CS 810: Complexity Theory 4/16/2007

Lecture 28: Computational Learning Theory

Instructor: Dieter van Melkebeek Scribe: Jake Rosin

Today's lecture discusses computational learning theory. This topic is expansive enough to fill an entire course, so this lecture will provide only an overview.

In this setting the goal is to learn some process through observation, generate a model for that process, and use the model to predict the process's future behavior. An example of this is a spam filter tuned by a user, which attempts to learn the process by which a human distinguishes between spam and legitimate mail.

We give a general algorithm for learning under certain conditions, and discuss decision lists as an application of that algorithm. We also use harmonic analysis to construct a learner which functions in a more restricted setting, and give example processes which it can learn.

1 Concept Learning

Our focus will be on concept learning: learning a certain predicate. Continuing our example from above, our filter constructs a predicate for whether a message is considered spam. A concept in the Boolean domain maps inputs to a single bit of output: $c : \{0,1\}^n \longrightarrow \{0,1\}$.

We need certain ingredients to form a learning problem.

1. A *concept class* C representing a set of concepts. C is the set of possible functions to be learned, with $c \in C$. Often the class will be partitioned and indexed by n and sometimes s.

   Definition 1. *$C_{n,s}$ represents the partition of concept class C for inputs of size n with implementations size-bounded by s (for example, predicates computed by circuits with size $\leq s$).*

   One variant of concept learning is agnostic learning, where there is no underlying concept class (or equivalently where C is the set of all predicates).

2. The process by which the learning algorithm learns the model. It is often assumed that the learner is a passive observer. In this context the learner makes a series of queries to a *sample oracle*; for each query the oracle selects an example at random from the domain according to some distribution D, which may or may not be known to the learner. In the case that D is unknown this is called "distribution-free" learning; in either case a general learning algorithm should work for any D.

   The examples are given to the learner either labeled or unlabeled. Unlabeled samples provide information about the distribution.

   In some cases the learner must make *membership queries.* These queries allow the learner to choose the x to be sampled; equivalently the learner asks an oracle for the label of x.

   Equivalence queries may also be used. Once the learner forms a hypothesis about the concept being learned it forms an equivalence query; the query is answered with either "yes" or a counterexample which the hypothesis misclassifies.

3. A hypothesis class H represents the class of predicates which the learning algorithm may output. It may be partitioned and indexed as $H_{n,s}$ as with a concept class. Proper learning is a variant of concept learning where $H = C$, which is ideal, but not always the case. Another variant is prediction learning, where H is the set of all predicates. In this context whether the learned predicate matches the concept c is irrelevant if it can accurately predict c's behavior. These two variants represent opposing extremes: often $C \subsetneq H \subsetneq \{\text{all predicates}\}$.

4. We measure the success of a learning algorithm by its *error*.

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

   When we require $\varepsilon = 0$ this is called exact learning. Achieving this goal in complex settings usually requires equivalence queries. Another measure of success is the number of errors made by the learner. During each step of the learning process the learner has some hypothesis h with which it can predict the next sample from D. Minimizing the number of times h mispredicts the next sample is the goal.

5. The final ingredient is the complexity measures used. An information-theoretic measure is the number of samples which must be used before the learner's error falls below a given ε. A complexity-theoretic measure is the running time of the learner, based both on the number of samples and the efficiency of the processing of the samples.

2 Distribution-Free Learning

Definition 3. *A concept class C is PAC-learnable (probabilistically approximately correct) by some hypothesis class H if there exists an 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 outputs with probability $\geq 1 - \delta$ a hypothesis $h \in H_{n,s}$ such that the error $\leq \varepsilon$ where L gets labeled samples from D. This process is efficient if the running time of L is poly$(n, s, \frac{1}{\varepsilon}, \frac{1}{\delta})$.*

2.1 An Example: Decision Lists

Decisions lists are a concept class which is PAC-learnable. For a given input x, a decision list queries a particular bit x_i, then either outputs 0 or 1 or queries another bit. This process may repeat no more than $|x|$ times. An example decision list is given in Figure 1.

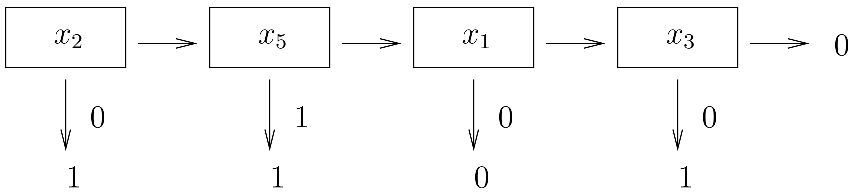

Figure 1: A decision list which queries at most 4 bits of the input.

The length of a decision list is bounded by the number of bits in the input. Each non-terminal step in the list may end the computation on either a 0 or 1, and then output either a 0 or 1. The final step may output the value of the bit being queried, or output its complement. Bits may be queried in any order. This results in $|C_n| \leq n!4^n$.

Our proper learning algorithm L is simple. It gets m samples from the oracle, then outputs a decision list that is consistent with those samples. The problem of constructing a decision list consistent with some fixed set of samples is left as an exercise; instead we find a bound for m.

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

Proof. We want to ensure that bad hypotheses are output with low probability. A bad hypothesis is an $h' \in H_n$ such that $\Pr_{x \leftarrow D}[c(x) \neq h(x)] \geq \varepsilon$. We fix h to be one such bad hypothesis.

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

The final term follows from our definition of h. h differs from the concept on any sample $x \leftarrow D$ with probability at least ε; m such samples are chosen independently.

This provides an upper bound for failure (Equation 2). We want this value to be at most δ.

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

It suffices to find $m \geq \frac{1}{\varepsilon}(\log |H_n| + \log \frac{1}{\delta})$. In this case $\log |H_n| = O(n \cdot \log n)$, so $m = \text{poly}(n, \frac{1}{\varepsilon}, \log \frac{1}{\delta})$. Since finding a consistent hypothesis can be done efficiently, this bound on m shows that L is an efficient PAC-learning algorithm.

□

We point out that the algorithm and analysis above can be applied to a number of settings. A similar algorithm and analysis works for conjunctions, disjunctions, k-CNF, k-DNF, and k-decision lists for any constant k.

2.2 VC-dimension

In this section, we define a quantity that will allow us to improve the number of samples required by the above learning algorithm. We will also be able to use the ideas of this subsection to later give a general learning algorithm.

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:

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

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: $\sum_{i=1}^{d} a_i x_i \geq$

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

t. The VC-dimension of this class is the maximum number of points for which all possible settings may by classified appropriately. For perceptrons 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 $d + 1$.

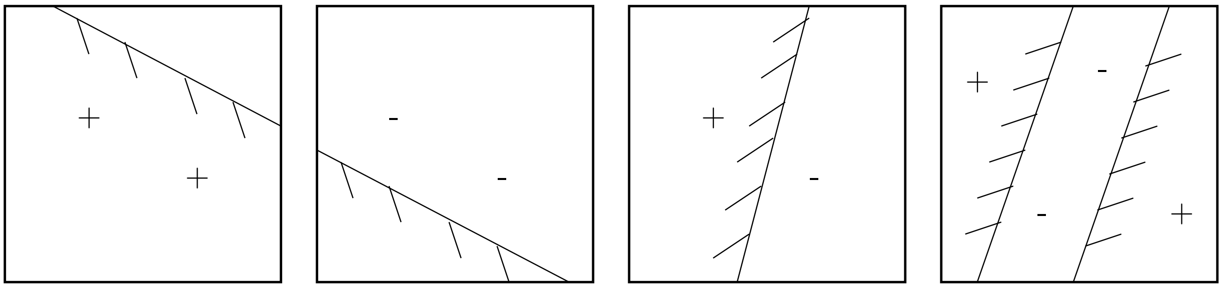

Figure 2: For $m = 2$, any possible setting may be realized by a single linear boundary. Convince yourself that this is true for $m = 3$. For $m = 4$, the setting on the right cannot be classified by a single line (we leave this as an exercise). Thus VC-dim(perceptrons in $\mathbb{R}^2$)=3.

2.3 Improving PAC-Learning for Decision Lists

We showed in subsection 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 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 subsection 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 which can will be robust in a setting with random classification noise.

3 Learning w.r.t. the Uniform Distribution

PAC-learning algorithms function under any possible distribution D. For some problems this is unrealistic, but algorithms can be devised which function with respect to some fixed distribution. The uniform distribution is a natural choice. In this context harmonic analysis turns out to be useful for learning concepts for which the Fourier transform is concentrated on very few coefficients. Recall that by Parseval's equality, the sum of the squares of the coefficients of the Fourier transform of a Boolean function is 1. The 2^n coefficients may have uniform values, but there could be a few large ones. In the latter case there exists a learning algorithm with respect to the uniform distribution.

In particular this condition holds for decision trees, as well as for constant-depth circuits. The latter assertion corresponds to the proof from earlier in the course that constant-depth circuits can be approximated well by multivariate polynomials of low degree. The Fourier transform corresponds to expressing the function as a linear combination of characters which are essentially parity functions over certain subsets. If bit settings are in $\{\pm 1\}$ rather than $\{0, 1\}$ these parity functions become products. This means that constant-depth circuits can be approximated very well by linear combinations of characters that correspond to small subsets.

We now see how we can efficiently learn with respect to the uniform distribution in such a case using membership queries.

3.1 Learning Algorithm Using List Decoding

Suppose we have a concept class C which has its power spectrum concentrated over few coefficients. We can use the list decoding algorithm for the Hadamard code as a component in a learning algorithm for C. We view the concept c as a function $f : \{\pm 1\}^n \to \{\pm 1\}$. Recall the properties of the list decoding procedure for the Hadamard code when applied to the characteristic sequence of f: in $\text{poly}(n, 1/\epsilon)$ time, we produce a list of all information words with Hadamard encoding within distance $1/2 - \epsilon$ of f. By the fact that the characteristic sequence of χ_g is equal to the Hadamard encoding of g, we have a list of characters χ_g that with high probability includes all those such that $\Pr_x[\chi_g(x) = f(x)] \geq 1/2 + \epsilon$. Plugging in the equality $\hat{f}(g) = 2\Pr[f(x) = \chi_g(x)] - 1$, we convert this inequality to a bound on the size of the Fourier coefficients, concluding that we have a list of characters that includes all those s.t. $\hat{f}(g) \geq 2\epsilon$.

We conclude that given some threshold τ, all Fourier coefficients of absolute value $\geq \tau$ can be found in time $\text{poly}(n, \frac{1}{\tau})$. We point out that using the list-decoding algorithm to determine the coefficient with large weight requires evaluating the received word at specific points of our choosing. For the learning algorithm, this means we will need membership queries. We now give the learning algorithm for f using membership queries.

1. Generate L, a list including indices of all Fourier coefficients with absolute value at least τ using the list-decoding algorithm for the Hadamard code, with the characteristic sequence of f as the "received word" for the list-decoding.

2. For each $y \in L$, estimate $\hat{f}(y) = 2\Pr_x[f(x) = \chi_y(x)] - 1$ by picking a polynomial number of points at random. Let α_y be the approximation of $\hat{f}(y)$. Ensure this approximation is within η of the true value with high probability.

3. Recall that
$$f(x) = \sum_y \hat{f}(y)\chi_y(x) \tag{7}$$
We define an approximation of f as
$$g(x) = \sum_{y \in L} \alpha_y \chi_y(x).$$
We consider g an approximation of f; we have dropped (some of) the small coefficients of f and approximated the other ones. This will be a good approximation provided most of the mass is concentrated on a few large coefficients, as we have assumed.

4. Output $h(x) = \text{sign}(g(x))$. Note that g is not necessarily a Boolean function, as it works over the set of real numbers, but h is.

What remains is finding a reasonable setting for the parameter τ, and showing that this algorithm will succeed with probability within a certain degree. We know that the complexity of this algorithm depends on τ, as τ determines the running time of the algorithm, which implicitly gives an upper bound on the length of the list L.

Suppose for some x, $h(x) \neq f(x)$. In this case, the difference between $g(x)$ and $f(x)$ is at least one. We square this to get the absolute value and apply Markov's inequality to get

$$\Pr[h(x) \neq f(x)] \leq \Pr[(g(x) - f(x))^2 \geq 1] \leq E_x[(g(x) - f(x))^2]. \tag{8}$$

Now we use Parseval's equality and get

$$\Pr[h(x) \neq f(x)] \leq E_x[(g(x) - f(x))^2] = \sum_y \widehat{(g(y) - f(y))}^2 = \sum_y (\hat{g}(y) - \hat{f}(y))^2 \tag{9}$$

where the last equality is because of linearity of the Fourier transform. We divide the sum over all ys between those in L and those not. For those in L the coefficient is α_y, a good approximation to $\hat{f}(y)$ to within η. For any $y \notin L$, $\hat{g}(y)$ is zero. This gives

$$\Pr[h(x) \neq f(x)] \leq \eta^2 |L| + \sum_{y \notin L} (\hat{f}(y))^2. \tag{10}$$

This depends only on f, and by exploiting the fact that the power spectrum is concentrated we can make that term small by setting τ appropriately.

3.2 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:

$$\sum_{y \notin L} (\hat{f}(y))^2 \leq \tau \cdot \sum_{y \notin L} |\hat{f}(y)| \leq \tau \cdot s. \tag{11}$$

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 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

$$E_x[(f(x) - g'(x))^2] \leq 4\frac{s}{3^t}, \tag{12}$$

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

$$\sum_{y \notin L} (\hat{f}(y))^2 = \sum_{y \notin L} (\hat{f}(y) - \widehat{g'}(y))^2 \leq \sum_{y} (\hat{f}(y) - \widehat{g'}(y))^2, \tag{13}$$

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 (13) is in a form we have seen earlier, so using (12) 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 (10) 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 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 3.1. 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 $\text{poly}(|L|, n, \frac{1}{\varepsilon})$. 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 lecture we discuss query complexity.