
$$y = (1 \cdot X_1 + 1 \cdot X_3 + 1 \cdot X_5 \geq 2)$$

However, linear functions are not useful for:
Exclusive-OR:

$$y = (X_1 \wedge \bar{X_2}) \wedge (\bar{X_1} \wedge X_2)$$

Non-trivial DNF:

$$y = (X_1 \wedge X_2) \vee (\bar{X_3} \wedge X_4)$$

0.2 Perceptron Learning Rule

Perceptron is an on-line, mistake driven algorithm. Rosenblatt (1959) suggested that when a target output value is provided for a single neuron with fixed input, it can incrementally change weights and learn to produce output using the Perceptron learning rule. The perceptron equals the Linear Threshold Unit.

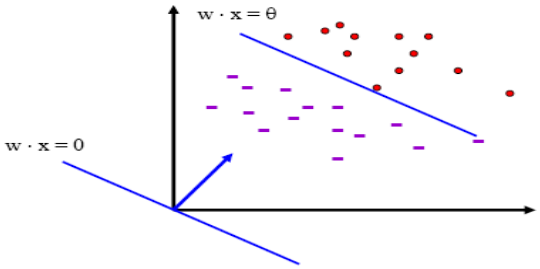

Figure 1: Linear Function

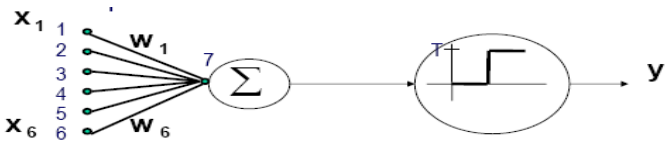

Figure 2: Perceptron

0.3 Perceptron Learning Rule

We learn $f : X \rightarrow \{-1, +1\}$ represented as $f = sgn\{w \cdot x)$, where $X = \{0, 1\}^n$ or $X = R^n, w \in R^n$. Given labeled examples:$\{(x^1, y^2), (x^2, y^2), \ldots, (x^m, y^m)\}$, we sketch the following outline for an algorithm:

1. Initialize $w = 0 \in R^n$
2. Cycle through all examples
   (a) Predict the label of instance *x* to be $y' = sgn\{w \cdot x)$
   (b) If $y' \neq y$, update the weight vector: $w = w + ryx$ (r – a constant, the learning rate)
   Otherwise, if $y' \neq y$, leave weights unchanged.

If x is Boolean, only weights of active features are updated. $w \cdot x > 0$ is equivalent to $\frac{1}{1+exp(w \cdot x)} > \frac{1}{2}$.

Perceptron has no threshold in this algorithm. However, that does not mean that we lose generality:

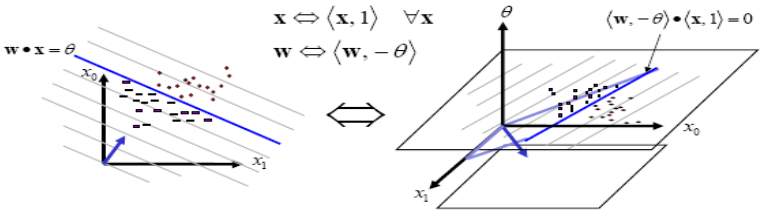

Figure 3: Perceptron Threshold

0.3.1 Perceptron Learnability

Obviously Perceptron cannot learn that which it cannot represent, which means that it can only be used with linearly separable functions. Minsky and Papert (1969) wrote an influential book demonstrating Perceptron's representational limitaions. Among these limitations are that parity functions cannot be learned (XOR) and, in vision, if patterns are represented with local features, then they cannot represent symmetry or connectivity. Research on neural networks stopped for years. Rosenblatt himself(1959) asked, "What pattern recognition problems can be transformed so as to become linearly separable?"

Recall from previous sections, that patterns that are not linearly separable can be represented in higher dimensions and thereby become linearly separable.

0.3.2 Perceptron Covergence

The Perceptron Convergence Theorem states that, If there exists a set of weights that are amenable to treatment with Perceptron (i.e., the data is linearly separable), then the Perceptron learning algorithm will converge.

An interesting question which will be discussed in the following section is how long it takes for the learning algorithm to converge.
The Perceptron Cycling Theorem states that, if the training data is not linearly seperable, then the Perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.

How can we provide robustness and more expressivity?

0.3.3 Perceptron: Mistake Bound Theorem

This maintains a weight vector $w \in \Re^N, w_0 = (0, \ldots, 0)$, and upon receiving an example $x \in^N$, makes a prediction according to the linear threshold function $w \cdot x \geq 0$.

Theorem [Novikoff, 1963] Let $(x_1; y_1), \ldots, : (x_t; y_t)$ be a sequence of labeled examples with $x_i \in \Re^N, ||x_i|| \leq R$ and $y_i \in \{-1, 1\}$ for all i. Let $u \in \Re^N, \gamma > 0$ be such that, $||u|| = 1$ and $y_i u \cdot x_i \geq \gamma$ for all i. Then Perceptron makes at most $||u||^2 R^2 / \gamma_2$ mistakes on this example sequence. In this theorem, γ is the margin complexity parameter.

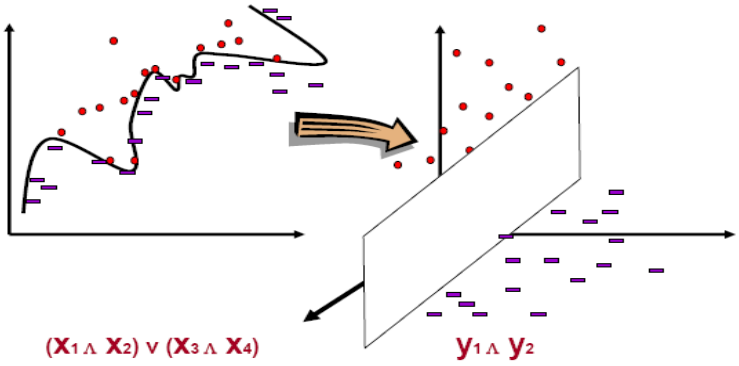

Figure 4: Representing the Pattern in a Higher Dimensionality

Proof: Let v_k be the hypothesis before the $k-th$ mistake. Assume that the $k-th$ mistake occurs on the input example $(\vec{x}_i, \vec{y}_i)$.

$$\therefore y_i(\vec{v}_k \cdot \vec{x}_i) \leq 0$$

Assumptions
$v_1 = 0$
$||u|| \leq 1$
$y_i u \cdot x_i \leq \gamma$

$$\begin{gathered}\vec{v}_{k+1} = \vec{v}_k + y_i\vec{x}_i \\ \vec{v}_{k+1} \cdot \vec{u} = \vec{v}_k \cdot \vec{u} + y_i(\vec{u} \cdot \vec{x}_i) \\ \geq \vec{v}_k \cdot \vec{u} + \gamma \\ \therefore \vec{v}_{k+1} \cdot \vec{u} \geq k\gamma\end{gathered}$$

$$\begin{gathered}\|\vec{v}_{k+1}\|^2 = \|\vec{v}_k\|^2 + 2y_i(\vec{v}_k \cdot \vec{x}_i) + \|\vec{x}_i\|^2 \\ \geq \|\vec{v}_k\|^2 + R^2 \\ \therefore \|\vec{v}_{k+1}\|^2 \leq kR^2\end{gathered}$$

Therefore,

$$\sqrt{k}R \geq ||\vec{v}_{k+1}|| \geq \vec{v}_{k+1} \cdot \vec{u} \geq k\gamma. \quad \Longrightarrow K < R^2/\gamma^2$$

The second inequality follows because $\|\vec{u}\| \leq 1$.

0.3.4 Perceptron for Boolean Functions

It is important to consider how many mistakes the Perceptron algorithms make when learning a k-disjunction, and try to figure out the bound. We can try to find a sequence of examples that will cause Perceptron to make $O(n)$ mistakes on k-disjunction on n attributes.

1 Winnow Algorithm

The Winnow Algorithm is another type of on-line mistake-driven algorithm that learns Linear Threshold Functions. Initialize: $\Theta = n; w_i = 1;$
Prediction is 1 iff $w \cdot x \geq \Theta$
If no mistake: do nothing
If $f(x) = 1$ but $w \cdot x < \Theta, w_i \leftarrow 2w_i$ (if $x_i = 1$) (promotion)

If $f(x) = 0$ but $w \cdot x \geq \theta, w_i \leftarrow w_i/2$ (if $x_i = 1$) (demotion)

For the class of disjunction, it is possible to use *elimination* instead of demotion.

1.1 Winnow - Example

The Winnow Algorithm works as follows on the example below:
$f = x_1 \vee x_2 \vee x_{1023} \vee x_{1024}$
Initialize: $\Theta = 1024; w = (1, 1, \ldots, 1)$
$\langle (1, 1, \ldots, 1), + \rangle \quad w \cdot x \geq \Theta \quad w = (1, 1, \ldots, 1)$ ok
$\langle (0, 0, \ldots, 0), - \rangle \quad w \cdot x < \Theta \quad w = (1, 1, \ldots, 1)$ ok
$\langle (0, 0, 111, \ldots, 0), - \rangle \quad w \cdot x < \Theta \quad w = (1, 1, \ldots, 1)$ ok
$\langle (1, 0, 0, , 0), + \rangle \quad w \cdot x < \Theta \quad w = (2, 1, \ldots, 1)$ mistake
$\langle (1, 0, 1, 1, 0, , 0), + \rangle \quad w \cdot x < \Theta \quad w = (4, 1, 2, 2, \ldots, 1)$ mistake
$\langle (1, 0, 1, 0, 0, , 1), + \rangle \quad w \cdot x < \Theta \quad w = (8, 1, 4, 2, \ldots, 2)$ mistake
. . . . . $log(n/2)$ (for each good variable)
$w = (512, 1, 256, 256, \ldots, 256)$
$\langle (1, 0, 1, 0, \ldots, 1), + \rangle \quad w \cdot x \geq \Theta \quad w = (512, 1, 256, 256, \ldots, 256)$ ok

$\langle (0, 0, 1, 0, 111, \ldots, 0), - \rangle \quad w \cdot x \geq \Theta \quad w = (512, 1, 0, \ldots, 0, \ldots, 256)$ mistake **(elimiation version)**
. . . . . .
$w = (1024, 1024, 0, 0, 0, 1, 32, \ldots, 1024, 1024)$ **(final hypothesis)**

Notice that the same algorithm will learn a conjunction over these variables $(w = (256, 256, 0, \ldots, 32, \ldots, 256, 256))$.

1.2 Winnow - Mistake Bound

Claim: Winnow makes $O(k \; log \; n)$ mistakes on k-disjunctions.
Initialize: $\Theta = n; \; w_i = 1$
Prediction is 1 iff $w \cdot x \geq \Theta$
If no mistake: do nothing
If $f(x) = 1$ but $w \cdot x < \Theta, \; w_i \leftarrow 2w_i$ (if$x_i = 1$) (promotion)
If $f(x) = 0$ but $w \cdot x \geq \Theta \; w_i \leftarrow w_i/2$(if $x_i = 1$) (demotion)

- u – number of mistakes on positive examples (promotions)
- v – number of mistakes on negative examples (demotions)

1. $u < k\ log(2n)$
A weight that corresponds to a good variable is only promoted. When these weights get to n there will be no more mistakes on positives.

2. $v < 2(u + 1)$
Total weight: $TW = n$
Mistake on positive: $TW(t + 1) < TW(t) + n$
Mistake on negative: $TW(t + 1) < TW(t) = n/2$
$0 < TW < n + un - vn/2 \Longrightarrow v < 2(u + 1)$

Number of mistakes: $u + v < 3u + 2 = O(k\ log\ n)$

1.3 Winnow - Extensions

This algorithm learns montone functions.
For the general case:

- Duplicate variables: For the negation of variable x, introduce a new variable y and learn monotone function over $2n$ variables.
- Balance version: Keep two weights for each variable. The effective weight is the difference.

Update rule:
If $f(x) = 1$ but $(w^+ - w^-) \cdot x \leq \Theta, \quad w_i^+ \leftarrow 2w_i^+ \ \ w_i^- \leftarrow \frac{1}{2}w_i^-$
where $x_i = 1$(promotion)
If $f(x) = 0$ but $(w^+ - w^-) \cdot x \geq \Theta, \quad w_i^+ \leftarrow frac12w_i^+ \ \ w_i^- \leftarrow 2w_i^-$
where $x_i = 1$ (demotion)

SNoW is a version of Winnow that supports the general case. (Infinite attribute domain)
Multi Class predictor – one vs. all vs. the rightway. . . (later)

1.4 Winnow - A Robust Variation

Winnow is robust in the presence of various kinds of noise such as classification noise and attribute noise. The target function changes with time and can be thought of as a 'Moving Target'. This is helpful when we learn under some distribution but test under a slightly different one. For example, in Natural Language applications where the data might change significantly based on the time of its collection.
For the sake of Modeling, it is possible to think of the Winnow Algorithm in terms of an Adversary and Learner:

- Adversary's turn: May change the target concept by adding or removing some variable from the target disjunction. The cost of each addition move is [1.
- Learner's turn: Makes prediction on the examples given and is then told the correct answer (according to the current target function).

There is a variation of Winnow, *Winnow-R*, which is the same as Winnow, only it doesn't let the weights go below 1/2. The claim is that Winnow-R makes $O(c\ log\ n)$ mistakes, (c - cost of adversary). This is a generalization of previous claim.

1.5 Winnow-R Mistake Bound

u - number of mistakes on positive examples (promotions).
v - number of mistakes on negative examples (demotions).

2. $v < 2(u + 1)$
Total weight $TW = n$ initially.
Mistake on positive: $TW(t+1) < TW(t)+n$ Mistake on negative: $TW(t+1) < TW(t) - n/4$

2 Algorithmic Approaches

We will focus on two families of algorithms, one of them representing on-line algorithms. The first are *Additive update algorithms* of which Perceptron is an example. Also, *SVM*, which is not on-line, is a close relative of Perceptron and an Additive update algorithm. The second family is the family of *Multiplicative*

update algorithms. An example is SNoW and close relatives include Boosting and Max Entropy.

2.1 Which Algorithm to Choose?

There are various questions which arise in determining which algorithm to choose. First we consider *Generalization*.

The l_1 norm: $\|x\|_1 = \Sigma_1 |x_i|$ The l_2 norm: math$\|x\|_2 = (\Sigma_1^n |x_i|^2)^{1/2}$
The l_p norm: $\|x\|_p = (\Sigma_1^n |x_i|^p)^{1/p}$ The l_∞ norm: $\|x\|_\infty = max_i |x_i|$

With Multiplicative algorithms, the bounds depend on $\|u\|$, the separating hyperplane, and

$M = 2 \ln n \|u\|_1^2 max_i \left\|x^{(i)}\right\|_\infty^2 / min_i (u \cdot x^i)^2$.
This type of algorithm has an advantage when there are few relevant feature in concept.

With Additive algorithms, the bounds depend on $\|x\|$ (Kivinen / Warmuth, 1995), and $M = \|u\|^2 max_i \|x^i\|^2 / min_i (u \cdot x_i)^2$.
This type of algorithm has an advantage when there are few active features per example.

2.1.1 Algorithm Descriptions

The following is a summary of how weights are updated in the two families of algorithms:

Examples: $x \in \{0,1\}^n$; Hypothesis: $w \in R^n$
Prediction is 1 iff $w \cdot x \geq \Theta$

Additive weight update algorithm (Perceptron, Rosenblatt, 1958. Variations exit)
If Class = 1 but $w \cdot x \leq \Theta$, $w_i \leftarrow w_i + 1$ (if math $x_i = 1$) (promotion)
If Class = 0 but $w \cdot x \geq \Theta$, $w_i \leftarrow w_i / 2$ (if $x_i = 1$) (demotion)

Multiplicative weight update algorithm (Winnow, Littlestone, 1988. Variations exit)
If Class = 1 but $w \cdot x \leq \Theta$, $w_i \leftarrow 2 w_i$ (if $x_i = 1$) (promotion)
If Class = 0 but $w \cdot x \geq \Theta$, $w_i \leftarrow w_i / 2$ (if $x_i = 1$) (demotion)

2.2 How to compare Algorithms?

There are several areas in which algorithms can be compared. The first is Generalization. Since the representation is the same, we can compare how many examples are needed to get a given level of accuracy. The second is Efficiency. Determining the efficiency of an algorithm means asking how long it takes, per example, to learn a hypothesis and evaluate it. The third is Robustness. Robustness refers to the ability of the algorithm to adapt to new domains. The next sections will compare the Additive and Multiplicative Algorithms in a particular domain: that of context sensitive spelling correction.

2.2.1 Sentence Representation

Given a set of sentences, containing examples such as S below: S = "I don't know whether to laugh or cry."
Define a set of *features*. Features are relations that hold in the sentence. If we map a sentence to its feature-based representation, this representation will provide some of the information in the sentence and can be used as an example to your algorithm.
Conceptually, there are two steps in coming up with a feature-based representation.

What are the information sources available? Sensors: words, order of words, properties(?) of words

What features should we construct based on these sources? Why needed?

2.2.2 Domain Characteristics

In this domain, the number of potential features is very large. The instance space is sparse since we are only looking at particular pairs of words. This means that decisions will depend on a small set of features (sparse). In other words, we want to learn from a number of examples that is small relative to the dimensionality.

2.2.3 Generalization

The factors in generalization are dominated by the sparseness of the function space. Most features are irrelevant. The number of examples required by multiplicative algorithms depends mostly on the number of relevant features (generalization bounds depend on $\|w\|$).

A lesser issue in determining the generalization is the sparseness of features space. Here there is an advantage for additive approaches. Generalization depends on $\|x\|$ (Kivinen/Warmuth, 1995).

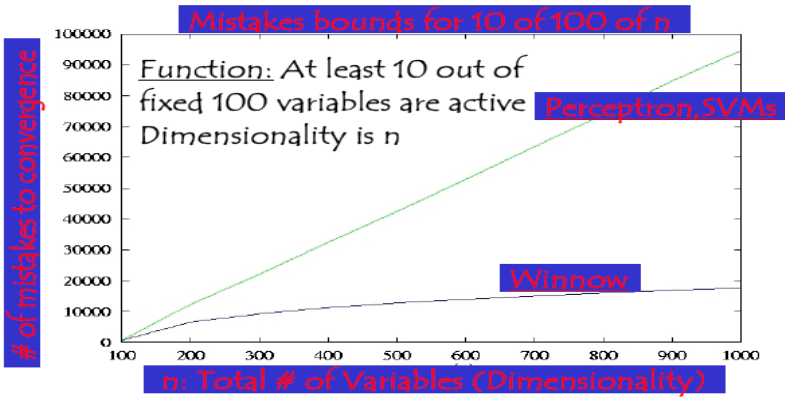

Figure 5: Mistake Bounds

2.2.4 Efficiency

Determining efficiency is dominated by the size of the feature space. Most features are functions (e.g. conjunctions) of raw attributes. For example:
$X(x_1, x_2, x_3, , \ldots x_k) \rightarrow X(\chi_1(x), \chi_2(x), \chi_3(x) \ldots \chi_n(x))$
Where $n >> k$. Additive algorithms allow the use of kernals, with no need to explicitly generate the complex features. The basic formula for a kernel is: $f(x) = \sum_i c_i K(x, x_i)$. This could be more efficient since work is done in the original feature space rather than in more dimensions. Kernels will be discussed in more detail in the next section.

2.3 Practical Issues and Extensions

There are many extension that can be made to these basic algorithms. Some of the extensions are necessary for them to achieve good performance and some are for ease of use and tuning. In the latter category, some extensions include converting the output of a Perceptron/Winnow algorithm to a conditional probability, adding Multiclass classification, and adding Real valued features. The key issues in efficiency involve cases in which there is an infinite attribute domain. The key generalization issues involve regularization. This will be motivated in the section in later notes on COLT.

One extension of Perceptron or Winnow to improve the generalization is introducing a *Thick Separator*. This involves adding some margin γ to the threshold and the weight update rules are re-defined as:

Promote if $w \cdot x > \theta + \gamma$.

Demote if $w \cdot x < \theta - \gamma$.

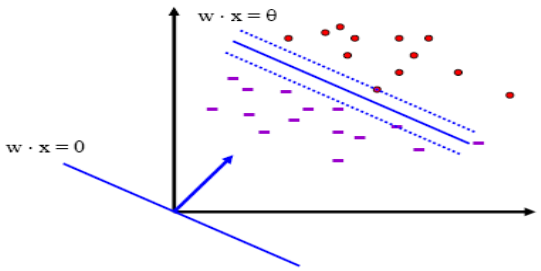

Figure 6: Thick Separator

Another extension of the Perceptron algorithm that helps with generalization is *Threshold Relative Updating*. In this case, the weights are changed relative to r as below:

$$w \leftarrow w + rxr = \frac{\theta - w \cdot x}{x \cdot x}$$

2.4 Regularization Via Averaged Perceptron

An Averaged Perceptron Algorithm is motivated by a number of considerations. Every Mistake-Bound Algorithm can be converted efficiently to a PAC algorithm in order to yield global guarantees on performance. In the mistake bound model, we do not know when we will make the mistakes. In the PAC model we want the dependence to be on the number of examples seen and not on the number of mistakes. In order to convert from one mistake-bound model to PAC model, it is necessary to do the following:

Wait for a long stretch without mistakes (there must be one).

Use the hypothesis at the end of this stretch; its PAC behavior is relative to the length of the stretch. Average Perceptron returns a weighted average of a number of earlier hypothesis; the weights are a function of the length of the mistake-free stretch.

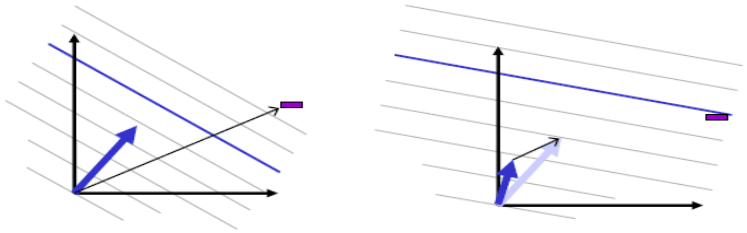

Figure 7: Threshold Relative Updating