CS 446 Machine Learning Fall 2016 SEP 27, 2016

Online Learning With Kernel

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

Overview

- Stochastic Gradient Descent Algorithms
- Regularization
- Algorithm Issues
- Kernels

1 Stochastic Gradient Algorithms

Given examples $\{z = (x, y)\}_{1,m}$ from a distribution over $X \times Y$, we are trying to learn a linear function parameterized by a weight vector w such that we minimize the expected risk function

$$J(w) = E_z Q(z, w) \approx \frac{1}{m} \sum_{i=1}^{m} Q(z_i, w_i)$$

In Stochastic Gradient Descent (SGD) Algorithms we approximate this minimization by incrementally updating the weight vector w as follows:

$$w_{t+1} = w_t - r_t g_w Q(z_t, w_t) = w_t - r_t g_t$$

Where $g_t = g_w Q(z_t, w_t)$ is the gradient with respect to w at time t

With this in mind, all algorithms in which we perform SGD vary only in the choice of loss function $Q(z, w)$.

1.1 Loss functions

Least Mean Squares Loss
One loss function that we have discussed already is Least Mean Squares (LMS), given by:

$$Q((x, y), w) = \frac{1}{2}(y - w \cdot x)^2$$

Combining this with the update rule above, we produce the LMS update rule, also called *Widrow's Adaline*:

$$w_{t+1} = w_t + r(y_t - w_t \cdot x_t)x_t$$

Note that even though we make binary predictions based on $sgn(w \cdot x)$, we do not take the actual sign of the dot product into account in the loss.

Hinge Loss
In the Hinge loss case, we want to assign no loss if we make the correct prediction, as given by

$$Q((x, y), w) = max(0, 1 - yw \cdot x)$$

Hinge loss leads to the perceptron update rule

If $y_i w_i \cdot x_i > 1$
    `No update`
`Else`
    $w_{t+1} = w_t + r y_t x_t$

Adagrad
Thusfar we've focused on fixed learning rates, but these can change. Imagine an algorithm that adapts its update based on historical information; frequently occurring features get small learning rates and infrequent features get higher ones. Intuitively, this algorithm would learn slowly from features that change a lot, but really focus on those features that don't make frequenct changes.

Adagrad is one such algorithm. It assigns a per-feature learning rate for feature j at time t, defined as

$$r_{t,j} = \frac{r}{G_{t,j}^{\frac{1}{2}}}$$

Where $G_{t,j} = \sum_{k=1}^{t} g_{k,j}^2$, or the sum of squares of gradients at feature j until time t. The update rule for Adagrad is then given by

$$w_{t+1,j} = w_{t,j} - \frac{r g_{t,j}}{G_{t,j}^{\frac{1}{2}}}$$

In practice this algorithm should update weights faster than Perceptron or LMS.

1.2 Regularization

One problem in theoretic machine learning is the need for regularization. In addition to the risk function, we add R, a *regularization term*, that is used to

prevent our learned function from overfitting on the training data. Incorporating R, we now seek to minimize

$$J(w) = \sum_{i=1}^{m} Q(z_i, w_i) + \lambda R_i(w_i)$$

In decision trees, this idea is expressed through pruning.

We can apply this to any loss function

LMS: $Q((x, y), w) = (y - w \cdot x)^2$

Ridge Regression: $R(w) = ||w||_2^2$

LASSO problem: $R(w) = ||w||_1$

Hinge Loss: $Q((x, y), w) = max(0, 1 - yw \cdot x$

Support Vector Machines: $R(w) = ||w||_2^2$

Logistic Loss: $Q((x, y), w) = \log(1 + exp\{-yw \cdot x\})$

Logistic Regression: $R(w) = ||w||_2^2$

It is important to consider why we enforce regularization through the size of w. In later lectures we will discuss why smaller weight vectors are preferable for generalization.

Clarification on Notation
Note that the above equations for R reflect different norms, which can be understood as different ways for measuring the size of w.

The L1-norm given by $||w||_1$ is the sum of the absolute values, or $\sum_i |w_i|$.

The L2-norm given by $||w||_2||$ is the square root of the sum of the squares of the values, or $\sqrt{\sum_i w_i^2}$.

Thus, the squared L2-norm given above – $||w||_2^2$ – refers to the sum of the squares, or $\sum_i w_i^2$

Note also that in the general case, the L-p norm is given by $||w||_p = (\sum_{i=1}^{n} |w_i|^p)^{\frac{1}{p}}$.

2 Comparing Algorithms

Given algorithms that learn linear functions, we know that each will eventually converge to the same solution. However, it is desirable to determine a measure

of comparison; generalization (how many examples do we need to reach a certain performance), efficiency (how long does it take to learn the hypothesis), robustness to noise, adaptation to new domains, etc.

Consider the context-sensitive spelling example

I don't know {whether,weather} to laugh or cry

To learn which {whether,weather} to use, we must first define a feature space (properties of the sentence) and then map the sentence to that space. Importantly, there are two steps in creating a feature representations, which can be framed as the questions:

1. Which information sources are available? (sensors)
2. What kinds of features can be constructed from these sources? (functions)

In the context-sensitive spelling example, the sensors include the words themselves, their order, and their properties (part-of-speech, for example). These sensors must be combined with functions because the sensors themselves may not be expressive enough.

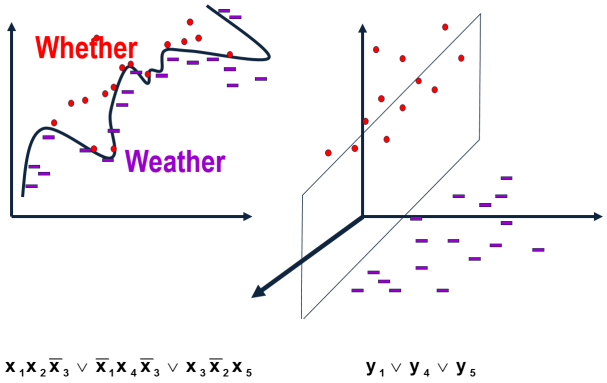


Figure 1: Words as Features v. Sets of Consecutive Words

In our example, words on their own may not tell us enough to determine which {whether,weather} to use, but conjunctions of pairs or triples of words may be expressive enough. Compare the left of Figure 1 – representing words on their own – with the right of the figure, representing the feature space of functions over those words.

3 Generalization

In most cases, the number of potential features is very large, but the instance space – the combination of features we actually see – is sparse. Further, decisions

usually depend on a small set of features, making the function space sparse as well. In certain domains – like natural language processing – both the instance space and the function space is sparse, and it is therefore necessary to consider the impact of this sparsity.

3.1 Sparsity

Generalization bounds are driven by sparsity. For multiplicative algorithms, like Winnow, the number of required examples[1] largely depends on the number of relevant features, or the size of the target hyperplane: $||w||$). For additive learning algorithms like Perceptron, the number of required examples largely depends on the number of relevant input examples $||x||$.

Multiplicative Algorithms
Bounds depend on the size of the separating hyperplane $||u||$. Given n, the number of features, and i the index of a given example, the number of examples M that we need is given by

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

where $||x||_\infty = \max_i |x_i|$. Multiplicative learning algorithms, then, do not care much about the data (since $||x||_\infty$ is just the size of the largest example point) and rely instead on the L1-norm of the hyperplane, $||u||_1$.

Additive Algorithms
Additive algorithms, by contrast, care a lot about the data, as their bounds are given by

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

These additive algorithms rely on the L2-norm of X.

3.2 Examples

The distinction between multiplicative and additive algorithms is best seen through the extreme examples where their relative strengths and weaknesses are most prominent.

Extreme Scenario 1
Assume the u has exactly k active features, and the other $n - k$ are zero. Only k input features are relevant to the prediction. We thus have

[1] Though we discuss this in later lectures, 'required examples' can be thought of as the number of examples the algorithm needs to see in order to produce a good hypothesis

$$||u||_2 = k^{\frac{1}{2}}$$

$$||u||_1 = k$$

$$\max ||x||_2 = n^{\frac{1}{2}}$$

$$\max ||x||_\infty = 1$$

We can now compute the bound for perceptron as $M_p = kn$, while that for Winnow is given by $M_w = 2k^2 \ln(2n)$. Therefore, in cases where the number of active features is much smaller than the total number of features ($k << n$), Winnow requires far fewer examples to find the right hypothesis (logarithmic in n versus linear in n).

Extreme Scenario 2
Now assume that all the features are important ($u = (1, 1, 1, ..., 1)$), but the instances are very sparse (only one feature on). In this case the size (L1-norm) of u is n, but the size of the examples is 1.

$$||u||_2 = n^{\frac{1}{2}}$$

$$||u||_1 = n$$

$$\max ||x||_2 = 1$$

$$\max ||x||_\infty = 1$$

In this setting, perceptron has a much lower mistake bound than Winnow, given by $M_p = n; M_w = 2n^2 \ln 2n$.

An intermediate case...
Assume an example is labeled positive when l out of m features are on (out of n total features). These l of m of n functions are good representatives of linear threshold functions in general. A comparison between the perceptron and Winnow mistake bounds for such functions is shown in Figure 2. In l out

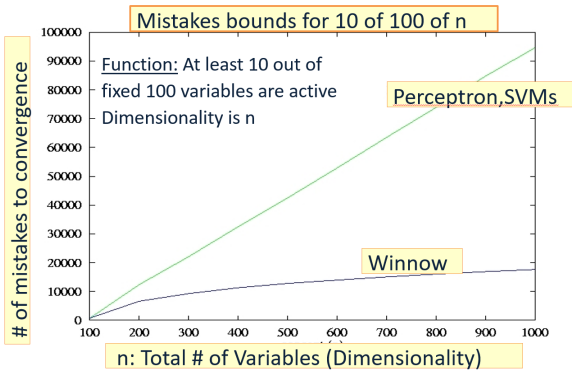


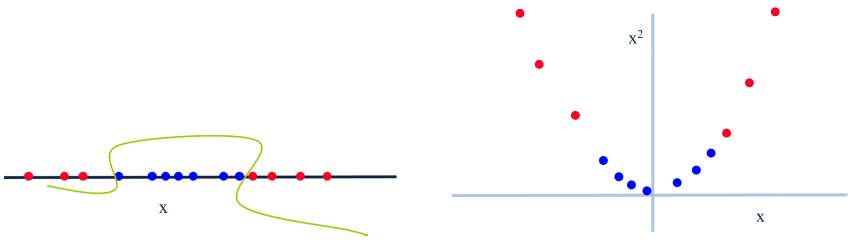
Figure 2: Comparison of Perceptron and Winnow

of m out of n functions, the perceptron mistake bound grows linearly, while the Winnow bound grows with $\log(n)$.

In the limit, all algorithms behave in the same way. But the realistic scenario – that is, the one with a limited number of examples – requires that we consider which algorithms generalize better.

3.3 Efficiency

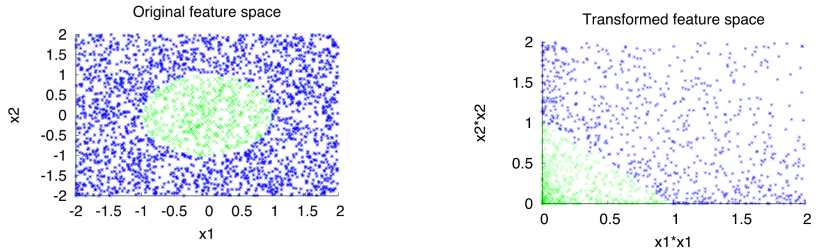
Efficiency depends on the size of the feature space. It is often the case that we don't use simple attributes, and instead treat functions over attributes (ie. conjunctions) as our features, making efficiency more difficult. Consider the case shown in Figure 3a, which is not linearly separable until the feature space is blown up, shown in Figure 3b.



(a) Not separable in one dimension (b) Separable by blow up dimension

In additive algorithms we can behave as if we've generated complex features while still computing in the original feature space. This is known as the *kernel trick*.

Consider the function $f(x) = 1$ iff $x_1^2 + x_2^2 \leq 1$, shown in Figure 4a.



(a) Not separable in original space (b) Separable in transformed space

This data cannot be separated in the original two dimensional space. But if we transform the data to be $x_1' = x_1^2$ and $x_2' = x_2^2$, the data is now linearly separable.

Now we must consider how to learn efficiently, given our higher dimensional space.

4 Dual Representation

Consider the perceptron: given examples $x \in \{0,1\}^n$, hypothesis $w \in \mathbb{R}^n$ and function $f(x) = Th_\theta(\sum_{i=1}^{n} w_i x_i(x))$

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

Note, here, that rather than writing x_i, we are writing $x_i(x)$, which can be read as a function on x that returns the i^{th} value of x. Note, also, that Th_θ refers to the θ threshold on the dot product of w and x. This notation will be useful later.

Assume we run perceptron with an initial w and we see the following examples: $(x^1, +)$, $(x^2, +)$, $(x^3, -)$, $(x^4, -)$. Further assume that mistakes are made on x^1, x^2 and x^4.

The resulting weight vector is given by $w = w + x^1 + x^2 - x^4$; we made mistakes on positive examples x^1 and x^2 and negative examples on x^4. This is the heart of the dual representation. Because they share the same space, w can be expressed as a sum of examples on which we made mistakes, given by

$$w = \sum_{i=1}^{m} r\alpha_i y_i x_i$$

where α_i is the number of mistakes made on x_i.

Since we only care about $f(x)$, rather than w, we can replace w with a function over all examples on which we've made mistakes.

$$f(x) = w \cdot x = (\sum_{1,m} r\alpha_i y_i x_i) \cdot x = \sum_{1,m} r\alpha_i y_i (x_i \cdot x)$$

5 Kernel Based Methods

Kernel based methods allow us to run perceptron on a very large feature space, without incurring the cost of keeping a very large weight vector.

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

The idea is that we can compute the dot product in the original feature space instead of the blown up feature space.

It is important to note that this method pertains only to efficiency. The resulting classifier should be identical to the one you compute in the blown up feature space. Generalization is still relative to that of the original dimensions.

Consider a setting in which we're interested in using the set of all conjunctions between features, The new space is the set of all monomials in this space, or 3^n (possibly $x_i, \neg x_i, 0$ in each position). We can refer to these monomials as $t_i(x)$, or the i^{th} monomial for x.

Thus the new linear function is

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

In this space, we can now represent any Boolean function. We can still run perceptron or Winow, but the convergence bound will suffer exponential growth.

Consider that each mistake will make an additive contribution to w – either $+1$ or -1 – *iff* $t(z) = 1$. Therefore, the value of w is actually determined by the number of mistakes on which $t()$ was satisfied.

To show this more formally, we now denote P as the set of examples on which we promoted, D to be the set of examples on which we demoted, and M as the set of mistakes ($P \cup D$).

$$\begin{aligned} f(x) &= Th_\theta(\sum_{i \in I}[\sum_{z \in P, t_i(z)=1} 1 - \sum_{z \in D, t_i(z)=1} 1]t_i(x)) \\ &= Th_\theta(\sum_{i \in I}[\sum_{z \in M} S(z)t_i(z)t_i(x)]) \\ &= Th_\theta(\sum_{z \in M} S(z) \sum_{i \in I} t_i(z)t_i(x)) \end{aligned} \tag{1}$$

where $S(z) = 1$ if $z \in P$ and $S(z) = -1$ if $z \in D$

In the end, we only care about the sum of $t_i(z)t_i(x)$. The total contribution of z to the sum is equal to the number of monomials that satisfy both x and z.

We define this new dot product as

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

We call this the *kernel function* of x and z. Given this new dot product, we can transform the function into a standard notation

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

Now we can think of the function $K(x, z)$ to be some distance between x and z in the t-space. However, it can be calculated in the original space, without explicitly writing the t-representation of x and z.

Monomial Example
Consider the space of all 3^n monominals (allowing both positive and negative literals), then

$$K(x, z) = \sum_{i \in I} t_i(z) t_i(x) = 2^{same(x,z)}$$

where $same(x, z)$ is the number of features that have the same value for both x and z.

Using this we can compute the dot product of two size 3^n vectors by looking at two vectors of size n. This is where the computational gain comes in.

Assume, for example, that $n = 3$, where $x = (001), z = (011)$. There are $3^3 = 27$ features in this blown up feature space. Here we know $same(x, z) = 2$ and in fact, only $\neg x_1$, x_3, $\neg x_1 \vee x_3$ and *null* are satisfying conjunctions that $t_i(x)t_i(z) = 1$.

We can state a more formal proof. Let $k = same(x, z)$. A monomial can only survive in two ways: choosing to include one of the k literals with the right polarity in the monomial (negate or not) or choosing to not include it at all. That gives us 2^k conjunctions.

5.1 Implementation

We now have an algorithm to run in the dual space. We run a standard perceptron, keeping track of the set of the set of mistakes M, which allows us to compute $S(z)$ at any step.

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

where $K(x, z) = \sum_{iinI} t_i(z) t_i(x)$

Polynomial Kernel
Given two examples $x = (x_1, x_2, ...x_n)$ and $y = (y_1, y_2, ...y_n)$, we want to map them to a high dimensional space. For example,

$$\Phi(x_1, x_2, ...x_n) = (1, x_1, ...x_n, x_1^2, ...x_n^2, x_1 x_2 ...x_n)$$

$$\Phi(y_1, y_2, ...y_n) = (1, y_1, ...y_n, y_1^2, ...y_n^2, y_1 y_2 ...y_n)$$

Let $A = \Phi(x)^T \Phi(y)$

Instead of computing quantity A, we want to compute quantity the quantity $B = k(x, y) = [1 + (x1, x_2...x_n)^T (y_1, y_2...y_n)]^2$. The claim is that $A = B$; though coefficients differ, the learning algorithm will adjust the coefficients anyway.

5.2 General Conditions

A function $K(x, z)$ is a valid kernel if it corresponds to an inner product in some (perhaps infinite dimensional) feature space.

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

Consider the following quadratic kernel

$$\begin{aligned} K(x, z) &= (x_1 z_1 + x_2 z_2)^2 \\ &= x_1^2 z_1^2 + 2 x_1 z_1 x_2 z_2 + x_2^2 z_2^2 \\ &= (x_1^2, \sqrt{2} x_1 x_2, x_2^2)(z_1^2, \sqrt{2} z_1 z_2, z_2^2) \\ &= \Phi(x)^T \Phi(z) \end{aligned} \tag{2}$$

It is not always necessary to explicitly show feature function Φ. We can instead construct a kernel matrix $\{k(x_i, z_j\}$, and if matrix is positive semi definite, it is a valid kernel.

Kernel Matrix
The Gram matrix of a set of n vectors $S = \{x_1, ...x_n\}$ is the $n \times n$ matrix G with $G_{ij} = x_i x_j$. The kernel matrix is the Gram matrix of $\{\Phi(x_1), ...\Phi(x_n)\}$

The size of the kernel matrix depends on the number of examples, not the dimensionality.

A direct way can be done if you have the value of all $\Phi(x_i)$. You can just see if the matrix is semi-definite or not.

An indirect way is if you have the kernel functions, write down the Kernel matrix K_{ij} and show that it is a legitimate kernel, without explicitly construct $\Phi(x_i)$.

Example Kernels

Linear Kernel

$$K(x, z) = xz$$

Polynomial Kernel of degree d

$$K(x, z) = (xz)^d$$

Polynomial Kernel up to degree d

$$K(x, z) = (xz + c)^d, (c > 0)$$

5.3 Constructing New Kernels

It is possible to construct new kernels from existing ones.

Multiplying kernels by constants

$$k'(x, x') = ck(x, x')$$

Multiplying kernel $k(x, x')$ by a function f applied to x and x'

$$k'(x, x') = f(x)k(x, x')f(x')$$

Applying a polynomial (with non-negative coefficients to $k(x, x')$

$$k'(x, x') = P(k(x, x'))$$

with

$$P(z) = \sum_i a_i z^i, (a_i \geq 0)$$

Exponentiating kernels

$$k'(x, x') = exp(k(x, x'))$$

Adding two kernels

$$k'(x, x') = k_1(x, x') + k_2(x, x')$$

Multiplying two kernels

$$k'(x, x') = k_1(x, x')k_2(x, x')$$

Also, if $\Phi(x) \in R^m$ and $k_m(z, z')$ is a valid kernel in R^m, then

$$k(x, x') = k_m(\Phi(x), \Phi(x'))$$

is a valid kernel.

If A is a symmetric positive semi-definite matrix, $k(x, x\prime) = xAx\prime$ is a valid kernel.

5.4 Gaussian Kernel

Consider the Guassian Kernel, given by

$$k(x, z) = exp(-(x - z)^{\frac{2}{c}}$$

where $(x - z)^2$ is the squared Euclidean distance between x and z and $c = \sigma^2$ is a free parameter.

This can be thought of in terms of the distance between x and z; if x and z are very close, the value of the kernel is 1, and if they are very fall apart, the value is 0.

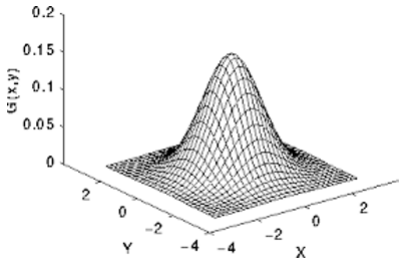


Figure 5: Gaussian Kernel

We can also consider the property of c; a very small c means $k \approx I$ (every item is different), and a very large c means $k \approx$ union matrix (all items are the same).

The Guassian Kernel is valid, given the following

$$\begin{aligned} k(x,z) &= exp(\frac{-(x-z)^2}{2\sigma^2}) \\ &= exp(\frac{-(xx+zz-2xz)}{2\sigma^2}) \\ &= exp(\frac{-xx}{2\sigma^2})exp(\frac{xz}{\sigma^2})exp(\frac{-zz}{2\sigma^2}) \\ &= f(x)exp(\frac{xz}{\sigma^2})f(z) \end{aligned} \tag{3}$$

$exp(\frac{xz}{\sigma^2})$ is a valid kernel because xz is the linear kernel and we can multiply it by constant $\frac{1}{\sigma^2}$ and then exponetiaite it.

Here however, we cannot easily explicitly blow up the feature space and get an identical representation since it is an infinite dimensional kernel.

6 Generalization / Efficiency Tradeoffs

There is a tradeoff between the computational efficiency with which these kernels can be computed and the generalization ability of the classifier.

For perceptron, for example, consider using a polynomial kernel when you're unsure if the original space is expressive enough. If it turns out that the original space was expressive enough, however, the generalization will suffer because we're now unnecessarily working in an exponential space.

We therefore need to be careful to choose whether to use the dual or primal space. This decision depends on whether you have more examples or more features.

Dual space has $t_1 m^2$ computation time

Primal space has $t_2 m$ computation time

where t_1 is the size of dual representation feature space, t_2 is that of the primal space, and m is the number of examples.

Typically $t_1 << t_2$ because t_2 is the blown up space, so we need to compare the number of examples with the growth in dimensionality.

As a general rule of thumb, *if we have a lot of examples, we should stay in the primal space.*

In fact, most applications today use explicit kernels; that is, they blow up the feature space and work directly in that new space.

6.1 Generalization

Consider the case in which we want to move to the space of all combinations of three features. In many cases, most of these combinations will be irrelevant; you may only care about certain combinations. In this case, the most expressive kernel – a polynomial kernel of size 3 – will lead to overfitting.

Assume a linearly separable set of points $S = \{x_1 ... x_n\} \in \mathbb{R}^n$ with separator $w \in \mathbb{R}^n$.

We want to embed S into a higher dimensional space $n' > n$ by adding zero-mean random noise e to the additional dimensions.

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

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

Now we will look at $\frac{\gamma}{||x||}$ which we have shown to be inversely proportional to generalization (mistake bound).

$$\begin{aligned} \frac{\gamma(S, w\prime)}{||x\prime||} &= \frac{min_s w\prime^T x\prime}{||w\prime||||x\prime||} \\ &= \frac{min_s w^T x}{||w||||x\prime||} \\ &< \frac{\gamma(S, w)}{||x||} \end{aligned} \tag{4}$$

Since

$$||x\prime|| = ||(x, e)|| > ||x||$$

We can see we have a larger ratio, which means generalization suffers. In essence, adding a lot of noisy/irrelevant features cannot help.