CS 446 Machine Learning Fall 2016 Sep 13 - 22 2016

Online Learning

Professor: Dan Roth *Scribe: Ben Zhou, C. Cervantes*

Overview

- Quantifying Performance
- On-Line Learning
- Representation
- Perceptron
- Winnow
- Algorithms and Extensions

1 Quantifying Performance

We are interested in quantifying the number of examples one needs to see before we can say that a learned hypothesis is good.

1.1 Learning Conjunction

Assume the following hidden monotone conjunction[1]

$$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

We want to learn this function, and we want to know how many examples are needed to do so.

There are multiple possible learning protocols, such as the following.

Protocol I The learner proposes instances as queries to the teacher, who knows the hidden conjunction function.

Protocol II The teacher provides training examples. Since the teacher knows the hidden function f, it provides good training sets, that allow the learner to learn quickly.

[1]Monotone conjunctions, or conjunctions of non-negated variables, can be understood as functions $f(x)$ that do not decrease as x increases; in our case $f(x)$ goes from 0 to 1 and then stays 1

Protocol III Some random source (e.g. Nature) provides training examples; the teacher (Nature) provides the labels ($f(x)$)

1.2 Protocol I

Since we know that this is a monotone conjunction, if the label for one training example is positive, then the variables in the conjunction function must be positive in this example.

So if we only set one variable in the example to 0 and rest of the variables to 1, if the label is negative, it means that the variable set to 0 is necessary in the conjunction.

For example, we want to know if x_{100} is in then conjunction, we test the example $[1, 1, 1....1, 0]$. and $f(x) = 0$. Thus we know x_{100} is necessary, we must have it in the conjunction.

Then we will ask if x_{99} is in the conjunction. After test $f([1, 1, 1, 1...1, 0, 1]) = 1$, the conclusion is that we do not need it in the conjunction.

We can continue to drop variables, bit by bit. This straightforward algorithm requires n queries, and will produce as a result exactly the hidden conjunction $f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$.

1.3 Protocol II

When we consider a protocol between a teacher and a learner, we must assume that the teacher is going to give good examples but the teacher is not going to collude with the learner and simply provide the function.

If the teacher gives a positive example, we will know that none of the variables that have value 0 are in the conjunction.

For example, if we have an instance of $([0, 1, 1, 1, 1, 0, ...0, 1], 1)$, we know those 0 valued variables are not in the conjunction. In this way, we leaned a super set of the good variables (those have 1).

If the teaching examples are good enough that they contain all and only the good variables, it only requires 6 examples to produce this hidden function (one for each bit plus the super set). In this protocol, it takes minimal $k+1$ examples to learn where k is the number of variables in the hidden conjunction.

1.4 Protocol III

This protocol is most often studied in machine learning, partially because it's more natural, partially because it's easier to analyze.

An example set for this learning protocol could be:

Features	Label
$[1, 1, 1, 1, 1, 1, ..., 1, 1]$	1
$[1, 1, 1, 0, 0, 0, ..., 0, 0]$	0
$[1, 1, 1, 1, 1, 0, ..., 0, 1, 1]$	1
$[1, 0, 1, 1, 1, 0, ..., 0, 1, 1]$	0
$[1, 1, 1, 1, 1, 0, ..., 0, 0, 1]$	1
$[1, 0, 1, 0, 0, 0, ..., 0, 1, 1]$	0
$[1, 1, 1, 1, 1, 1, ..., 0, 1]$	1
$[0, 1, 0, 1, 0, 0, ..., 0, 1, 1]$	0

One algorithm we can use in this protocol is elimination.

Start with the set of all literals as candidates, if we see a positive example, and some of the literals are zeros in that example, we can conclude that those 0 literals are unimportant (given that our conjunction is monotone). These variables with value 0 can thus be eliminated.

The key difference between this protocol and the previous two protocols is that the input in this protocol in random (naturally generated). This means that it is not guaranteed to learn the hidden conjunction exactly. Given that our target function is a conjunction, however, we can still say something meaningful about the behavior of the output.

We still want to quantify the time it takes before we learn a satisfying function, but we cannot do so using the number of examples given that there are 2^{100} possible examples. Rather than focus on number of examples, we can continue in two directions:

Probabilistic Intuition
We consider the probability of one variable in the hidden conjunction never appearing in the example set. This probability is very small, so we can argue that the learned concepts perform well on future data that is distributed similarly to our training data. This approach is a key idea of the Probably Approximately Correct (PAC) framework.

Mistake Driven Learning
Rather than reasoning about the number of examples, we can say something about the number of mistakes our algorithm makes. Intuitively, if we make a mistake, we correct it, and the output performs better on future data. Now we think about how many mistakes we are going to make until we learn a good function, where we measure good in terms of how many mistakes you make before you stop and be happy with the hypothesis.

While we consider on-line mistake driven algorithms, not all on-line algorithms are mistake-driven. Some of them will update the hypothesis without a mistake.

2 On-Line Learning

2.1 Motivation

Consider a learning problem in a very high dimensional space. We want to develop an algorithm that depends only weakly on the space dimensionality and mostly on the number of relevant attributes. The key here is to extract the relevant rules to predict future data instead of memorizing all the examples.

2.2 Overview

Model
Instance space: X (dimensionality: n)
Target: $f : X \to \{0, 1\}, f \in C$

Where C is the concept class – a collection of all target functions – parameterized by n

Protocol
Learner is given $x \in X$
Learner predicts $h(x)$, and is then given $f(x)$ (feedback)

Performance
Learner makes a mistake when $h(x) \neq f(x)$

We want to measure the number of mistakes algorithm A makes on sequence S of examples, for the target function f

$$M_A(C) = max_{f \in C,S} M_A(f, S)$$

We say A is a mistake bound algorithm for the concept class C if $M_A(C)$ is a polynomial in n, the complexity parameter of the target concept. We say that A is good if it has this property.

2.3 Mistake Bound Learning

We want to know how many mistakes to get to $\epsilon - \delta(PAC)$ behavior; that is, how many mistakes before we can say there is a high possibility that the error rate is less than a small number. We can also look for exact learning: how many mistakes are made before the function stops making mistakes. This is easier to analyze.

This view has the notable drawback of being too simple, as it's not clear when mistakes will be made, but the simplicity can also be advantageous.

2.4 Generic Mistake Bound Algorithms

Let's first think about if it is clear that we can bound the number of mistakes. Ideally if we make a mistake, we update the hypothesis so that it will not make the same mistake again. However, most learning algorithms will not have this property, unless you do something special.

In the general case, let C be a finite concept class and we want to learn $f \in C$. Consider the following algorithm: consistency (CON).

In the i^{th} stage of the algorithm, let C_i be all the concepts in C that are consistent with all $i-1$ previously seen examples. We choose randomly $f \in C_i$ and use it to predict the next example.

It is clear that $C_{i+1} \subseteq C_i$, so that if a mistake is made on the i^{th} example, then $|C_{i+1}| < |C_i|$. In this way, each time we make a mistake we remove a hypothesis and thus make progress toward learning the correct function.

As defined above, CON makes at most $|C| - 1$ mistakes.

2.5 The Halving Algorithm

Let C be a concept class. We want to learn $f \in C$

Halving:

In the i^{th} stage of the algorithm, Let C_i be all concepts in C that is consistent with all $i-1$ previously seen examples. Given an example e_i, consider the value $f_j(e_i)$ for all $f_j \in C_i$ and predict by majority.

In this case if the hypothesis still makes a mistake, that means the majority is wrong and we can get rid of most of the concepts.

For example in this algorithm, we predict 1 if $|\{f_j \in C_i; f_j(e_i) = 0\}| < |\{f_j \in C_i; f_j(e_i) = 1\}|$

Thus, since $C_{i+1} \subseteq C_i$, when a mistake is made we can eliminate at least half of the concepts, $|C_{i+1}| < \frac{1}{2}|C_i|$

In this case, the halving algorithm makes at most $\log(|C|)$ mistakes.

Conjunction Example
Assume a hidden conjunction:

$$f = x_2 \wedge x_3 \wedge x_4 \wedge x_5 \wedge x_{100}$$

The number of all conjunction is 3^n (since each variable can be positive, negative, or absent in the conjunction). As defined above, the halving algorithm makes at most $\log(|C|) = \log(3^n) = n$ mistakes when learning the target function.

Now let's consider k-conjunctions. Assume that $k << n$ where n is the number of attributes and that could be a large number. The total number of k-conjunctions is $2^k C(n,k) \approx 2^k n^k$. In this case, $\log(|C|) = k \log n$.

3 Representation

Representation is a key issue in learning. Assume, for example, that we are interested in learning conjunctions. Should our hypothesis space be the set of conjunctions?

We want a hypothesis space that is large enough to contain the target function while being small enough to efficiently find a valid hypothesis.

Consider the following theorem:

> Given a sample on n attributes that is consistent with a conjunctive concept[2], it is NP-hard to find a pure conjunctive hypothesis that is both consistent with the sample and has the minimum number of attributes.

In effect, if we want to learn conjunctions, we may not want to use the set of conjunctions as our hypothesis space. We could therefore move to a larger hypothesis space – like the set of linear functions – where finding a valid hypothesis may be combinatorially easier.

3.1 Linear Functions

Consider a function $w \cdot x = 0$, as the bottom line of Figure 1. In this space,

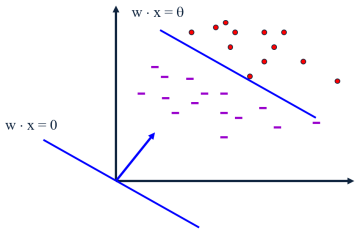

Figure 1: Linear function

we're interested in learning the linear separator, which shifts the line on the bottom to that on the top, representing $w \cdot x = \theta$.

[2]This theorem holds for disjunctions as well

4 Perceptron

Almost all the machine learning algorithm we learn today is a variation of perceptron. The perceptron algorithm is **online**[3] and **mistake driven**[4], meaning that – given an example – perceptron tries to predict the label using the current hypothesis. If the prediction is correct, then the algorithm does nothing; otherwise, it corrects the hypothesis.

4.1 Linear Threshold Units

In the literature, perceptrons and **linear threshold units** are often conflated. A linear threshold unit, as shown in Figure 2, takes inputs X, assigns weights W, and applies a threshold to produce label, y. In this class we'll treat perceptron

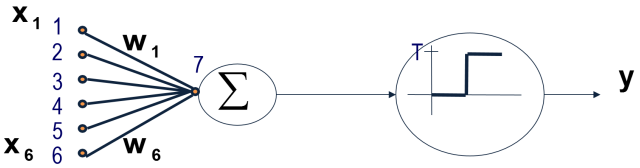

Figure 2: Perceptron as a Linear Threshold Unit

as one algorithm among many that learns this architecture.

4.2 Algorithm

We want to learn a mapping $f : X \to \{-1, +1\}$ represented as

$f = sgn(w \cdot x)$

Where $X = \{0, 1\}^n$ (R^n) and $w \in R^n$

Initialize $w = 0 \in R^n$
Cycle through all examples:
    Predict the label of instance x to be $y' = sgn(w \cdot x)$
  If $y' \neq y$, update the weight vector:
    $w = w + ryx$ (r is the learning rate)
  Else: Leave weights unchanged

The intuition in the above updating rule is that if y is positive and we predicted it negative, then we are adding value to the weight vector. Similarly if y is negative, we are subtracting value from the weight vector.

[3] Online algorithms operate over examples one-at-a-time, rather than all-at-once
[4] Mistake driven algorithms update their hypothesis when they make a mistake

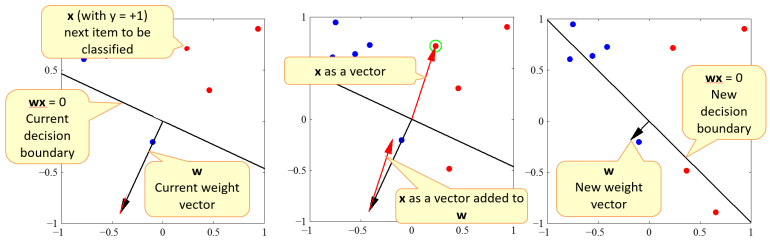

Figure 3: Updating rule intuition

Figure 3 gives some intuition about the updating rule. Here red points are positive examples and blue are negative examples. In the leftmost subfigure, the arrow represents w and any point on the side that the arrow points to will be classified as positive (negative otherwise).

In the leftmost subfigure, the next example to be classified (denoted by the note bubble) is red and will be misclassified by the linear separator. Thus, an update – as shown in the middle subfigure – is performed, resulting in the boundary in the rightmost subfigure. In effect, each update rotates w in our space until we correctly classify the examples.

If we know that if x is Boolean, it is only the weights of active features that are updated. This is important because it brings efficiency when dimensionality is really high; you thus do not need to update an array representing weights for all features, only active features.

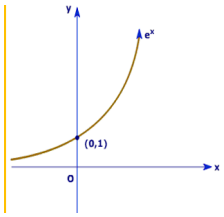

Figure 4: An equivalent form

An equivalent from of $w \cdot x > 0$ is $\frac{1}{1+exp\{-(w \cdot x)\}} > \frac{1}{2}$.

Remember also that at each iteration θ must also be updated, according to

$$\theta = \theta + ry\theta_{Init}$$

In cases where θ is not folded into the weight vector update, this is important to remember.

4.3 Learnability

Perceptron can only learn linearly separable functions. Minsky and Papert (1969) wrote an influential book demonstrating perceptron's representational limitations (ie. parity functions like XOR can't be learned; in vision – if patterns are represented with local features – perceptron can't represent symmetry and connectivity).

In 1959, Rosenblatt himself asked

> "What pattern recognition[5] problems can be transformed so as to become linearlly separable?"

Rosenblatt had the perspective that if we have a very complex function, and the process for learning it is unknown, we can transform the feature space such that the data becomes linearlly separable, as shown in Figure 5

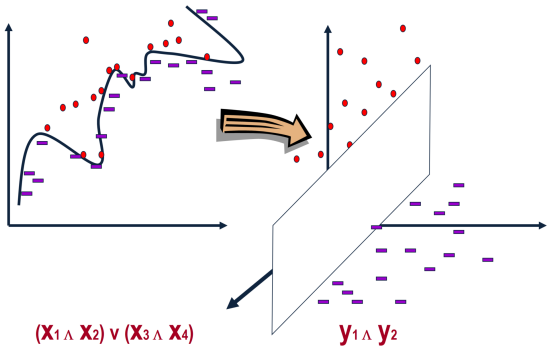

Figure 5: Transformed feature space

There are two important considerations when thinking about perceptron learnability, as given by the following theorems.

Perceptron Convergence Theorem

> If there exist a set of weights that are consistent with the data (ie. the data is linearly separable), the perceptron learning algorithm will converge

Perceptron Cycling Theorem

> If the training data is not linearly separable, the perceptron learning algorithm will eventually repeat the same set of weights and therefore enter an infinite loop.

[5]Machine learning used to be called pattern recognition

4.4 Mistake Bound Theorem

When using perceptron, we can bound the number of mistakes in the infinite case.

Assume that a weight vector $w \in \mathbb{R}^N, w_0 = (0, ...0)$.. Upon receiving an example $x \in \mathbb{R}^N$, we predict according to a linear threshold function.

Let $(x_1; y_1), ...(x_t; y_t)$ be a sequence of labeled examples with $x_i \in \mathbb{R}^N, ||x_i|| \leq R$ and $y \in \{-1, 1\}$ for all i.

Let $u \in \mathbb{R}^N, \gamma > 0$ be such that $||u|| = 1$ and $y_i u \cdot x_i \geq \gamma$ for all i. Perceptron makes at most $\frac{R^2}{\gamma^2}$ mistakes on this example sequence[6].

This theorem assumes that all examples are bounded by some R; for all x_i, find the largest one, and R is at least this size. The theorem further assumes that there exists some u that separates the data (that is, the data is linearlly separable). Requiring that $||u|| = 1$[7] is simply a constant that could be arbitrarily scaled.

Finally, the theorem assumes that there exists some γ such that the inequality is satisfied. This is simply a compact way to say positive examples are greater than γ and negative examples are less than $-\gamma$. If the data is linearlly separable, the closest point to the hyperplane is γ, and thus this assumption always holds for linearaly separable data

We refer to γ as the *complexity parameter*, which refers to how difficult the learning problem is. If γ is small, positive and negative examples are very close and it is difficult to define a hyperplane. If γ is very large,finding a hyperplane is much easier. Thus, it makes sense to measure the difficulty (i.e. the number of mistakes) using the complexity parameter.

Proof
Let v_k be the be the hypothesis before the k^{th} mistake. Assume that the k^{th} mistake occurs on the input example (x_i, y_i).

$$\begin{aligned} \therefore y_i(\mathbf{v}_k \cdot \mathbf{x}_i) &\leq 0 \\ \mathbf{v}_{k+1} &= \mathbf{v}_k + y_i \mathbf{x}_i \\ \mathbf{v}_{k+1} \cdot \mathbf{u} &= \mathbf{v}_k \cdot \mathbf{u} + y_i(\mathbf{u} \cdot \mathbf{x}_i) \\ &\geq \mathbf{v}_k \cdot \mathbf{u} + \gamma \\ \therefore \mathbf{v}_{k+1} \cdot \mathbf{u} &\geq k\gamma \end{aligned} \tag{1}$$

$$\begin{aligned} ||\mathbf{v}_{k+1}||^2 &= ||\mathbf{v}_k||^2 + 2y_i(\mathbf{v}_k \cdot \mathbf{x}_i) + ||\mathbf{x}_i||^2 \\ &\leq ||\mathbf{v}_k||^2 + R^2 \\ \therefore ||\mathbf{v}_{k+1}||^2 &\leq kR^2 \end{aligned} \tag{2}$$

[6]The perceptron mistake bound algorithm by Novikoff (1963)

[7]Remember that $||u||$ is the L2 norm of the vector, given by $\sqrt{u_0^2 + ... + u_n^2}$

Therefore,

$$\sqrt{k}R \geq ||\mathbf{v}_{k+1}|| \geq \mathbf{v}_{k+1} \cdot \mathbf{u} \geq k\gamma$$
$$\therefore k < \frac{R^2}{\gamma^2}$$

This holds because $||\mathbf{u}|| \leq 1$

Note that the bound does not depend on the dimensionality nor on the number of examples. It's also important to note that weight vectors and examples are in the same $\mathbb{R}^n$ space.

4.5 Robustness to Noise

One important relaxation to the perceptron algorithm allows us to be more robust to noise.

Consider the case of non-linearly separable data, as in Figure 6. Here, we do not

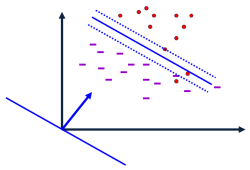

Figure 6: Non-linearly separable case

have a margin γ. In order to replace γ, we can use a **slack variable**, defined as

$$\xi_i = max(0, \gamma - y_i w \cdot x_i)$$

Intuitively, when $\xi_i = 0$, the example x_i is on the right side of the hyperplane with at least γ distance to the plane, otherwise, it grows linearly with $-y_i w \cdot x_i$

Let's now denote

$$D_2 = [\sum \{\xi_i^2\}]^{\frac{1}{2}}$$

Now we can refine the Novikoff theorem from before such that perceptron is guaranteed to make no more than $(\frac{R+D_2}{\gamma})^2$ mistakes on any sequence of examples satisfying $||x_i||^2 < R$

Here D_2 can represent how far the data set is from linearly separable, so it is reasonable to use it to measure how many mistakes we are going to make.

5 Winnow

Winnow is another linear, mistake-driven learning algorithm which is similar to perceptron. Note that in the following definitions we consider Winnow with a multiplicative term of 2, but in practice any $\alpha > 1$ can be used (typically close to 1).

5.1 Algorithm

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 $w \cdot x < \theta$, $w_i = 2w_i$ (if $x_i = 1$) (promotion)

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

In principle this is similar to perceptron – increase the weights on positive mistakes, decrease on negative mistakes – but winnow does this multiplicatively.

Additionally, when learning disjunctions winnow can use elimination rather than demotion; instead of dividing w_i by 2, it can just be set to 0.

5.2 Mistake Bound

Claim: Winnow makes $O(k \log n)$ mistakes on k-disjunctions.

Proof

Let u be the number of mistakes on positive examples (promotions) and v be the number of mistakes on negative examples (demotions).

The mistakes on positive examples is bounded as in

$$u < k \log(2n)$$

When learning k-disjunctions, u is bounded because 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.

The mistakes on negative examples, however, is slightly more subtle, given by

$$v < 2(u + 1)$$

Consider the total weight, TW, which is equal to n initially. If the learner makes a mistake on a positive example, TW doubles. However, $w \cdot x < \theta = n$, meaning

that the current TW cannot be larger then n. So in the worst case TW will grow by n, giving us

$$TW(t+1) < TW(t) + n$$

During demotion, $w \cdot x \geq \theta = n$, so when the learner makes a mistake on a negative example we know that the total weight must be greater than n, so dividing TW by 2 means that the new total weight will be at least $\frac{n}{2}$ smaller, or

$$TW(t+1) < TW(t) - \frac{n}{2}$$

These two bounds are sufficient to prove the bound on the total number of mistakes.

Given u and v as defined above, we know that the total weight is always positive, starts at size n, increases with n and the number of mistakes on positive examples, and decreases with $\frac{n}{2}$ and the number of negative examples.

$$0 < TW < n + un - \frac{vn}{2} \Rightarrow v < 2(u+1)$$

Combining all of these equations, we get that

$$u + v < 3u + 2 = O(k \log n)$$

6 Algorithms and Extensions

As we've mentioned before, perceptron and winnow are very similar.

Examples $x \in \{0,1\}^n$ or $x \in \mathbb{R}^n$ (indexed by k);

Hypothesis $w \in \mathbb{R}^n$

Prediction $y \in \{-1,+1\}$: Predict: $y = 1$ iff $w \cdot x > \theta$

Update Mistake Driven; the learner learns by correcting mistakes

Under mistake driven algorithms, we have seen two kinds of update algorithm, which is the main distinction between perceptron and winnow.

Perceptron is an additive weight update algorithm, where $w \leftarrow w + r y_k x_k$

Winnow is a multiplicative weight update algorithm, where $w_i \leftarrow w_i exp\{y_k x_i\}$

6.1 Practical Issues and Extensions

There are many extensions that can be made to these basic algorithms. Some are necessary for them to perform well (eg. regularization), and some are for ease of use and tuning.

For example, we can convert the output of a Perceptron/Winnow to a conditional probability $P(y = +1|x) = [1 + exp(-Awx)]^{-1}$

There are some other issues that we will talk about later, like multiclass classification and infinite attribute domain.

6.2 Regularization Via Averaged Perceptron

In mistake driven algorithms, we generate a new hypothesis every time we make a mistake. It is reasonable to choose the last hypothesis as the learned output. However, consider the case where a mistake was made on the penultimate example. That hypothesis survived all prior examples before making a mistake, and is thus more reliable than the final hypothesis, which has only seen one example.

This intuition doesn't come naturally in mistake-bound algorithms, but it is what drives the PAC model that we'll discuss later. In the PAC model, the output depends on the number examples rather than the number of mistakes, which can yield global guarantees on performance.

Consider a mistake bound algroithm...
that makes no more than 500 mistakes. Assume we want to see – given an infinite stream of examples – a set of 1000 examples on which the algorithm makes no mistakes.

In the optimal case, we start running the algorithm and we make no mistakes on the first thousand examples. Our goal is satisfied in 1000 examples.

In the worst case, we start running the algorithm and make no mistakes on 999 examples, only to make our first mistake on the thousandth. This behavior (999 correct, 1 mistake) can repeat, but only 500 times, given the bound of the algorithm, thus satisfying our goal in ∼500,000 examples.

In general, then, we show that simple-to-analyze mistake-bound algorithms give us meaningful information about the number of examples we need before we stop making mistakes on long stretches of examples.

Averaged Perceptron...
builds on this intuition by taking all learned hypothesis – each of which has learned something about the target function – and weigh them according to the number of examples on which they made no mistakes (the length of the stretch). If a hypothesis survives a long stretch, it gets a larger weight than one that survived a short stretch.

Let m be the number of examples, k the number of mistakes c_i be the consistency count for hypothesis v_i; on how many examples did v_i make no mistakes.

Given a labeled traning set $\{(x_1, y_1), ...(x_m, y_m)\}$, we want to produce a list of weighted peceptrons $\{(v_1, c_1), ...(v_k, c_k)\}$

Initialize $k = 0; v_1 = 0, c_1 = 0$
Repeat T times
  For $i = 1...m$
    $y\prime = sign(v_k \cdot x_i)$
    If $y\prime = y$
      $c_k = c_k + 1$
    else
      $v_{k+1} = v_k + y_i x$
      $c_{k+1} = 1$
      $k = k + 1$

Having produced this collection of weighted perceptrons – $\{(v_1, c_1), ...(v_k, c_k)\}$ – we can now predict the label of new example x by

$$y(x) = sgn[\sum_{i=1}^{k} c_i sgn(v_i \cdot x)]$$

6.3 Perceptron with Margin

Perceptron with Margin is also known as Thick Separator, and the method described below also applied to Winnow.

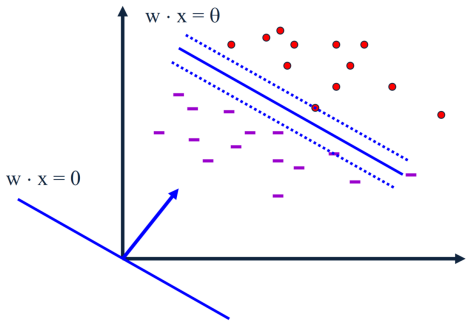

Figure 7: Perceptron with Margin

Figure 7 shows the case where there is some margin between positive and negative examples. Though we could choose any hyperplane within the range of θ, we want to choose the one that is as much in the middle as possible. Doing so results in higher tolerance to noise on new data and thus better test set performance.

To guarantee that we choose a hyperplane in the middle, we enforce some margin γ such that we

promote if $wx - \theta < \gamma$

demote if $wx - \theta > \gamma$

Note that γ here is a functional margin. Its effect could disappear as w grows. Nevertheless, this has been shown to be a very effective algorithmic addition.

6.4 Aggressive Perceptron

Assume we've run perceptron and we've made a mistake on an example, as in the left side of Figure 8. Consider that, at some point in the future, we see this example again. We are not guaranteed to classify this example correctly because our step size may be too small to have changed the weight vector enough.

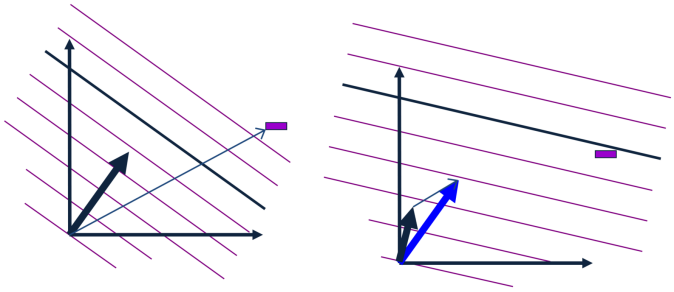

Figure 8: Threshold relative updating

In order to avoid making a mistake on the same example, then, we could either feed an example enough times such that the weight vector is increased enough, or we can change the step size on an example on which we made a mistake, according to

$$r = \frac{\theta - w \cdot x}{x \cdot x}$$

This behaves equivalently, and prevents perceptron from making the same mistake twice.

6.5 SnoW

Several of these extensions (and a couple more) are implemented in the SNoW learning architecture found in LBJava that supports several linear update rules (Winnow, Perceptron, Nave Bayes).

Download from http://cogcomp.cs.illinois.edu/page/software

6.6 Winnow Extensions

In Winnow, all the weights are positive, meaning that only monotone functions are learned.

Duplication
To make Winnow more general, we can duplicate variables and treat these duplicates as negations; the first n variables could be x_i, the next n variables could be $\neg x_i$. Winnow then simply learns a monotone function over $2n$ variables.

Using the duplication approach is theoretically correct, but there are practical concerns. First, there are memory concerns – we double the variables – but there is a larger concern.

Assume only k features are on in any given example. In the original case, Winnow is only concerned with k weights: those for positive features. In this duplicated setting, though, Winnow must retain k weights for positive features and $n-k$ weights for negative features, such that for each example (now of size $2n$), Winnow must carry n weights, making it significantly less efficient in practice.

Balanced Winnow
Assume we keep two weights for each variable, where the effective weight is the difference between these weights.

In this setting, we have two weight vectors ($\mathbf{w}^+$ and $\mathbf{w}^-$), and if we made a mistake on a positive example we promote w_i^+ and demote w_i^-, and the reverse if we make a mistake on a negative example, given by.

Where $x_i = 1$
If $f(x) = 1$ but $(\mathbf{w}^+ - \mathbf{w}^-) \cdot x \leq 0$
  $w_i^+ \leftarrow 2w_i^+$
  $w_i^- \leftarrow \frac{1}{2} w_i^-$
If $f(x) = 0$ but $(\mathbf{w}^+ - \mathbf{w}^-) \cdot x \geq 0$
  $w_i^+ \leftarrow \frac{1}{2} w_i^+$
  $w_i^- \leftarrow 2w_i^-$

Note that these weight vectors are not independent. When we update one we update the other. Later we will discuss why this is the right way to do multiclass classification.

6.7 Winnow Robustness

Both winnow and perceptron are robust in the presence of various kinds of noise. To demonstrate this, imagine a noisy setting in which the target function changes over time. This is similar to many real life settings in which we learn under some distribution but test under a slightly different one.

Modeling
Consider a game in which we have an adversary and a learner taking turns, where

> The adversary may change the target disjunction by adding or removing some variable; the cost of each addition is 1
>
> The learner makes a prediction on the given examples, and is told the correct answer according to the current target function

Consider a new Winnow – Winnow-R – that never lets the weights go below $\frac{1}{2}$; doing so enables the learner to recover if a previously unimportant variable becomes important (since, unlike in standard Winnow, the weight is nonzero).

Winnow-R makes $O(c \log n)$ mistakes, where c is the cost of adversary.

As in Section 5.2, we define u to be the mistakes on positive examples, v to be the mistakes on negative examples, and the total weight of positive mistakes is given by $TW(t+1) < TW(t) + n$.

In the case of negative mistakes, however, we bound the weight at $\frac{1}{2}$, meaning that the mistakes on negative examples is given by $TW(t+1) < TW(t) - \frac{n}{4}$

Then similarly, we have $0 < TW < n + un - \frac{vn}{4} \Rightarrow v < 4(u+1)$