These notes outline the discussion we will have on discriminative approaches to classification.

# 1 Outline

- Most approaches in NLP make decisions based on linear functions over their feature space.
- The key difference between algorithms is how they determine the weight of the coefficients. This is related to the *loss function* optimized by different algorithms.
- Different algorithms can be compared based on *generalization* and *efficiency* (both in training and evaluation). Properties of the instance space (features and examples) and the function space can be very significant in affecting generalization.
- Algorithms: Winnow, Perceptron, SVM, Adaboost.
- Feature space: Kernels vs. blowing up the feature space. What's better and when?
- Multiclass classification: 1-vs-all; pairwise classification; constraint classification.
- Semi-supervised learning

# 2 Introduction

We have seen that the decision surface of the naive Bayes classifier and several other classifiers is a linear function in its feature space.

In most cases, we are interested approximating a function $y = f(x)$ from a collection of examples $(x, y)$. In binary classification problems we are interested in predicting $y \in \{-1, +1\}$ which is typically done by predicting

$$y = sgn(f(x)).$$

The classification error of the function $f$ is given by:

$$Err(f(x, y) = I_{\{yf(x)<0\}},$$

where $I_C$ denote the characteristic function of the set $C$, taking value 1 in $C$ and 0 outside it.

We are interested in finding a function $f$ that *minimizes* the empirical error

$$Err_{emp} = \frac{1}{|S|} \sum_{x \in S} I_{\{yf(x)<0\}}.$$

This can be viewed as an approximation of the true error that we want to minimize.

However, in most cases, and specifically in the case of linear functions, minimizing this cost function is NP-hard. Most algorithms therefore attempt instead to optimize a different cost function, which is an upper bound of the true error function. Different algorithms optimize different cost functions, and provide different justifications for why this is a reasonable cost functions.

There are many cost functions that have been studied. Examples include:

- Max Entropy: optimizes the logistic regression cost function.

$$\sum_{\{x \in S\}} ln(1 + exp\{-f(x)y\})$$

- Adaboost optimizes an exponential cost function:

$$\sum_{\{x \in S\}} exp\{-f(x)y\}$$

- SVM optimizes with respect to

$$\sum_{\{x \in S\}} max\{1 - f(x)y, 0\}$$

- Least Mean Square optimizes with respect to:

$$\sum_{\{x \in S\}} (y - f(x))^2$$

Justification vary from margin based analysis (in the case of Boosting and SVM) to relation to maximum-likelihood (in the case of maximum entropy) and more direct methods.

In this lecture we will present two important families of algorithms the learn linear functions - additive update algorithms and multiplicative update algorithms - analyze their properties and study their suitability to NLP tasks.

# 3 Linear Functions

We study functions $f : X \to Y$ were $Y = \{0, 1\}$ (or sometimes, for convenience, we will use $Y = \{-1, +1\}$ ). The domain $X$ can be in most cases either $\{0, 1\}^n$ or $R^n$. The algorithms we present can apply in both cases.

We are concerned with linear functions, that is, functions that can be represented as a weight vector $w \in R^n$. Given $w \in R^n, x \in X$, we predict:

$$y = 1 \quad \text{iff} \quad w \cdot x \geq \theta$$

and otherwise we predict $y = 0$

W.l.o.g we can always assume $\theta = 0$ by appending a constant feature to the input vector $x$ to offset the effect of $\theta$. In this new dimensional space we can now discuss only hyperplanes that go through the origin.

We say the a sample $S = \{(x, y)\}_1^m$ is *linearly separable* if there exist a linear function $f$ that is consistent with it.

### Many functions are linear:

- $k-$disjunctions
- $k-$conjunctions
- Threshold functions: $Th_{k,m}$, ($k$ out of $m$).

Which means that when we see a sample taken according to a function of this sort we know that it is linearly separable.

Notice: some algorithms that learn linear representations may not be able to represent *all* linear functions.

In Problem set 3 you will show that a hypothesis generated using the naive Bayes algorithm cannot represent the function $Th_{3,7}$ (sample takes over the uniform distribution and labeled according to this function).

### Not all functions are linearly separable:

The function

$$f(a, b) = a \oplus b = a\bar{b} \vee b\bar{a}$$

is not linearly separable over the domain $\{a, b\}$. Function that belong to

- $k-$DNF
- $k-$DL

are not linearly separable over their original space; but, there is a simple and efficient (polynomial) transformation that makes them linear.

## 4 On Line Learning

We will study the algorithms for learning linear functions in the on-line learning model. This is a natural model of learning in which we assume that a change to the hypothesis is made only given the last example observed. (This is non-trivial to formalize in a rigorous way). See comments later.

**Learning protocol:** Assume there is a hidden function $f : X \to Y$. (Same can be done if we are learning agnostically and simply see examples over $(X, Y)$ without the notion of a hidden function.)

For $i = 1, 2, \ldots$

1. Learner is given $x^{(i)} \in X$
2. Learner predicts $h(x^{(i)}) \in Y$
3. Learning is told $y^{(i)} \in X$

**Performance:** The learner makes a mistake when $h(x^{(i)}) \neq y^{(i)}$.

Define $\mathcal{M}_A(c, S)$ to be the number of mistakes the algorithm A makes on the sequence $S$ of examples when the target concept is $c$.

$$\mathcal{M}_A(c) = max_s \mathcal{M}_A(c, S)$$

is the number of mistakes $A$ makes on the worst sequence of examples.

$$\mathcal{M}_A(C) = max_{c \in Cs} \mathcal{M}_A(C)$$

is the worst case number of mistakes $A$ makes over all possible concepts and sequences.

**Definition:** We say that $A$ is a *mistake bound learning algorithm* if $\mathcal{M}_A(C)$ is polynomial in $n$, where $n$ is the complexity parameter of the target concept.

# 5 Comments

1. We could ask: how many mistakes the learner makes before reaching an $\epsilon$ - $\delta$ PAC behavior. Instead we are talking about the number of mistakes before learning the concept exactly.

2. Before, in the PAC models, we had a notion of distribution; we did not have to know it, but we were trained and tested on the same distribution. Now, we are tested on the worst case.

3. In the PAC model our performance was local – what is the probability of correct prediction in a randomly drawn example (although we may take many examples to evaluate this probability). Now, it is global – performance along time. Actually, even if we say that the algorithm makes 17 mistakes, in general, we do not know *when* we are going to make these mistakes.

4. Main drawback: too simple; does not seem to model many important learning situations. Also – we have no way to tell whether we have made all the mistakes. We do not know where we are. But, there are some relations between mistake bound guarantees and pac guarantees. In fact, every mistake bound algorithm can be shown to be pac (in the sample complexity sense).

5. Advantages: Very simple; many issues arise even in this setting. There is a generic way to transform results here to results in other models. Although the definition seem to be weak, it is more strong than we think. There is no *natural* learning problem for which we know how to prove that the concept is PAC learnable but not mistake bound learnable.

6. Memory: Sometimes people like to think of on-line learning as a model in which we do not remember the examples. We see an example, react, perhaps update our internal representation, but get rid of the example, and go one. It turns out that this is not so easy to define formally. With the current definition, it is possible for the algorithm to remember all the previous examples, and at every point run a batch learning algorithm.

   But, informally, you can think of an on-line algorithm as an algorithm that makes an update that depends on a single example, the last one observed.

Moreover, some of our algorithms will even be *conservative* algorithms, in the sense that they will make an update only when they made a mistake. These are also called *mistake driven algorithms.*

7. Although we have defined mistake bound algorithms as those that make a polynomial number of mistakes (where $n$, say, is the dimensionality of the instance space) in general, it is non trivial that algorithms even have a *finite* mistake bound. If the hypothesis space is finite, than clearly there are "generic" algorithms with a finite mistake bound (but they might not be efficient).

# 6 Perceptron

We will assume $X = \mathcal{R}^n, Y = \{-1, +1\}$. The input is a sample $S = \{(x^1, y^1), \ldots (x^m, y^m)\}$ that is treated in an on-line way. Typically, the algorithm is run repeatedly on this batch of data until it finds the weight vector that is correct on all the training data.

We use $||x||_2$ to denote the Euclidean length of $x$, $||x||_2 = \sqrt{\sum_1^n x_i^2}$ (see appendix)

The algorithm was first studied by Rosenblatt [**?**].

1. Initialize $w = 0 \in R^n$
2. Predict the label of a new instance $x$ to be $\hat{y} = sgn(w \cdot x)$.
3. If $\hat{y} \neq y$ update the weight vector to
$$w = w + rxy,$$
where $r$ is some constant, called the *learning rate.* We will assume from now on that $r = 1$. (Notice that in the case $X = \{0, 1\}^n$, only coefficients of *active* features are updated.)
4. If the prediction is correct then $w$ is not updated.

It was shown in the early sixties that if the data is linearly separable, then the perceptron algorithm will make a finite number of mistakes. Therefore, if it is used to repeatedly cycle through the training set, it will converge to a vector which correctly classifies all the examples. Moreover, the number of mistakes is upper bounded by a function of the minimal gap between true separator and the examples, also known as the *margin.*

**Before we go on to prove it, notice the implications on some of your assignments, in cases where you got 100% with one of the algorithms.**

**Theorem 6.1 (Block, Novikoff)** *Let $S = \{(x^1, y^1), \ldots (x^m, y^m)\}$ be a sequence of labeled examples with $||x^i|| \leq R$. Suppose that there exists a vector $u$ such that $||u|| = 1$ and*

$$y^i(u \cdot x^i) \geq \gamma$$

*for all examples in the sequence. Then the number of mistakes made by the online perceptron algorithm on this sequence is at most $R/\gamma$.*

**Proof:** The intuition of the proof is very simple. The idea is that the weight vector you maintain, $w$, should come pretty close to $u$. (It does not have to be identical to $u$, though).

We can, for example, consider the distance $||w-u||^2$ which is non negative, and show that it decreases with every mistake the algorithm makes by some amount. This will bound the number of mistakes that the algorithm can make. Actually, we will not consider exactly this distance but, for technical reasons, a slightly different *potential function*:

$$W_t = ||w_t - \frac{R^2}{\gamma}u||_2^2.$$

Let $w_k$ be the weight vector used prior to the $k$th mistake ($w_1 = 0$). We will look at

$$W_{k+1} - W_k.$$

Recall that if the $k$th mistake occurs on $(x^i, y^i)$ then

$$y^i(w_k \cdot x^i) \leq 0$$

and that

$$w_{k+1} = w_k + y^i x^i.$$

Therefore

$$\begin{aligned}
W_{k+1} &= ||w_{k+1} - \frac{R^2}{\gamma}u||_2^2 = ||(w_k - \frac{R^2}{\gamma}u) + y^i x^i||_2^2 && (1)\\
&= W_k + 2y^i x^i(w_k - \frac{R^2}{\gamma}u) + ||x^i||_2^2 = W_k + 2y^i x^i w_k - 2y^i x^i \frac{R^2}{\gamma}u + ||x^i||_2^2 && (2)\\
&\leq W_k - 2\gamma \cdot \frac{R^2}{\gamma} + R^2 && (3)\\
&= W_k - R^2 && (4)
\end{aligned}$$

Where the inequality is due to (1) $y^i(w_k \cdot x^i) \leq 0$ (second term is replace by 0), (2) $y^i(u \cdot x^i) \geq \gamma$ and, $||x||_2^2 < R^2$.

Now, since $W_t > 0$ for all $t$, its initial value is $W_1 = R^4/\gamma^2$, and it goes down by *at least* $R^2$ every time the algorithm makes a mistake, then the number of mistakes it can make is at most $R^2/\gamma^2$ mistakes before it gets to 0.

■

Therefore, the general bound we get (not restricting now $||u||_2 = 1$) is: The perceptron algorithm learns a weight vector that correctly classifies all training data after at most $M$ updates, where,

$$M = \frac{||u||_2^2 \max_i ||x^i||_2^2}{\min_i (u \cdot x^i)^2}.$$

# 7 Winnow

Similar to Perceptron, the Winnow [**?**] algorithm is also on-line and mistake driven. There are several versions of the algorithm and we will discuss here the unnormalized positive Winnow version and mention transformations to the general case later.

We will assume $X = \mathcal{R}^n, Y = \{-1, +1\}$. The input is a sample $S = \{(x^1, y^1), \ldots (x^m, y^m)\}$, that is treated in an on-line way.

1. Initialize $w = \mu \in R^n$
2. Predict the label of a new instance $x$ to be $\hat{y} = sgn(w \cdot x)$.
3. If $\hat{y} \neq y$ update the weight vector to
$$w = w \cdot \exp rxy,$$
where $r$ is some constant, called the *learning rate*. We will assume from now on that $r = 1$. (Notice that in the case $X = \{0, 1\}^n$, only coefficients of *active* features are updated.)
4. If the prediction is correct then $w$ is not updated.

Notice that, as in the case of perceptron, this is a mistake driven algorithm, that is, we update the weights only when a mistake is made. Also, when the instance space is Boolean, only weights of *active* features are updated, and they can either be multiplied by a constant or divided by a constant.

Winnow has been generalize in several ways; like in the perceptron case, one can use it as a *regressor* rather than a *classifier*. In this case, a threshold is not applied to the output, and the amount of update depends on the difference $|y - \hat{y}|$. See [**?**].

The basic version of Winnow [**?**] actually uses two separate learning rates, one of promotion and another for demotion. Since the proof of the general bounds of Winnow is a lot harder than that one for perceptron, we will prove below a simple case, that will give some idea as to the properties of this algorithm. We will first write down the algorithm in a more explicit way.

| |
|---|
| Initialize $\theta = n$, $w = (1, 1, \ldots, 1)$ |
| Predict $c(x) = 1$ iff $w \cdot x \leq \theta$ |
| If prediction OK, do nothing |
| If $c(x) = 1$ but $w \cdot x \leq \theta$, double all the weights $w_i$ where $x_i = 1$ (promotion) |
| If $c(x) = 0$ but $w \cdot x \leq \theta$, divide all the weights $w_i$ where $x_i = 1$ by 2 (demotion) |

We will study the case of a learning the a function in the class $k$-disjunctions using Winnow. Clearly, this is a linear function. Notice that this class can be learned using a simpler *elimination* algorithm. (Think about it). Also, it is clear that Perceptron can also learning this class of functions. However for both these algorithms, the mistake bound is $O(n)$, where $n$ is the dimensionality of the instance space. (That is, there are sequences of examples on which they will make these many mistakes.) For winnow, we will show:

**Theorem 7.1** *Winnow makes $O(k \lg n)$ mistakes on $k$ disjunctions.*

**Proof:** Let $u$ be the number of false negatives (promotions), and $v$ the number of false positives (demotions) (Alternatively - eliminations). The total number of mistakes is $u + v$.

Intuition: the worry may be that we may go back a force many times, We could promote and then demote, and do it again and again.

Luckily, we are talking about disjunctions, so this intuition is false, since there are certain variables that we will never demote - those that really appear in the function. This intuition leads to the idea that we can bound the number of positive mistakes. Let us first see that we cannot promote too many times.

**Lemma 7.2** $u \leq k \cdot \log 2\theta$

**Proof:** We consider the number of mistakes done on **Positive examples.** Consider the $w_i$ that correspond to the variables in the disjunctions. Clearly, none of them is ever demoted, since they are never *active* in a negative example. Hence, they are only promoted. Every time a mistake is made on a positive example, at least one of these weights is promoted. This can go only until they all exceed *theta*. Then, there will not be any more mistakes on positive examples. This implies the bound.

■

Now, that we have a bound on the number of mistakes on positive examples, we see that it must bound also the number of mistakes on negative, since we cannot go back and forth too many times.

**Lemma 7.3** Lemma 1: $v \leq n/\Theta + u$
*We will argue by observing what happens to the total weight of the function. Initially,* $\sum_1^n = n$.

**Proof:** Here the argument is going to be a little more subtle. We will consider the total weight of the linear function, and use it to derive some relation between $u$ and $v$. What can be said on the total sum of weights ?

1. First consider the case of **mistakes on positive examples:** In each case, we double the weights of the *active* attributes. Since we made a mistake in this round, their sum is less than $\theta$, so we add **at most** $\theta$ to the total weight.
$$TW(t+1) \leq TW(t) + \theta$$
2. Now, the **Negative examples**
   Here, we demote (in WINNOW1 – eliminate) the weights of the *active* variables. Since we made a mistake on a negative example, we know that this sum exceeds $\theta$, and therefore, the decrease in the weights is **at least** $\theta/2$.
3. Putting these together, we can tie $u$ and $v$. We know that the sum of weights is bounded from above by its initial value ($n$) plus all the additions (due to the **at most**), minus all the subtractions (due to the **at least**). We also know that it remains positive. (The sum of the weights can never go below zero, since they are either doubled or zeroed, and they start positive.)

Thus
$$0 \leq \sum w_i \leq n + u.\theta - v.\theta \Rightarrow v \leq n/\theta + u$$
which implies the bound on $u + v$. ■

### 7.1 The general case

Proving the general mistake bound for the family of the multiplicative update algorithms is harder. See [**?**] for a discussion and some pointers. However, similar to the bound we gave for perceptron, we can give here a bound.

**Theorem 7.4** *The winnow algorithm learn a weight vector that correctly classifies all training data after at most $M$ updates, where,*

$$M = \frac{2 \ln 2N ||u||_1^2 \max_i ||x^i||_\infty^2}{\min_i (u \cdot x^i)^2}.$$

## 8 Relative Merits: Generalization (Convergence) Issues

To understand the relative merits of the two algorithms, we need to study the perceptron's bound:

$$M_p = ||u||_2^2 \max_i ||x^i||_2^2$$

and winnow's bound:

$$M_w = 2 \ln 2n ||u||_1^2 \max_i ||x^i||_\infty^2.$$

Notice that although these bounds are for the realizable case, very similar bounds could be shown also for the non-realizable case. That is, the case in which the data is not linearly separable. In this case, the quantities presented will be the *loss* of the algorithm relative to the best possible. That is, a bound of the form:

$$M = M_best + M_p(M_w)$$

where $M_best$ would be the number on mistakes made by the best linear separator on this data.

The bounds $M_p, M_w$ are not comparable. Each has cases in which it is better. Also, these bounds assume a good setting of parameters, which could be non trivial in both cases, and is typically a little harder for Winnow. (Two learning rate parameters; non zero initial weights).

For clarity, we consider two extreme cases, for the Boolean instance space $X = \{-1, 1\}^n$.

- First, assume that $u$ has exactly $k$ active features and the rest$n - k$ are inactive. That is, only $k$ input features are relevant for the prediction task. Then:
  - $||u||_2 = \sqrt{k}$
  - $||u||_1 = k$
  - $max||x||_2 = \sqrt{n}$
  - $max||x||_\infty = 1$
  - We now have:

    $$M_p = kn$$

    and

    $$M_w = 2k^2 \ln 2n.$$

  - Therefore, for $k << n$ the winnow bound can be much better.

- Assume now that $u = (1, 1, \ldots, 1)$, and the instances are very sparse, say, the rows of a $n \times n$ unit matrix. Then,
  - $||u||_2 = \sqrt{n}$
  - $||u||_1 = n$
  - $max||x||_2 = 1$
  - $max||x||_\infty = 1$
  - We now have:
    $$M_p = n$$
    and
    $$M_w = 2n^2 \ln 2n.$$
  - Therefore, perceptron algorithm clearly has a better bound.

Thus, the bounds are incomparable, and for large $n$ the difference can be arbitrarily large in either direction. Notice that although we talk about it in terms of mistake bounds, it is very easy to translate this to generalization performance. Intuitively, having better mistake bounds mean that, given unlimited data, one algorithm will converge faster to the true concept. Equivalently, if the amount of data is limited, it means that after a fixed amount of examples have been seen, the algorithm with the smaller mistake bound is *closer* to convergence, and thus is likely to make less mistake on new, unseen examples.

## 8.1 Comments on Features and Relevancy

In many cases people refer to learning algorithms for linear functions, in particular, Winnow, as algorithms that find the *relevant features*. This comment is meant to clarify that this is not a correct terminology. Actually, an old tradition in Machine Learning, suggested to (1) preprocess the data to find what are the relevant features (the feature selection process) and then (2) run a learning algorithm. This paradigm is due to the lack of some algorithms to deal with high dimensional data. In our case, though, there is not feature selection stage, but it is also not accurate to view the algorithms as selecting the "important" of features. The algorithms simply converge fairly rapidly to a good linear function.

In the extreme case, which I will phrase here for the case of conjunctive concept, the following holds:

**Theorem 8.1 ([?], Thm 5.1)** *Given a sample of data of dimensionality n that is consistent with some conjunctive hypothesis, it is NP-hard to find a conjunctive hypothesis that is both consistent with this sample and has the minimum number of attributes.*

This shows that the problem of finding the best set of features is computationally hard. Winnow is not doing it. It just finds a function that is consistent with the data (or, with the target concept), but not the relevant features in it.

# 9 Computational Issues

So far we have discussed the positive Winnow but, in general, this is not expressive enough. There is a standard mapping of the input space into a higher dimensional one $x = [x, -x]$. In this way, one can run the positive Winnow algorithm and use all the theoretical results as is. The problem is that in this way there is no way to take advantage of the sparse instance space, for computational reasons. Using this mapping, half of the features are going to be active in each example. When the dimensionality (number of potential features) is very high and the number of active features in each example is relatively small, this will have significant implications computationally.

The infinite attribution space [**?**] gives a way to do it. Basically, it represents examples as a list of all active features. Working this way, the examples observed by the learning algorithm are actually examples of variables size. This is the representation used in SNoW.

When working of a Boolean instance space, the general results that hold for Perceptron and Winnow hold also in this model.

# 10 Discussion: Relationship with PAC learning

Is the previous algorithm good also in a PAC sense ?
Given the hard bound on the number of mistakes, one might expect that a mistake bound algorithm is also a PAC learning algorithm.

Notice – the setting is different, but we can assume that we are give a mistake bound algorithm, and we are using it now in a PAC setting. The question is, will it guarantee a PAC performance ?

Of course, if we wait until we make all the mistakes, are guaranteed. But we can see many examples (w/o making mistakes) before we get to this number of mistakes.

**Theorem 10.1** *If the algorithm A learn a concept class C in the mistake bound model, then A also learns C in the PAC model.*

**Proof:** From the algorithm $A$ with a mistake bound $M$, we will construct $A_{PAC}$ in the following way:

Run $A$ on the data seen, halting if the hypothesis survives from more than $\frac{1}{\epsilon}\log\frac{1}{\delta}$ subsequent examples.
Return $h$ as the hypothesis of the algorithm $A_{PAC}$.

Since the number of mistakes of $A$ is bounded by $M$ we know for sure that $A_{PAC}$ will terminate within $\frac{M}{\epsilon}\log\frac{1}{\delta}$ examples.

The probability that the algorithm accepts a hypothesis with error greater than $\epsilon$ is bounded by the probability that a bad such hypothesis predicts $\frac{1}{\epsilon}\log\frac{1}{\delta}$ example correctly, which is at most

$$(1-\epsilon)^{\frac{1}{\epsilon}\log\frac{1}{\delta}} = ((1-\epsilon)^{\frac{1}{\epsilon}})^{\log\frac{1}{\delta}} < e^{\log\frac{1}{\delta}} = \delta$$

■

# 11 Other linear classifiers

Several other classification methods have been shown to be quite successful in NLP applications but we will not have time to present them in details.

*Transformation Based Learning* [**?**, **?**, **?**] is a methods that, in most applications, is equivalent to Decision Lists, although in general (e.g., that way it was used for POS tags) it is not even a classifier citeBrill95. See [**?**] for a presentation of TBL as a linear function.

*Memory Based Learning* [**?**, **?**] is another popular and successful classification method that had been used a lot in NLP. It is a lazy learning algorithm in the sense that it stores the training data as is and uses it to make decisions, based on a distance metric and a k nearest neighbor algorithm at test time. See [**?**] for details and analysis.

# 12 Appendix: Norms

**Definition 12.1** *We define the following* norms *on* $\mathcal{R}^n$*:*

1. *The* $l_1$ *norm:* $||x||_1 = \sum_{i=1}^n x_i$
2. *The* $l_2$ *norm:* $||x||_2 = (\sum_{i=1}^n x_i^2)^{1/2}$
3. *The* $l_p$ *norm:* $||x||_p = (\sum_{i=1}^n x_i^p)^{1/p}$
4. *The* $l_\infty$ *norm:* $||x||_\infty = \max_{1 \leq i \leq n} |x_i|$

**Theorem 12.1** *Any two norms on* $\mathcal{R}^n$ *are equivalent. In particular,*

1. $||x||_2 \leq ||x||_1 \leq \sqrt{n}||x||_2$
2. $||x||_\infty \leq ||x||_2 \leq \sqrt{n}||x||_\infty$
3. $||x||_\infty \leq ||x||_1 \leq n||x||_\infty$