CS 710: Complexity Theory 5/4/2010

Lecture 29: Computational Learning Theory

Instructor: Dieter van Melkebeek Scribe: Dmitri Svetlov and Jake Rosin

Today we will provide a brief introduction to computational learning theory, a subject that could easily be the focus of an entire course. We will begin by more precisely defining the learning setting, wherein the goal is to learn a process through observing some examples of it and generating a model for the process that could subsequently be used to predict its outcome in future instances. A common example of such learners is the adaptable spam filter, which attempts to learn the process by which human readers distinguish between spam and legitimate mail, with the ability to account for individual preferences and to evolve over time as spam also changes.

We will then give a general algorithm for learning in a certain context and discuss decision lists (and decision trees generally) in relation to that algorithm. Finally, for another, more restricted setting we use techniques of harmonic analysis to construct a learner and discuss some examples of processes it can learn.

1 Concept Learning

The notion we will develop is that of *concept learning* –the learning of a predicate so as to be able to classify objects based on the truth value of the predicate on a given subject. We can thus conceptualize this as a mapping c from $\{0,1\}^n$ to $\{0,1\}$ for inputs of length n and using a single bit to represent the truth or falsehood of the predicate. For our example above, we can think of an input message as a string of text of length n and an output of 1 as indicating that the message is believed to be spam. Note that this formulation is essentially analogous to the problem of deciding membership in a language in computational complexity theory.

To precisely define a learning problem, we will need the following ingredients:

1. A *concept class* C capturing both the nature of the function to be learned and all *a priori* information known about the class, its restrictions, *etc.* Also, C will be the set of all learnable functions for the problem, with $c \in C$. Typically we will partition and index this class by n and s as follows.

   Definition 1. *$C_{n,s}$ is the partition of the concept class C for inputs of size n and implementations size-bounded by s (e.g., predicates computable by circuits of size no larger than s).*

2. An *observation phase* during which the learner acquires information in attempting to learn the model. Various ways of observation are typically categorized as follows.

   - *Passive learning.* In this setting (the one that typically applies), the learner makes queries to a *sample oracle* to obtain a set of samples $S(c, D)$ from some underlying distribution D on $\{0,1\}^n$. This distribution need not be known to the learner; if it is unknown the learning is said to be "distribution-free," although in general a learning algorithm should work correctly for any D. The samples may be given to the learner either

labeled or unlabeled; if they are labeled, this is called *supervised learning*. Unlabeled samples lead to *unsupervised learning* and can give information about the underlying distribution.

- *Active learning*. In this setting the learner can actively ask questions of its teacher. Generally this is in the form of *membership queries*, which allow the learner to obtain the labels for specific inputs x of its choosing. More powerfully, the learner can ultimately make *equivalence queries*: after having formed a hypothesis about the concept to be learned, it asks an oracle whether this hypothesis is exactly correct. The oracle then responds either by confirming equivalence or providing a specific counterexample misclassified by the hypothesis.

3. A *hypothesis class* H, likewise partitioned according to n and s, representing the set of all possible predicates the learner could output. In *proper learning*, we have the ideal case where $H = C$. However, we typically settle for *prediction learning*, where we allow H to be the set of all predicates; in this case it is only important that the learned predicate accurately predicts the behavior of c, not that it is equivalent to the concept in any deeper sense. These are, of course, two extremes, and generally we have $C \subsetneq H \subsetneq P$, where P is the set of all predicates. We might choose to do this simply for efficiency; learning is usually easier when $H \supsetneq C$. On the other hand, we occasionally perform *agnostic learning*, when we have no *a priori* information about the concept; then we have $C = P$ and we must try to approximate C by as restrictive an H as we can.

4. A *quality measure*. We choose the *error* of a learning algorithm as the metric of its success.

   Definition 2. *The* error ε *of a learner equals* $\Pr_{x \leftarrow D}[h(x) \neq c(x)]$.

   Here h represents the specific hypothesis chosen by the learner from the set H. When we desire *exact learning*, we require that $h = c$ everywhere or alternatively that $\varepsilon = 0$. This usually requires equivalence queries except in the most simple settings. We could also use the number of prediction mistakes made by the learner as a success metric; the goal is to minimize the number of times the learner misclassifies the next sample from D.

5. A *complexity measure*. An information-theoretic measure is the number of samples necessary to bring the error below a given ε, for instance. A complexity-theoretic measure is the running time of the learning algorithm, taking into account the number of samples used and the efficiency of their processing.

2 Distribution-Free Learning

As mentioned above, we would like our learning algorithm and its successful operation to be independent of the underlying distribution chosen by the sampling oracle. Also, given that only in a few cases can we actually manage exact learning, it generally suffices that the produced hypothesis is mostly correct most of the time, *i.e.* that is probabilistically approximately correct. We formalize these goals and the kinds of concepts for which we can achieve them as follows.

Definition 3. *A concept class C is* PAC-learnable *(probabilistically approximately correctly) by some hypothesis class H if there exists a learning algorithm $L(n, s, \delta, \varepsilon)$ such that* $(\forall n, s, \delta > 0, \varepsilon >$

0)$(\forall c \in C_{n,s})(\forall D$ *on* $\{0,1\}^n)$ L *produces a hypothesis* $h \in H_{n,s}$ *such that with probability at least* $1-\delta$ *the error is less than* ε *when* L *learns with labeled samples from* D. *Further, this process is* efficient *if the running time of* L *is* $\mathrm{poly}(n, s, \frac{1}{\delta}, \frac{1}{\varepsilon})$.

2.1 Decision Lists

A simple but nevertheless very common example of a PAC-learnable class is the set of concepts that can be predicted by a *decision list*. Decision lists are a subset of binary decision trees (which we have encountered earlier in this course) wherein at each node, at least one response to the corresponding query leads to a conclusive answer; *i.e.*, the tree will make a definitive classification for at least one value of the queried bit. Figure 1 below gives an example of such a list.

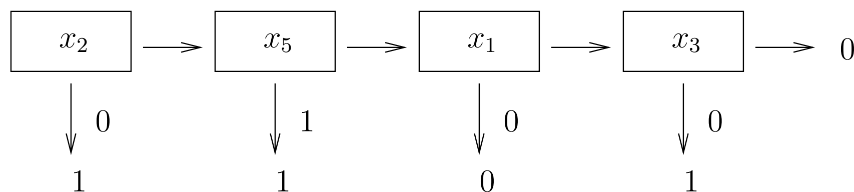

Figure 1: A decision list that queries at most four bits of its input.

The length of a decision list is bounded by the length of its input; while it may query the bits in any order, it cannot query any bit more than once. Upon querying any bit at a non-terminal step, the list may end the computation on either a 0 or a 1 and output a result of either 0 or 1. At the final step, the list may either output the value of the final bit queried or output its complement. Therefore we find that $|C_n| \leq n! \cdot 4^n$.

We also have a simple proper learning algorithm for this concept class: having seen m samples from the distribution, output a decision list that is consistent with them (if such a consistent list exists). How exactly to do this we leave as an exercise, but we will find a bound for m.

Theorem 1. *Decision lists are PAC-learnable from* m *sample queries, with* $m = \mathrm{poly}(n, \log\frac{1}{\delta}, \frac{1}{\varepsilon})$.

Proof. We want to ensure that our learning algorithm L outputs bad hypotheses with low probability. A bad hypothesis $h \in H_n$ is one such that $\Pr_{x \leftarrow D}[c(x) \neq h(x)] \geq \varepsilon$. In this analysis we fix h' to be a specific bad hypothesis. We then obtain

$$\Pr[L \text{ outputs this } h'] \leq \Pr[\text{all } m \text{ samples are consistent with } h'] \leq (1-\varepsilon)^m. \tag{1}$$

The second inequality follows from our definition of h': the m samples are drawn independently according to D, and on each sample, h' differs from c with probability at least ε.

Taking the union of all such possible bad hypotheses, we obtain in Equation (2) an upper bound for failure, which we wish to be at most δ.

$$\Pr[L \text{ fails}] \leq |H_n| \cdot (1-\varepsilon)^m < \delta. \tag{2}$$

Thus it suffices to find $m \geq \frac{1}{\varepsilon}(\log|H_n| + \log\frac{1}{\delta})$. Since $\log|H_n| = O(n \cdot \log n)$, $m = \mathrm{poly}(n, \log\frac{1}{\delta}, \frac{1}{\varepsilon})$. Therefore, because (as the reader can show) processing each individual sample can be done efficiently, this upper bound on m demonstrates that L is an efficient PAC-learning algorithm. □

We point out that a general PAC-learning algorithm exists for any PAC-learnable class that is simply a generalization of the above algorithm. In the next subsection, we define a quantity that will allow us to improve the number of samples used by the decision-list algorithm and generalize it to other concept classes.

2.2 VC-Dimension

Definition 4. *$P_{H_n}(k)$ is the maximum over all sets of unlabeled samples $(\xi_1, \ldots, \xi_k)$ of the number of different vectors of the form $(h(\xi_1), \ldots, h(\xi_k))$ with $h \in H_n$.*

In other words, $P_{H_n}(k)$ is the number of hs which form distinct characteristic vectors over a set of k samples, maximized over all such sets. Using this we define the VC-dimension [1] of hypothesis class H_n:

$$\text{VC-dim}(H_n) = \max_m (P_{H_n}(m) = 2^m) \quad (3)$$

In words, this quantity is the largest m for which all possible characteristic vectors of length m may be realized by choosing the appropriate hypotheses from H_n. This quantity may be small even if $|H_n|$ is large. As an example we consider perceptrons.

A perceptron is a linear threshold function within d-dimensional real space, outputting a 1 if some linear combination of the inputs is at least some threshold value, and 0 otherwise: $\sum_{i=1}^{d} a_i x_i \geq t$. The VC-dimension of this class is the maximum number of points for which all possible settings may by classified appropriately. For perceptrons $\text{VC-dim}(H_n) = d + 1$; see Figure 2 for details for the case $d = 2$. As we will see a bit later, this means that while in principle the number of patterns from k samples is bounded by 2^k, in fact this bound is a polynomial in k of degree $d + 1$.

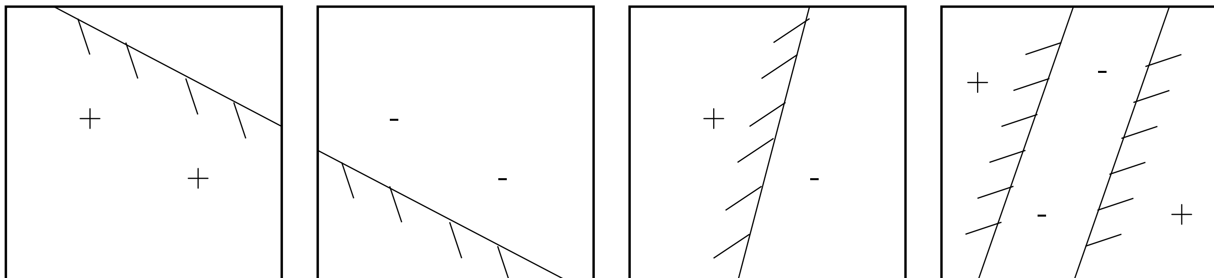

Figure 2: For $m = 2$, any possible setting may be realized by a single linear boundary. Convince yourself that this is true for $m = 3$. For $m = 4$, the setting on the right cannot be classified by a single line (we leave this as an exercise). Thus VC-dim(perceptrons in $\mathbb{R}^2$)=3.

2.3 Improving PAC-Learning for Decision Lists

We showed in 2.1 that using $m = \frac{1}{\varepsilon}(\log |H_n| + \log \frac{1}{\delta})$ is sufficient for PAC-learning the concept class of decision lists. This bound comes from Equation 2 , where the $|H_n|$ term is present as an upper bound on the number of distinct h that could be output using m samples. We can lower the number of samples needed by getting a tighter bound for the latter quantity. It turns out that this

[1]VC stands for Vapnik-Chervonenkis, so the letters VC have no particular meaning.

quantity is upper bounded by $P_{H_n}(2m)$, a fact we do not prove here. Given this unproven fact, our goal is to upper bound $P_{H_n}(2m)$, and then we can replace the $|H_n|$ term above with this value.

Obviously the maximum possible value of $P_{H_n}(2m)$ is 2^{2m}, but depending on the hypothesis class the value may be much smaller. The argument that $P_{H_n}(2m)$ bounds the number of nonequivalent hypotheses over a sample size of m is non-trivial, but it should be obvious that this number is an improvement over $|H_n|$. $P_{H_n}(2m)$ can be bounded by the VC-dimension of the hypothesis class, VC-dim(H_n) defined above. Specifically

$$P_{H_n}(k) \leq \sum_{c=0}^{d} \binom{k}{c} \leq \left(\frac{ek}{d}\right)^d \text{ where } d = \text{VC-dim}(H_n). \tag{4}$$

The log of this term is $d \log k$, which replaces $\log |H_n|$ in the value of m above.

2.4 A General PAC-Learning Algorithm

Given the analysis above, we can give a general PAC-learning algorithm. Recall that the learning algorithm for decision lists was to obtain a number of samples and output a hypothesis consistent with those samples. The same analysis given there applied to the general case shows that $m = O(\frac{1}{\varepsilon}(\text{VC-dim}(H_n) \log \frac{1}{\varepsilon} + \log \frac{1}{\delta}))$ samples suffice. The general PAC-learning algorithm, then, is as follows.

1. Query the sample oracle m times.
2. Output a hypothesis $h \in H_n$ consistent with those samples.

At the same time, a lower bound, which we do not prove here, is known which shows that $\Omega(\frac{1}{\varepsilon}(\text{VC-dim}(H_n) + \log \frac{1}{\delta})$ samples are needed for this algorithm. This shows that the general algorithm given above is essentially tight with respect to the number of samples needed.

We point out that this PAC-learning algorithm assumes that finding a consistent hypothesis can be done efficiently. This may not always be the case, as described in the next section.

2.5 Complexity of the Consistency Problem

Up to now we have argued from an information-theoretic perspective. Complexity-theoretic arguments deal with Step 2 above - finding a consistent hypothesis.

If P = NP and H is polytime computable, the consistency problem is trivial (guess the hypothesis and verify that it is consistent). We can also show that for some simple examples the consistency problem can be NP-hard, depending on the choice of the hypothesis class. Finding a consistent hypothesis for $H = C = \{$DNF formulas with ≤ 3 clauses (i.e., disjunctions of at most three conjunctions, with the conjunctions being unbounded in size)$\}$ is NP-hard, for example. Using distributivity such a formula can be expanded into 3-CNF form; the relaxed version with $H = \{$3-CNF formulas$\}$ is easy. The important point here is that for some concept classes proper-learning is difficult, but learning the same concept class with a somewhat larger hypothesis class H can be much easier.

As mentioned last lecture, if one-way functions exist then we can't efficiently PAC-learn $C_{n,s} = \{$circuits of size $\leq s\}$ by any H. This follows from the fact that if one-way functions exist we can construct pseudorandom bit generators, and pseudorandom function generators. Given a pseudorandom function generator as the concept c, the learner must construct a hypothesis which predicts

the output of c on the next sample. By the definition of pseudorandom generators this cannot be done by computationally limited processes.

2.6 Notes on PAC-learning

- In our definition we required our algorithm to work for any choice of (δ, ε). As we saw with randomized algorithms we can relax restrictions on error rate and boost the result through multiple applications of the algorithm. The same is true for PAC-learning; given the following settings of δ and ε we can reach any arbitrary degree of confidence and accuracy.

$$\delta = 1 - \frac{1}{poly(n)} \tag{5}$$

$$\varepsilon = \frac{1}{2} - \frac{1}{poly(n)} \tag{6}$$

  Confidence can be boosted in a similar way that was used for error reduction with randomized algorithms: by generating a large number of hypotheses h, then testing each on samples taken from the distribution, estimating the accuracy of each using the Chernoff bound, and choosing the best.

- Reducing the error of the algorithm does not follow from a simple use of the techniques for error reduction of randomized algorithms, but with more work we can also reduce the error of the learning algorithm. The key intuition is that our PAC-learning algorithm is required to perform correctly for all distributions D, not just the one being learned. We can exploit this by taking our hypothesis h and re-weighting the distribution to place equal weight on the inputs for which h performs correctly and those for which it fails. Running the PAC-algorithm on this new distribution will provide new information. After enough iterations of this procedure we output a weighted majority vote of all the hypotheses. Arguing that this boosting procedure works is non-trivial.

- In an agnostic learning setting (one with no underlying C) there is no guarantee that the concept c exists within our hypothesis class H. The best we can hope for is to get a hypothesis as close to c as possible within H, with some margin of error. We look for error at most ε plus the minimum distance between H and c.

  The generalized PAC-learning algorithm given in 2.2 may be used in this setting with one modification: rather than output h consistent with all m samples, we output an h which is as consistent as possible given H. Finding this consistent hypothesis becomes much more complicated, even for simple problems. For example with $H = \{\text{conjunctions}\}$ the consistency problem is NP-hard.

- One unrealistic assumption made by our PAC algorithm is that the samples received from the oracle are completely error-free. Errors could occur on the labels or the inputs themselves (which may be inconsistent with D), and may occur due to noise, or maliciousness on the part of some attacker. The simplest error-aware model to which PAC-learning work has been extended is one which allows random classification noise. Formally we assume that for every possible input, the label is flipped with probability $\zeta < \frac{1}{2}$ independently for each input.

Clearly the closer ζ is to $\frac{1}{2}$ the more difficult the learning problem becomes. The running time of an efficient PAC-learner becomes $\text{poly}(n, s, \frac{1}{\varepsilon}, \frac{1}{\delta}, \frac{1}{\frac{1}{2}-\zeta})$.

Statistical query algorithms are a class of algorithms which work in this setting. Rather than query a sample oracle, statistical query algorithms ask the teacher for an approximation for the probability that a certain predicate holds with respect to D. For example, one valid query asks for the probability over D that the label is the parity of the input bits.

Many learning algorithms can be cast in this framework; any algorithm that can will be robust in a setting with random classification noise.

3 Learning with respect to the Uniform Distribution

For every PAC-learnable class, any PAC-learning algorithm will function under any possible distribution D. Certainly this is not reasonable for all problems, but we can typically devise algorithms that do function with respect to a particular distribution –a natural choice is the uniform distribution. Here again we may make use of the techniques of harmonic analysis, which is helpful for learning concepts whose Fourier transform is concentrated on a few coefficients. By Parseval's inequality, the sum of the squares of the Fourier transform coefficients of a Boolean function is 1. It may be that the 2^n coefficients have uniform values, or there may be a few large ones; in this case, there is a learning algorithm that works with respect to the uniform distribution.

This holds for decision trees, for example, and for constant-depth circuits. Recall that earlier in the course we proved that such circuits can be approximated well by low-degree multivariate polynomials. Also, the Fourier transform corresponds to expressing the function as a linear combination of characters that are basically parity functions over certain subsets (of inputs). These parity functions can be made products by choosing $\{-1, 1\}$ rather than $\{0, 1\}$ as the bit settings. So constant-depth circuits can be approximated well by linear combinations of characters corresponding to small subsets.

We will now see how to efficiently learn with respect to the uniform distribution in these cases by making use of membership queries.

3.1 A Learning Algorithm using List Decoding

Suppose our concept class C has a power spectrum that is concentrated over a few coefficients. It turns out that the list-decoding algorithm for the Hadamard code can be used as a component in a learning algorithm L for C. Let us view the concept c as a function $f : \{\pm 1\}^n \to \{\pm 1\}$. Recall that when applied to the characteristic sequence of f, the Hadamard list-decoding procedure gives a list of all information words with a Hadamard encoding within a distance $1/2 - \epsilon$ of f, and it does this in $\text{poly}(n, 1/epsilon)$ time. Since the characteristic sequence of χ_g equals the Hadamard encoding of g, we have a list of characters χ_g that with high probability includes all those characters such that $\Pr_x[\chi_g(x) = f(x)] \geq 1/2 + \epsilon$. Using the equality $\hat{f}(g) = 2\Pr[f(x) = \chi_g(x)] - 1$, we then obtain a bound on the size of the Fourier coefficients and conclude that our list of characters includes all those such that $\hat{f}(g) \geq 2\epsilon$. So we know that given some threshold τ, all Fourier coefficients with absolute value at least τ can be found in time $\text{poly}(n, \frac{1}{\tau})$.

Note that using the list-decoding algorithm to determine the large-weight coefficients requires evaluating the received word at specific points we choose. Therefore, the learning algorithm will need to be able to make membership queries. We now present the learning algorithm L for f.

(1) Generate l, a list of indices of all Fourier coefficients whose absolute value is at least τ using the Hadamard list-decoding algorithm, with the characteristic sequence of f as the "received word."

(2) For each $y \in l$, estimate $\hat{f}(g) = 2\Pr[f(x) = \chi_g(x)] - 1$ by picking a polynomial number of points at random, and let α_y be this approximation. Ensure that with high probability it is within η of the true value.

(3) Recall that

$$f(x) = \sum_y \hat{f}(y)\chi_y(x). \tag{7}$$

We then define an approximation g of f as

$$g(x) = \sum_{y \in l} \alpha_y \chi_y(x). \tag{8}$$

We consider this an approximation since we have dropped some of the small coefficients of f and approximated the others. If, as we have assumed, most of the weight is concentrated on a few large coefficients, this will be a good approximation.

(4) Output $h(x) = \text{sign}(g(x))$. We do this since g works over the set of real numbers, and so might not be a Boolean function, but h is.

Now it only remains to find a reasonable setting for τ and to show that the algorithm will succeed with a certain probability. Since τ determines the running time of the algorithm, the complexity of it also depends on τ, so we find an implicit upper bound on the length of the list l.

So consider the inputs x for which $h(x) \neq f(x)$. Then the difference between $g(x)$ and $f(x)$ is at least one. We square this difference to obtain an absolute value and then apply Markov's inequality to show that

$$\Pr[h(x) \neq f(x)] \leq \Pr[(g(x) - f(x))^2 \geq 1] \leq E_x[(g(x) - f(x))^2]. \tag{9}$$

We then use Parseval's inequality and the linearity of the Fourier transform to find that

$$\Pr[h(x) \neq f(x)] \leq E_x[(g(x) - f(x))^2] = \sum_y \left(g(\widehat{y) - f}(y)\right)^2 = \sum_y (\hat{g}(y) - \hat{f}(y))^2. \tag{10}$$

Now we divide the last sum between those in l and those that outside it. For those in l the coefficient is α_y, a good approximation to $\hat{f}(y)$ to within η. For those outside, $\hat{g}(y)$ is zero. So we have

$$\Pr[h(x) \neq f(x)] \leq \eta^2 |L| + \sum_{y \notin L} (\hat{f}(y))^2. \tag{11}$$

Since this depends only on f, we can make use of the concentration of the power spectrum to set τ appropriately so as to make that term small.

As mentioned above, similar PAC-learning algorithms can be created for decision trees and for constant-depth circuits. We present these as exercises for the reader, but also work out analyses for them below.

3.2 Exercise 1: Decision Trees

Consider decision trees as a specific example. We will need a result relating the Fourier spectrum of a function f and its representation as a decision tree.

Exercise 1. *If f is computed by a decision tree T of size s, then $\sum_y |\hat{f}(y)| \leq$ #leaves of $T \leq s$.*

Now suppose we apply the learning algorithm using list-decoding described above. We first want to bound the error term of $\sum_{y \notin L} (\hat{f}(y))^2$. Since we know that $|\hat{f}(y)| \leq \tau$ for all $y \notin L$, the exercise implies the following:

$$\sum_{y \notin L} (\hat{f}(y))^2 \leq \tau \cdot \sum_{y \notin L} |\hat{f}(y)| \leq \tau \cdot s. \tag{12}$$

To have error at most ε, we want to set τ and η so that $\tau \cdot s \leq \frac{\varepsilon}{2}$ and $\eta^2 |L| \leq \frac{\varepsilon}{2}$. We achieve the former by setting $\tau = \frac{\varepsilon}{2s}$. From this we know that $|L|$ is polynomial in n, s and $\frac{1}{\varepsilon}$. To ensure the latter, we set $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$. Recall that η is the maximum error we want to allow on the approximations we calculated for the Fourier coefficients. Using a Chernoff bound, we can achieve $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$ with $\text{poly}(|L|, \frac{1}{\varepsilon})$ samples.

We conclude that the algorithm, which uses membership queries to run the list-decoding algorithm, has error at most ε and runs in $\text{poly}(n, s, \frac{1}{\varepsilon})$ time.

3.3 Exercise 2: Constant-Depth Circuits

Constant-depth circuits also have the property that their Fourier spectrum is concentrated on a small number of coefficients. This will allow us to make use of the above analysis to give a learning algorithm for constant-depth circuits. The key difference is that we will not need to use the list-decoding algorithm to generate the list containing large coefficients, so the learning algorithm will only require samples and not membership queries. The intuition is that because constant-depth circuits cannot even approximate parity, the characters with large Hamming weight must have small coefficients (otherwise, the function would have high agreement with parity over the bits indexed by that character). The list of coefficients with large weight, then, will just be the list of coefficients corresponding to characters with small Hamming weight.

We now give the analysis. Let f be a Boolean function on n variables computed by a depth d unbounded fanin circuit of size s. In the lecture on constant-depth circuits, we constructed a low-degree approximation of f as a step towards proving circuit lower bounds for parity. In particular, we constructed a Boolean function g' computed by a polynomial over $GF(3)$ of degree $\Delta \leq (2t)^d$ such that

$$\Pr[f(x) \neq g'(x)] \leq \frac{s}{3^t}.$$

This implies that

$$E_x[(f(x) - g'(x))^2] \leq 4\frac{s}{3^t}, \tag{13}$$

which is in a form that is useful for applying Fourier analysis. But to use Fourier analysis, we want a polynomial over $\mathbb{R}$ rather than $GF(3)$. The polynomial g' was constructed using the approximation method. Using the switching lemma method instead, g' can be constructed with the same properties mentioned above but over $\mathbb{R}$. Let L be the set of binary y's with Hamming weight $\Delta \leq (2t)^d$. This

is our list of characters with large coefficients, so we want to show that characters outside of L have small coefficients. We have the following

$$\sum_{y \notin L} (\hat{f}(y))^2 = \sum_{y \notin L} (\hat{f}(y) - \widehat{g'}(y))^2 \leq \sum_{y} (\hat{f}(y) - \widehat{g'}(y))^2, \tag{14}$$

where $\widehat{g'}(y) = 0$ for $y \notin L$ because g' can be expressed as a polynomial over $\mathbb{R}$ of degree at most $(2t)^d$. Now (14) is in a form we have seen earlier, so using (13) we get

$$\sum_{y} (\hat{f}(y) - \widehat{g'}(y))^2 = E_x[(f(x) - g'(x))] \leq \frac{4s}{3^t}.$$

We ensure this is at most $\frac{\varepsilon}{2}$ by setting $t = \Omega(\log \frac{s}{\varepsilon})$. Setting $\eta = \sqrt{\frac{\varepsilon}{2|L|}}$ implies that the RHS of (11) is at most ε.

We have given the analysis, so we recap the learning algorithm and see how efficient it is. We set $t = \Theta(\log \frac{s}{\varepsilon})$ and let L be the set of characters with a Hamming weight at most $\Delta = (2t)^d = \Theta((\log \frac{s}{\varepsilon})^d)$. Once we have L, we proceed from the second step of the learning algorithm in Section 3. The analysis there and here shows that we get an approximation with error at most ε. The running time and number of samples needed by the algorithm are poly$(|L|, n, \frac{1}{\varepsilon})$. As $|L| = \sum_{d=0}^{\Delta} \binom{n}{d} \approx n^{\Delta}$, the running time of the learning algorithm is quasi-polynomial for polynomial size constant-depth circuits. However, recall that the algorithm we have given only requires samples and not membership queries.

For the specific case of depth $d = 2$, meaning circuits of DNF or CNF form, there does exist a polynomial time algorithm for learning with respect to the uniform distribution which uses membership queries and Fourier analysis.

4 Next lecture

In the next and final lecture, we briefly introduce quantum computing, defining the quantum model as a variant of the probabilistic model, presenting a common technique in the design of quantum algorithms, and giving some currently known upper bounds on the power of the quantum model.