Learning Languages from Examples

Vinay Chaudhary
Advised by: Leonid Kontorovich

May 2, 2007

1 Summary

In this project I explore the machine learning problem of learning formal languages from positive and negative examples. I describe a technique proposed by Kontorovich, Cortes, and Mohri in their paper titled "Learning Linearly Separable Languages." The premise of this paper is that there exists a family of languages that are linearly separable and efficiently computable under a certain computable embedding. The approach described is to learn languages by mapping positive and negative strings to a high dimensional feature space (an infinite dimensional hypercube) and learning a separating hyperplane for that space.

My original goal was to empirically deduce the separating margin for the hyperplane in terms of automaton complexity. After this, I had planned on using these results as I guiding force to a mathematical proof of a bound for this separating margin. Unfortunately, I was unable to achieve this goal. I did, however, manage to gain a firm understanding of support vector classification and the mathematics behind it. I also managed to gain insight into the inherit "hardness" of language learning based on number of states. Specifically I concluded that it is not enough to simply try many random machines, but instead study the structure of the hardest machines to learn, and use them as a guide the reevaluate the margin bound question.

In this paper, I will first explore thorougly the background material on support vector classification, and then delve into my own work.

2 Learning Languages

2.1 Motivation

Suppose a teacher gives strings $x_i \in \Sigma^*$ and labels $y_i \in \{-1, +1\}$ and tells you these are consistent with some regular language L. Your task is to produce a "reasonable" hypothesis $\hat{L}$. More specifically, given a new string $\hat{x}$, determine whether that string $\in L$. We now examine two possible approaches to solving this problem.

First, the "obvious" approach. We construct minimal DFA $\hat{A}$ consistent with the sample. That is,

$x_i \in L(\hat{A})$ iff $y_i = +1$. This defines the language $\hat{L}$

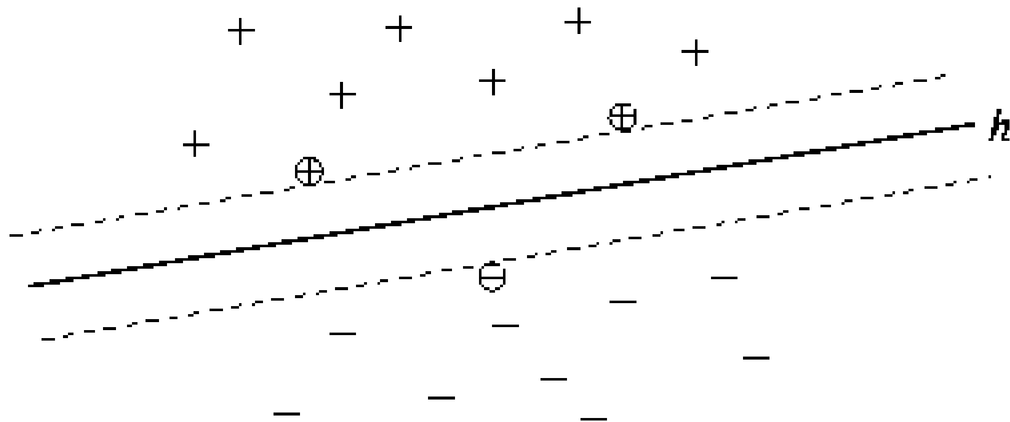
Now to classify the new string $\hat{x}$, run the DFA $\hat{A}$ on input $\hat{x}$, and output the result.

This approach seems like a good idea. Infact, the error bound is proven to decay rapidly with number of samples (This is known as the Occam's Razor bound)[5]. Unfortunately the problem of finding a minimally consistent DFA is NP-Complete[3][4]. So unless P=NP, this isn't going to be a polynomial time operation.

Another approach, which will be further discussed in this paper, is Support Vector Classification. The main idea of this approach is to embed the sample strings in an infinite dimensional vector space and learn separating hyperplanes.

Then in order to classify a new string $\hat{x}$, determine which side of the hyperplane the feature embedding of a given string lies on. Since the hyperplane can just be described as a normal vector of a plane through the origin, this is as simple as computing the dot product. The question of course arises, how can one perform a dot product of two infinite dimensional vectors in finite time? It can't always be done, but it can if the hyperplane has finite support, which is part of the requirement for linear separability. This topic will be discussed next.

Just to give a visual, here is an example of a hyperplane separating positive and negative examples. If you draw a normal to the plane, you can easily see how the sign of the dot product would give the classification of the strings. Also note the circled examples. These are the *support vectors* of the hyperplane h. This will also be discussed in more detail in the remainder of the paper. Finally, note the dotted line between h and the first support vector. The distance between h and this line is the *seperating margin*.



2.2 Linear separability

In order for a language to be learnable using support vector classification, it needs to be linearly separable. We define linear seperability as follows:

Fix some embedding $\phi : \Sigma^* \to \{0,1\}^{\mathbb{N}}$, and require ϕ to have finite support: $\|\phi(x)\|_0 < \infty$ for all $x \in \Sigma^*$. That is, the number of non-zero components in $\phi(x)$ is finite. Then, a language L is linearly

separable (under ϕ) if there is a $\mathbf{w} \in \mathbb{R}^{\mathbb{N}}$ with $\|\mathbf{w}\|_0 < \infty$ s.t.

$$L = \{x \in \Sigma^* : \langle \mathbf{w}, \phi(x) \rangle > 0\}$$

We say that a set of languages $\mathcal{C}$ is linearly separable if there is some ϕ with finite support s.t. all $L \in \mathcal{C}$ are linearly separable under ϕ. w here defines the separating hyperplane.

So what languages are actually linearly seperable? It turns out that all of the regular languages are actually linearly seperable! But before we go on to describe this result, we will describe kernels and support vectors.

2.3 Support Vectors and Kernels

Part of the goal of support vector classification is taking a less managable problem, and transforming it into a more computationally feasible problem. An obvious approach to classifying strings would be to actually learn w exactly, compute $\phi(x)$, and take take the dot product of the two vectors. This can unfortunately be very computationally expensive.

So instead, when we do support vector classification, we don't actually have to compute w or $\phi(x)$.

First we create "support strings". These "support strings" implicity define the separating hyperplane. But there is still a problem, because we still need to compute the embedding of each of the support strings to get the support vectors. Any linear separable language can be defined with a finite number of support strings and their corresponding vectors.

Now, we define $K(x, y) = \langle \phi(x), \phi(y) \rangle$. K is also important to support vector classification, as it allows us to recover the result of the inner product without actually performing it. In fact, we need not even create the embedded vectors!

2.4 Linear Separability of the Regular Languages

Now that we have the framework in place, lets define a regular language L.

Any regular language L can be represented by some "support strings" $\{s_i \in \Sigma^* : 1 \leq i \leq m\}$ with weights $\alpha \in \mathbb{R}^m$:

$$L = \left\{ x \in \Sigma^* : \sum_{i=1}^{m} \alpha_i K(s_i, x) > 0 \right\}$$

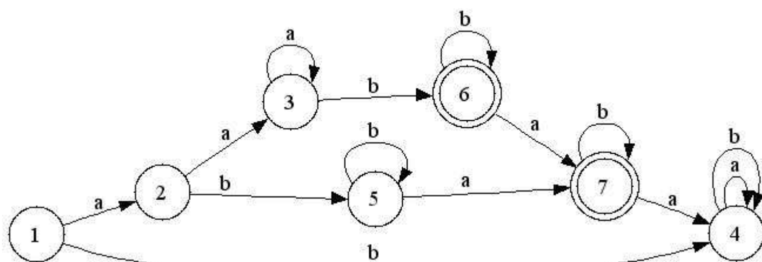
One problem though is that K may not be efficiently computatable. Fortunately, there is a very interesting subset of the regular languages where K is efficiently computable. These are the piecewise testable languages.

2.5 Piecewise Testable Languages

A language L is said to be *n-piecewise-testable* ($n \in \mathbb{N}$), if whenever u, v have the same subsequences of length atmost n, and u is in L, v is also in L. A language is said to be piecewise-testable if it is *n-piecewise-testable* for some $n \in \mathbb{N}$

Alternatively we could define a piecewise testable language as a finite boolean combination of shuffle ideas. A shuffle ideal of a string x is defined as the set of all strings containing x as a contiguous or discontiguous subsequence.

Furthermore, PT-DFAs exhibit an interesting structure. They essentially form a lattice work with a path of accept states. For visualization purposes, here is an example of a typical piecewise testable DFA. It was randomly created via a matlab script and then converted to graphviz format.



Piecewise testable languages are linearly seperable under a subsequence embedding ϕ. More specifically, ϕ is defined in the following way:

$$\phi(x)_u = \begin{cases} 1 & \text{if } u \sqsubseteq x \\ 0 & \text{otherwise} \end{cases}$$

You'll note that ϕ is indexed by subsequences in Σ^*. This may seem jarring at first, but remember that subsequences in Σ^* are countable, so this is equivalent to indexing with positive integers.

The kernel, K for the subsequence embedding is efficiently computable in time $O(|\Sigma| * |x| * |y|)$ using a dynamic programming approach. K computes the number of subsequences common to both x and y.

2.6 Classifying new strings

We already saw the definition of the language L in terms of the separating hyperplane. But lets formally define exactly what needs to be done to classify a new string in L given positive and negative samples from L.

First we use Support Vector Classification to find hyperplane support vectors. This gives us α, the vector of the weights given to each support vector. To classify a new string $\hat{x}$, do the following procedure.

Let n be the number of support vectors. Then the label of $\hat{x}$ is given by
$\hat{y} = sgn(\sum_{i=0}^{n} \alpha_i * y_i * K(\hat{x}, x_i))$.

2.7 Classification error, Margin Bound

It is true that given all the correct support vectors, the classification algorithm will correctly classify 100 percent of all strings. Recall that the support vectors define a language, so there can be no misclassification.

Often times in machine learning though, we are dealing with small, finite samples. We don't know if we have all the support vectors. In this case, errors in classification can be made. The good news is, the classification error is bounded.

The classification error, that is, the number of new samples misclassfied, equals (with probability $1 - \delta$)

$$\frac{\alpha_0}{m}\left(\frac{R^2}{\rho^2}log^2(m) + log(\frac{1}{\delta})\right)$$

where m is the size of sample, and ρ is the margin of the training set, and R is the radius of the embedding.

ρ can be considered the level of "confidence" in the hyperplane. One must be careful with this definition though, as the margin decreases with the sample size. So the margin with 2 samples will be larger than one with 100, but the one with 100 will perform better on new data.

R is defined to be the radius of the embedding. More specifically, R is the number of subsequences in the string in the training sample with the most subsequences. The quantity of interest we are often concerned with is $\frac{R^2}{\rho^2}$. This is often what we mean when we talk about margin bounds. The reason R is needed is because without it, the margin bound could be made artifically small by just blowing up the radius.

2.8 Complexity Comparison

Originally we started the paper discussing two different ways to learn a language from examples. Now that the second method has been explained in more detail, lets compare the complexity of the two approaches. Here we will examine in slightly more detail how support vector classification exactly proceeds. We will first only examine complexity in the context of piecewise testable languages.

Let m be the number of samples.

First compute the graham matrix of the sample. The graham matrix just applies the kernel K to every pair of strings in the sample. This will do m^2 computations. We can bound an individual computation using the longest string in the sample. This is just $|x_{\max}|^2|\Sigma|^2$. So the total time for this step is $(m^2)|x_{\max}|^2|\Sigma|^2$

Then we input the graham matrix along with the labels of the samples into our support vector classifier. This solves a quadratic programmming optimization problem to determine the values of α. It runs in time $O(m^3)$. The total running time of our classification procedure is then $m^3 + (m^2)|x_{\max}|^2|\Sigma^2| = O(m^3)$

In conclusion, we know that constructing minimally consistent DFA is NP-Complete, but learning a separating hyperplane is $O(QP(m)+m^2*|x_{max}|^2|\Sigma|) = O(m^3)$. The seems like a nice improvement.

Unfortunately, as previously stated, these results apply only to learning Piecewise-Testable languages, and in reality the computational complexity of separating any language is directly related to the complexity of computing the kernel that separates it. Let K_n be the kernel that separates the regular languages, and $C(K_n)$ be the complexity of computating this kernel. Then the complexity of learning any general regular language is $O(QP(m) + m^2 * C(K_n))$.

Sadly, the best known algorithm for K_n takes exponential time. Another student, Jeremiah Blochi, is working on the problem of determiningg the complexity of K_n. More information can be found in his paper.

3 My Work

So as previously stated, my goal was to see how various classification parameters (algebraic margin ρ and number of support vectors) vary with the number of states in the DFA that generated the language L. To do this, I first attempted to do some empirical testing using matlab scripts of my own combined with support vector classification and pt-dfa generating algorithms given to me by Leo.

3.1 Empirical Testing

3.1.1 Testing the error

My first task of empirical exploration was to explore how the error varied based on number of samples in the training set. I did this both to confirm the theoretical results, and to help myself more fully understand support vector classification.

The basic testing methodology was as follows

1. Generate a random PT (Piecewise-Testable) language. This will produce both the machinery needed to generate strings in the language, and give the number of states in the DFA that accepts the language.

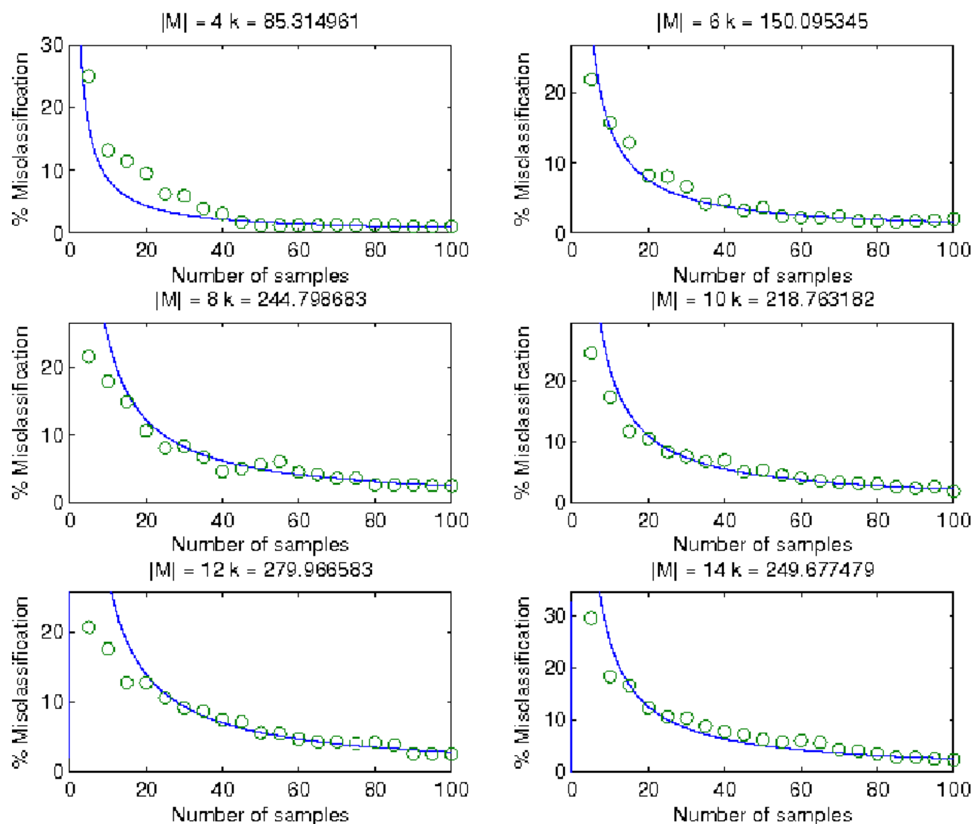
2. Generate a sparse set of both positive and negative samples from the PT language. A sparse set better tests the margin and error percentage as it does not give the classifier every string up to some length n (A dense set). It also mimics a more realistic input and so shows the true performance of the classifier more reliably.

3. Generate a new sparse set. This will be the test set.

4. Train the classifier with a subset of the training set.

5. Test the percentage of misclassification using the entire test set. Repeat step 4/5 with a larger subset

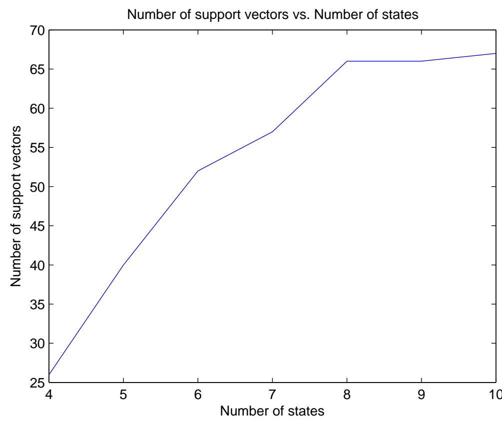
Here is the results of testing the error bound. The data clearly exhibits a $1/m$ shape. Furthermore classification constant k doesn't appear to be growing particularly fast. It may even be growing at a linear or logarithmic rate.



3.1.2 Margin ρ and Number of Support Vectors

For the margin ρ and the number of support vectors, I essentially did the same procedure as with testing the error. Except this time I took many more test machines and attempted to find the worse case behavior across many DFAs of m states. I would essentially generate n DFAs with m states, and record the smallest margin and the largest number of support vectors across those state samples.

The question of number of support vectors vs number of states seemed to yield decent results with just looking at the worst case of many random dfas. Here is a matlab plot of these results.



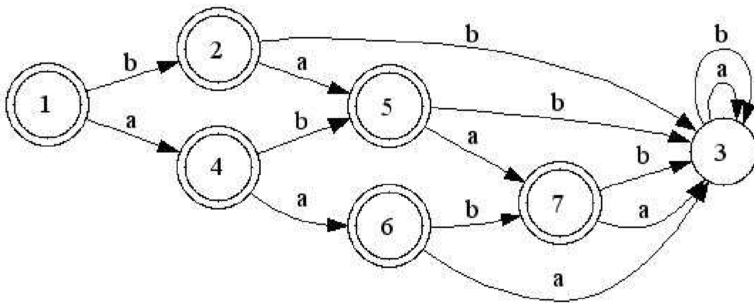
You'll note that the growth rate seems to decrease rapidly with number of states.

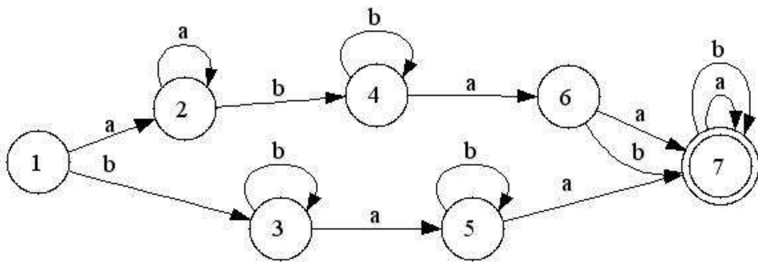
Unfortunately I could not get a similar good result with the margin. It may be because of an incorrect definition. Surprisingly, the margin is characterized differently depending on the literature read, so getting a precise definition was harder than imagined, especially taking into account hyperplane offsets. I think I now have the correct definition, but unfortunately time did not permit getting good data before this report was due.

3.2 Examining the DFA structure

Unfortunately, initial experimentation with varying the number of states in a random piecewise testable language did not seem to give vary good results. The main problem seems to be that number of states is not a solid indicator of number of support vectors or the separating margins. In one set of DFAs with number of states = 7, the number of support vectors varied from 6 to 42! This led me to start considering what specifically about the structure of a DFA makes their languages hard to learn.

Here are the two DFAs corresponding the 6 support vectors (top) and 42 support vectors (bottom).





As can be clearly seen, they are quite different. The first one accepts only a small finite number of strings. The second one accepts an infinite number. Furthermore all the strings accepted by the first machine are accepted immediately in the beginning. No reject states are traversed on the way to an accept state. In the second machines, ONLY reject states are traversed in the path to an accept state. Furthermore in the last machine, the sink state is an accept state, while in the first one the sink state is a reject state. Finally the first machine has no self loops (except in the sink state), while the second machine has many. Note that self loops are the only way to obtain an infinite language in a PT dfa, as there cannot be transitions backwards. So this is really a natural consequence of the cardinality of the strings in the language.

4 Conclusions

Progress was certainly made, but it seems like we must examine a much larger range of machines in terms of number of states. Unfortunately examining machines with a large number of states requires more strings, and obtaining data is a very slow process.

So for the immediate future, I plan on examining what about the structure of DFAs makes their languages hard to learn. I feel this may be more interesting than just looking at the number of states. Then, using this difficult structure, I'll try to construct a PT DFA with n states with a similar structure. Finally I'll reexamine the number of states question.

5 References

1. Leonid Kontorovich, Corinna Cortes, Mehryar Mohri. Learning Languages From Examples
2. Shai Ben-David, Nadav Eiron, Hans Ulrich Simon. Limitations of Learning Via Embeddings in Euclidean Half Spaces.
3. Dana Angluin. On the complexity of minimum inference of regular sets.
4. E. Mark Gold. Complexity of automaton identification from given data.
5. Leonid Kontorovich. Talk on Kernel Methods for Learning Languages.