CS 446: Machine Learning
Lecture 4, Part 3: On-Line Learning

---

1 Introduction

In the previous section we discussed SNoW which is a learning architecture that supports several linear update rules (Winnow, Perceptron, and Naive Bayes). SNoW has many options for regularization including pruning, and average Winnow/Perceptron. Later we will discuss what is meant by 'True' multi-class classification. SNoW allows variable sizes of examples and offers very good support for domains which are large-scale in terms of the number of examples and number of features. SNoW allows 'explicit' kernels, which will be discussed below. SNoW is very efficient, being 1-2 order of magnitude faster than SVMs. LBJ makes use of the SNoW architecture. SNoW can be downloaded from: `http://L2R.cs.uiuc.edu/~cogcomp`

2 Kernel Based Methods

A Kernel method is a method to run Perceptron on a very large feature set, without incurring the cost of keeping a very large weight vector. Computing the weight vector can still be done in the original feature space. The benefits pertain only to efficiency: The classifier is identical to the classifier that would result from blowing up the feature space. Generalization with Kernels is still relative to the real dimensionality (or related properties). Kernels were popularized by SVMs, although most applications actually use linear kernels. The basic equation for kernels (below) will be explained in the following sections.

$$f(x) = Th_\theta(\sum_{z \in M} S(z)K(x,z))$$

The basic idea behind kernels is as follows. First, promotion and demotion are defined as in previous examples:

If $Class = 1$ but $w \cdot x \leq \theta$, $w_i \leftarrow w_i + 1(if x_i = 1)$ (promotion)

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

Given the examples, $x \in \{0,1\}^n$, the hypothesis, $w \in R^n$, and the formula:

$$f(x) = Th_\theta(\sum_{i-1}^{n} w_i x_i(x))$$

Let I be the set $t_1, t_2, t_{3\ldots}$ of monomials (conjunctions) over the feature space $x_1, x_2, \ldots x_n$.

For example, $t_1 = x_1 x_2 x_4(11010) = 1$, The value of t_1 on the string 11010 is true. $t_2 = x_3 x_4(11010) = 0$. The value of t_2 on the string 11010 is false because x_3 is off.

We can then write a linear function over the new feature space:

$$f(x) = Th_\theta(\sum_{i \in I} w_i t_i(x))$$

This new representation allows a great increase in expressivity. It is possible to run Perceptron and Winnow, but the convergence bound may suffer exponential growth. In each example, an exponential number of monomials are true. Also, many weights still have to be stored. The problem is similar to that encountered by embedding, when we blow up the feature space to make the discriminator functionally simpler. The next section will discuss some ways of dealing with the accompanying problems.

2.1 The Kernel Trick

Given the rules for promotion and demotion in the previous section, consider the value of w used in the prediction. For each previous mistake, on and example, z, there is an additive contribution of $+/-1$ to w, iff $t(z) = 1$. The value of w is determined by the number of mistakes on which $t()$ was satisfied. The ones in which $t()$ was not satisfied make no contribution.

$$f(x) = Th_\theta(\sum_{i \in I} w_i t_i(x))$$

Let the set P be the set of examples which were Promoted and let the set D be the set of examples which were Demoted.

Let $M = P \cup D$

The weight vector w can be written equivalently as the sum of all promoted examples minus the sum of all demoted examples:

$$f(x) = Th_\theta(\sum_{i \in I} \left[\sum_{z \in P, t_i(z)-1} 1 - \sum_{z \in D, t_i(z)=1} 1 \right] t_i(x)) =$$

Let S be a function which tells what type of mistake was made on an example $z \in M$. Where $S(z) = 1$ if $z \in P$ and $S(z) = -1$ if $z \in D$. So S of z is $+1$ if there was a Positive mistake and -1 if there was a Negative. The formula is re-written as follows:

$$f(x) = Th_\theta(\sum_{i \in I} \left[\sum_{z \in M} S(z) t_i(z) t_i(x) \right])$$

Notice that, in the notation above, there is as sum on examples (i.e., z) and a sum on features, $i \in I$. So the formula can be reordered as below:

$$f(x) = Th_\theta(\sum_{z \in M} S(z) \sum_{i \in I} t_i(z) t_i(x))$$

A mistake on z contributes the value $+/-1$ to all monomials satisfied by z. The total contribution of z to the sum is equal to the number of monomials that satisfy both x and z. Therefore, we can define a dot product in the t-space:

$$K(x, z) = \sum_{i \in I} t_i(z) t_i(x)$$

By substitution, this gives the standard notation below:

$$f(x) = Th_\theta(\sum_{z \in M} S(z) K(x, z))$$

2.2 Kernel Based Methods

It has been shown above how the representation $f(x) = Th_\theta(\sum_{i-1}^{n} w_i x_i(x))$ can be written equivalently as $f(x) = Th_\theta(\sum_{z \in M} S(z) K(x, z))$. But what are the benefits? What does the new representation give us?

Consider the substitution instance below:

$$K(x,z) = \sum_{i \in I} t_i(z) t_i(x)$$

In this representation, we can view the Kernel as the distance between x, z in the $t-$space. But $K(x,z)$ can also be measured in the original space, without explicitly writing the t-representation of x, z.

With the Kernel Trick, given an example x, instead of computing t_i and then multiplying by the $+/-1$ weights, we take all examples z, on which we made mistakes, and compute the distance between the current point, x, and all those examples in M. Then we compute the distance via this Kernel function $K(x,z)$. The 'trick' is that it is still defined as a sum in the $I-$space. It is necessary to touch t_i of z, t_i of x. So far, the derivation is only about reordering terms. But the benefit is that $K(x,z)$ can be measured, or evaluated, in the original space without explicitly writing the representation of x and z. So, given x and z, you can compute this sum, the dot product over all I.

Consider the space of all 3^n monomials, allowing both positive and negative literals. Then, $K(x,z) = 2^{same(x,z)}$. Where $same(x,z)$ is the number of features that have the same value for both x and z. The resulting formula is:

$$f(x) = Th_\theta(\sum_{z \in M} S(z)(2^{same(x,z)}))$$

For example, take the case where $n = 2$; $x = (00), z = (01), \ldots$. Other Kernels can be used too.

2.2.1 Example

Consider a Boolean space over 4 variables: $X = \{x_1, x_2, x_3, x_4\}$, and let I be the space of all monomials over X. There are a total of 3^n in the $I-$space. In this example: $3^4 = 81$ for $|I|$. Some monomials in the $I - space$ are $x_1; x_1x_3; x_2x_3x_4$, for example.

Consider two points $x = (1100), z = (1101)$ in X. Their representation in the $I - space$ would be the list of all 81 monomials with values of 0 or 1 depending on whether or not the monomial represented that point:

$I(x)$ $x_1(1100) = 1$
$x_1x_2(1100) = 1$
$x_2x_3\overline{x_4}(1100) = 0$

$x_1x_2x_4(1100) = 0$
$\vdots$

$I(z)$ $x_1(1101) = 1$
$x_1x_2(1101) = 1$
$x_2x_3\overline{x_4}(1101) = 0$
$x_1x_2x_4(1101) = 1$
$\vdots$

Now, in order to compute $I(x) \cdot I(z)$, only the examples in which both $I(x)$ and $I(z)$ have a positive valuation contribute to the dot product. In this example, there are eight such cases:

$I(x) \cdot I(z)$

x_1
x_2
$\overline{x_3}$
x_1x_2
$x_1\overline{x_3}$
x_2x_3
$x_1x_2\overline{x_3}$
the constant monomial (everything is on)

Show that the following holds:

$$K(x, z) = I(x) \cdot I(z) = \sum it_i(z)t_i(x) = 2^{same(x,z)} = 8$$

Try to develop another kernel, for example where I is the space of all conjunctions of exactly size 3.

By doing the sum in the t-space looking over all the features, it is possible to just look at X and Z in the original space instead of the 81 dimensions of space. This type of kernel was the kernel of all monomials. For each type of kernel you will have to figure out how to compute it in the original space, but the general theorem says that if you have a kernel, then it is always possible to compute it in the original space.

2.3 Implementation

In order to implement the Kernel, Perceptron is run in an on-line mode and the algorithm keeps track of the set M. Keeping track of the set M allows us to keep track of $S(z)$. Rather than remembering the weight vector w, it is necessary to remember the set M (consisting of the sets P and D of all examples on which we made mistakes.

2.4 Summary - Kernel Trick (Polynomial Kernel)

Separating hyperplanes produced by Perceptron and SVM can be computed in terms of dot products over a feature based representation of examples.

We want to define a dot product in a high dimensional space. Given two examples $x = (x_1, x_2, \ldots, x_n)$ and $y = (y_1, y_2, \ldots, y_n)$, we want to map them to a high dimensional space.

An example of a mapping in a high dimensional space is that of quadratic examples:

$$\Phi(x_1, x_2, \ldots, x_n) = (1, x_1, \ldots, x_n, x_1^2, \ldots, x_n^2, x_1 \cdot x_2, \ldots, x_{n-1} \cdot x_n)$$

$$\Phi(y_1, y_2, \ldots, y_n) = (1, y_1, \ldots, y_n, y_1^2, \ldots, y_n^2, y_1 \cdot y_2, \ldots, y_{n-1} \cdot y_n)$$

And we want to compute the dot product $A = \Phi(x) \cdot \Phi(y)$. We could also compute this in the orginal space:

$$B = f(x \cdot y) = [1 + (x_1, x_2, \ldots, x_n) \cdot (y_1, y_2, \ldots, y_n)]^2$$

Theorem: $A = B$

2.5 Summary - Kernel Based Methods I

In summary, Kernels are a method to run Perceptron on a very large feature set, without incurring the cost of keeping a very large weigh vector. Computing the weight vector can, in fact, still be done in the original feature space. The benefits of Kernels pertain only to *efficiency*: The classifier is identical to the classifier that results from blowing up the feature space. *Generalization* is still relative to the real dimensionality (or related properties). Kernel were popularized by SVM, but the algorithms for using them with SVMs are different. Most applications today use linear kernels.

2.6 Efficiency-Generalization Tradeoff

There is a trade off between the computational efficiency with which these kernels can be computed and the generalization ability of the classifier. For example, using such kernels, the Perceptron algorithm can make an exponential number of mistakes, even when learning simple functions (Khardon, Roth, Servedio, NIPS'01; Ben David et al.). In addition to the high number of mistakes possible, computing with kernels depends strongly on the number of examples present. It turns out that sometimes working in the blown up space is more efficient than using kernels (Cumby, Roth ICML'03).

2.7 Learning from Structured Input

Consider the cases when we want to extract features from structured domain elements. It might be the case that we want to encode their internal (hierarchical) structure, for example, as in Figure 1 below. Features are useful for this. A feature is a mapping from the instance space to $\{0, 1\}$ or $[0, 1]$. With the appropriate representation language it is possible to represent expressive features that constitute an infinite dimensional space (for example, with a feature extractor like FEX). Learning can then be done in the infinite attribute domain.

What does it mean to extract features? Conceptually, it means that different data instantiations may be abstracted to yield the same representation (for instance, with quantified elements). Computationally it requires some kind of graph matching process as in Figure 2 below.

The challenge is to provide the expressivity necessary to deal with large scale and highly structured domains. It is also a challenge to meet the strong tractability requirements for these tasks.

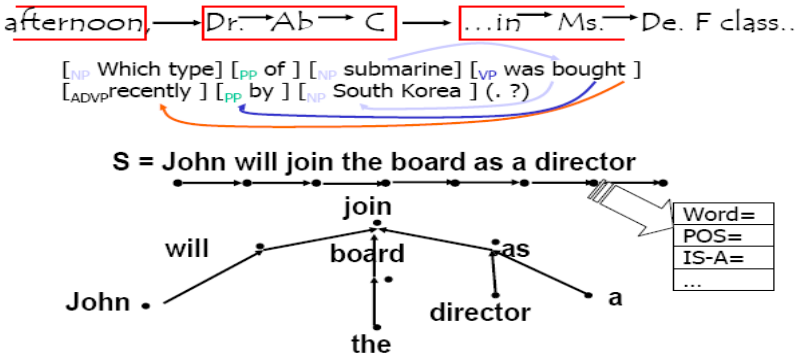

Figure 1: Structured Input

Only those descriptions that are ACTIVE in the input are listed. Kernels over parse trees have been developed (Collins) as well as parameterized kernels over structures (Cumby/Roth).

3 Kernels: Complexity

It's possible to define kernels for structured data. This is equivalent to blowing up the feature space by generating functions of primitive features. Is it worth doing in structured domains? Computationally, there is the hand-waving argument discussing which is preferable between $t_1 m^2$ and $t_2 m$. Where m is the number of examples and t_1, t_2 are the sizes of the feature space. Typically, $t_1 << t_2$, so the question really concerns the number of examples we need to consider. In theory, we do not need to consider all of the examples, but in practice, we usually need to consider quite a few.

4 Kernels: Generalization

It is important to use generalization with kernels. Using the most expressive kernels possible is equivalent to working in a larger feature space and will lead to overfitting.

The following is a simple argument that shows that simply adding irrelevant features does not help.

Given a linearly separable set of points $S = \{x_1, \ldots, x_n\} \in R^n$ with separator $w \in R^n$,

Embed S into an $n' > n$ dimensional space by adding zero-mean random noise e to the additional dimensions.

Then $w' \cdot x = (w, 0) \cdot (x, e) = w \cdot x$

So $w' \in R^{n\prime}$ still separates S.

What is the new margin?

$$\gamma(S, w') = min_s w'^T x' / \|w'\| \|x'\| = min_s w^T x / / \|w'\| \|x'\|$$

But $\|x'\| = \|(x, e)\| > \|x'\|$

The new margin is smaller and therefore bad for generalization.

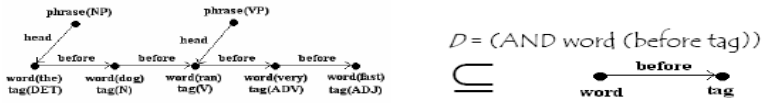

Figure 2: Graph of Features

5 Efficiency-Generalization Tradeoff

There is a tradeoff between the computational efficiency with which these kernels can be computed and the generalization ability of the classifier. For example, using such kernels, the Perceptron algorithm can make an exponential number of mistakes, even when learning simple functions (Khardon, Roth, Servedio, NIPS'01; Ben David et al.). In addition to this, computing with kernels depends strongly on the number of examples. It turns out that sometimes working in the blown up space is more efficient than using kernels (Cumby, Roth, ICML '03).

6 Winnow - General Setting

Given the update rules as before:

If $Class = 1$ but $w \cdot x \leq \theta$, $w_i \leftarrow w_i + 1(if x_i = 1)$ (promotion)

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

We have:

$$\forall i, w_{i+1} \leftarrow w_i \cdot \beta^{(y-1[w \cdot x])^{x_i}}$$

Where $(y - 1[w \cdot x])^{x_i}$ represents thresholding. In general, in vector notation, it can be written as:

$$w_{t+1} \leftarrow w_t \cdot \beta^{(y_t - w_t \cdot x_t)x_t}$$

This representation can be used and analyzed also for non-Boolean x's. The more common notation is $\alpha > 1$ for the promotion parameter and $0 < \beta < 1$ for the demotion parameter. The algorithm is no longer a mistake driven algorithm.

6.1 Predicting form Expert Advice

In this paradigm, an algorithm is given advice from a pool of experts. No quality or independence assumptions are made. An example is Weather Prediction. What is the best we can hope for?

In this learning protocol, a *trial* is a sequence of events in which the algorithm (1) receives the prediction of the experts (2) makes its own prediction, and (3) is told the correct answer. It is not possible in this case to achieve an absolute level of quality in the prediction. A more reasonable goal is to perform nearly as well as the best expert so far.

6.2 Weighted Majority Algorithm

This algorithm maintains a list of weights, $w_1, w_2, \ldots, w_n$. There is one weight for each expert and the algorithm makes predictions based on a weighted majority vote of the expert opinions.

1. Initialize: for all $I = 1, \ldots, n : w_i = 1$

2. Given a set of predictions, $x_1, x_2, \ldots, x_n$, by the experts, output the prediction with the highest total weight. That is, output 1 if $\sum_{\{i:x_i=1\}} w_i \geq \sum_{\{i:x_i=0\}} w_i$

3. If I is the correct answer and $l \neq x_i$, then penalize: $w_i \leftarrow w_i/2$

The claim is that the number of mistakes made by the weighted majority algorithm is never more than $2.41(m + logn)$ where m is the number of mistakes made by the best expert so far.

So, given:

$$\sum_{\{i:x_i=1\}} w_i \geq \sum_{\{i:x_i=0\}} w_i$$

Let W be the total weight of the experts. Initially, $W = n$.

If at least half of the experts made a mistake, W is reduced by at least a factor of $1/4$. So, generally, after M mistakes:

$$W \leq n(\frac{3}{4})^M$$

The *best expert* made m mistakes, thus $W \geq \frac{1}{2^m}$. Combining gets:

$$\frac{1}{2^m} \leq n(\frac{3}{4})^M \Rightarrow M \leq 2.41(m + logn)$$

6.2.1 Randomized Version

The weight of the experts can be viewed as probabilities:

Choose to output x_i with probability $w_i / \sum w_i$

The algorithm can be applied when the experts are strategies or other things that cannot be combined together. If the experts are programs, it is not necessary to run all of them in order to predict.

6.2.2 Summary - Weighted Majority

The Weighted Majority Algorithm provides a way to accommodate inconsistent training data. There are no requirements that the expert is mistake-bounded. Also, the experts can be very general. There is no statistical assumption and optimal bounds.

6.3 Mistake Bound and PAC

Every Mistake-Bound Algorithm can be converted efficiently to a PAC algorithm. In the mistake bound model, we don't know when we will make the mistakes. But in the PAC model, we want dependence on the number of examples seen and not the number of mistakes. In order to convert, we wait for a long stretch of examples with no mistakes (there must be one), then use the hypothesis at the end of this stretch. The algorithms PAC behaviour is relative to the length of the stretch.