1 Last time and today

Previously:
• Started the BKW algorithm for learning parity functions in the presence of random classification noise in time $2^{O\left(\frac{n}{\log n}\right)}$.

Today:
• Complete the proof of BKW algorithm.
• New Learning Model: Learning with Membership Query Oracles.

Relevant Readings:

  available at Avrim Blum's web page

• Vadim Lyubashevsky. "The Parity Problem in the Presence of Noise, Decoding Random Linear Codes, and the Subset Sum Problem" Proceedings of APPROX-RANDOM 2005 (for an improved algorithm for learning parity with noise, with sample size $n^{1+\varepsilon}$ and time $2^{O\left(\frac{n}{\log \log n}\right)}$)
  available at Vadim Lyubashevsky's web page

  available at Vitaly Feldman's web page
2 Learning Parity Functions in the presence of RCN

Overview and Background
The BKW algorithm efficiently learns parity functions in the presence of random classification noise under the uniform distribution. In the random classification noise (RCN) model, each example $x$ is drawn uniformly over $\{-1,1\}^n$ from a noisy oracle $E_x^\eta(f)$ which also returns label $f(x) \in \{-1,1\}$. The label is incorrect (i.e. $-f(x)$ instead of $f(x)$) with probability $0 < \eta < 0.5$.

The following observation was presented in lecture 11.

Observation 1. An example which has only the $i^{th}$ bit equal to 1 can be used to provide a good estimate of whether the target parity function includes the $i^{th}$ bit.

Let $c$ denote the target parity function, represented as an $n$-bit vector:

$c = (c_1, c_2, \ldots, c_n) \in \{0,1\}^n$

Let $e_i$ denote an example in which the $i^{th}$ bit is equal to 1 and all other bits are equal to 0. Any such vector $e_i$ (i.e. with only the $i^{th}$ bit equal to 1) will be referred to as a 'basis vector'. Since only the $i^{th}$ bit of $e_i$ is equal to 1, the parity function applied to $e_i$ reveals information about $c$:

$e_i \cdot c = (0, 0, \ldots, 1, \ldots, 0, 0) \cdot (c_1, c_2, \ldots, c_n) = c_i$

Since the correct label is by definition $f(e_i) = e_i \cdot c$, and since the label $f(e_i)$ is correct with
probability
1 – η, it follows that
Prob\[f(e_i) = c_i]\ = 1 – η > 0.5.

Observation 1 is the backbone of the BKW algorithm. Most of the algorithm involves
constructing basis vectors in a manner that is efficient and that does not introduce inordinate noise.
The basis vectors then allow the target parity to be learned in a straightforward manner.
Two lemmas were introduced and proved in lecture 11. These lemmas will be used in
demonstrating the correctness and performance of the BKW algorithm.

**Lemma 1.** Given \(l\) examples drawn from a noisy oracle \(EX^\eta(f)\), the sum \(^l\) of all \(l\) labels of the examples
is correct with probability:
\[
\frac{1}{2} + \frac{1}{2} (1 - 2\eta)^l
\]

Note that the second term, \(\frac{1}{2} (1 - 2\eta)^l\), can be considered the advantage over random guessing of
the summed examples. This advantage decreases exponentially with \(l\) since \(1 - 2\eta < 1\). Therefore
summing a small number of examples is preferred.

**Definition.** An \(i\)-sample is a set of \(n\)-bit vectors, each of which has all zeros in the final \(i\) 'blocks' and
which are distributed uniformly at random.

Consider the \(n\) bit positions of each vector (example) to be divided into \(a\) sets of \(b\) consecutive
bits, so \(n = ab\).

![Figure 1 - A vector from a 1-sample, showing all b bits in the last block equal to zero](image)

In an \(i\)-sample, every vector has all bits in the final \(i\) blocks equal to zero, and the bits in the
remaining blocks are distributed uniformly at random across all vectors in the \(i\)-sample. In other
words, blocks \((a - i + 1)\) through \(a\) contain all zeros across all vectors in the \(i\)-sample, and the bits in
blocks \(1\) through \((a – i)\) are random (distributed uniformly across all vectors).

A \(0\)-sample is defined as a set of vectors which are all distributed uniformly at random – the
final 0 blocks (i.e. no blocks) are guaranteed to contain all zeros. For example, any set of examples
drawn from a noisy oracle is a \(0\)-sample since no blocks in the examples are guaranteed to contain all
zero bits.

**Lemma 2.** Given any \(i\)-sample which includes \(s\) vectors (each segmented into \(a\) blocks of \(b\) bits), in
time \(O(sn)\) we can construct an \((i+1)\)-sample that includes at least \(s - 2^b\) vectors. Each vector in the
\((i+1)\)-sample is the sum of two vectors in the \(i\)-sample.

1 The term "sum" will mean "sum modulo 2", which is equivalent to bitwise exclusive-OR operation on binary vectors.
In other words, for all vectors in an i-sample, blocks 1 through (a - i) are not guaranteed to contain all zeros while blocks (a - i + 1) through a are guaranteed to contain all zeros. By strategically adding vectors from such a set, we can 'zero out' the bits in block (a – i), thereby creating a vector which contains all zeros in the final (i+1) blocks.

2.1 The BKW Algorithm

Below is a quick but useful sketch of the BKW algorithm, followed by a detailed description and a proof of its performance. In general, the algorithm exploits the fact that expressing a target n-bit vector as a sum of a few examples allows us to efficiently learn a bit of the parity function, and a very large data set provides a high probability that we can find vectors to sum in the manner desired.

Overview of the BKW Algorithm

1. Draw (a 2^b) examples from the noisy oracle EX^η(f).

2. Sum the (a 2^b) examples to obtain an (a-1)-sample containing 2^b vectors. Use the (a 2^b) examples (which are a 0-sample) to create a 1-sample using Lemma 2. Then use the vectors in the 1-sample to create a 2-sample, and continue until obtaining a (a-1)-sample (i.e. a set of vectors in which all blocks except the first block contain all zeros).

3. From the (a-1)-sample find the basis vector e_1 and its label, which is the sum of the labels of the vectors used to construct e_1. Since there are 2^b vectors which are uniformly distributed, and only the first b bits can be non-zero, each vector has a 1/2^b probability of being e_1. If there is no vector e_1, repeat Steps 1 and 2 until obtaining an (a-1)-sample that does include e_1.

4. Repeat Steps 1 – 3 to obtain a set of e_1 vectors and corresponding labels. The most frequently occurring label (-1 or 1) is our guess for the true label y_1.

5. Use y_1, the true label of e_1, to determine whether the target parity function includes x_1 (using Observation 1).

6. Repeat Steps 1 – 5 to obtain basis vectors e_2, … e_n and determine whether the target parity function includes x_2, …, x_n.

BKW algorithm in detail

Step 1. Draw a2^b examples from the noisy oracle EX^η(f).

We will see below that the algorithm benefit from selecting the following values:

\[ a = \frac{1}{2} \log n \; ; \; b = \frac{2n}{\log n} \]
Step 2. Sum the \(a^2b\) examples to obtain an \((a-1)\)-sample containing \(2^b\) vectors.

First, use the \(a^2b\) examples (which are a 0-sample) to create a 1-sample using Lemma 2. Also by Lemma 2, the number of vectors in the 1-sample is reduced by at most \(2^b\) (i.e. contains no less than \(a^2b - 2^b\) vectors). Then use the vectors in the 1-sample to create a 2-sample, and continue until obtaining a \((a-1)\)-sample. Since:

- we begin with \(a^2b\) vectors in the 0-sample,
- each time we reduce the number of vectors by at most \(2^b\), and
- there are \((a-1)\) reductions in transitioning from the 0-sample to the \((a-1)\)-sample,

the number of vectors in the \((a-1)\)-sample is at least:

\[
a^2b - (a-1)2^b = 2^b
\]

Step 3. From the \((a-1)\)-sample identify the basis vector \(e_1\) and its label, which is the sum of the labels of the vectors which were summed to construct \(e_1\).

The basis vector \(e_1\) has first bit equal to 1 and all other bits equal to 0. Note that for every vector in the \((a-1)\)-sample, all blocks except the first block contain all zeros while the first block contains \(b\) random bits. Therefore there are \(2^b\) possible values for any vector in this set, and each is equally likely by the definition of an \(i\)-sample. Thus each vector in the set has a \(1/2^b\) probability of being \(e_1\).

Accordingly, any vector in the set has a \((1 - 1/2^b)\) probability of \(not\) being \(e_1\). Since there are \(2^b\) vectors, the probability that \(none\) of the vectors is \(e_1\) is:

\[
\left(1 - \frac{1}{2^b}\right)^{2^b} < \frac{1}{e^b}
\]

This probability can be bounded easily. Since it is a monotonically increasing function of \(b\):

\[
\forall b, \left(1 - \frac{1}{2^b}\right) \geq \frac{1}{4}
\]

\[
\forall b, \left(1 - \frac{1}{2^b}\right) > \frac{1}{e} \approx 0.368 \quad \text{since} \quad \lim_{n \to \infty} \left(1 - \frac{1}{n}\right)^n = \frac{1}{e}
\]

for virtually all \(b\), \(\left(1 - \frac{1}{2^b}\right)^{2^b} \approx \frac{1}{e}
\]

For example, for only \(b = 6\) the above quantity approximates \(1/e\) with less than 1% relative error.

If none of the vectors in the \((a-1)\)-sample are the desired vector \(e_1\), simply repeat Steps 1 and 2 until obtaining an \((a-1)\)-sample that does include \(e_1\). Since the probability that a set does not include \(e_1\) is at worst \(1/e\), it is extremely unlikely we will need to repeat Steps 1 and 2 very many times. For example, the chance of repeating ten times without obtaining the vector \(e_1\) is

\[
\text{Prob} (\text{fail to find } e_1 \text{ 10 times}) < \left(\frac{1}{e}\right)^{10} < 5 \times 10^{-5}
\]

The vector \(e_1\) was constructed in accordance with Lemma 2 by summing examples which were drawn from the noisy oracle. We also sum the \(labels\) corresponding to these examples to obtain a label for \(e_1\).

Note that \(e_1\) is constructed by summing \(2^{a-1}\) examples. Each vector in the 1-sample is constructed from two examples. Each vector in the 2-sample is constructed from two vectors in the 1-sample, which is four examples. Since in Step 2 we performed \((a-1)\) such constructions, we sum \(2^{a-1}\)
examples to construct any vector (including $e_1$) in the (a-1)-sample.

By Lemma 1, setting $l = 2^{a-1}$, the probability that this label for $e_1$ is correct is:

$$\frac{1}{2} + \frac{1}{2}(1 - 2\eta)^{2^{a-1}} > \frac{1}{2}$$

Step 4. Repeat Steps 1 – 3 to obtain a set of $e_1$ vectors and corresponding labels. The most common label (-1 or 1) is our guess for the true label $y_1$.

Steps 1 through 3 construct a vector $e_1$ and a corresponding label. The label for $e_1$ is correct with the probability noted immediately above. We can repeat Steps 1 – 3 to generate several labels for $e_1$ and take a "majority vote" of these labels. Let $y_1$ be the label resulting from the majority vote. The label $y_1$ can be made correct to an arbitrary degree of confidence by taking a sufficient number of labels – i.e. repeating Steps 1 – 3 a sufficient number of times. Let:

$$y = \frac{1}{2}(1 - 2\eta)^{2^{a-1}}; \quad y^2 = \frac{1}{4}(1 - 2\eta)^{2^a}$$

By standard Chernoff bound, for $n$ trials, the probability that the majority vote is incorrect can be made arbitrarily small:

$$\text{Prob} \left[ \text{majority vote is incorrect} \right] \leq e^{-2n \gamma^2} \leq \epsilon$$

So the required number of trials is:

$$n \geq \frac{1}{\gamma^2} \ln \left( \frac{1}{\sqrt{\epsilon}} \right) = O \left( \frac{1}{(1 - 2\eta)^{2^a}} \right) = O \left( 2^{2^a} \right)$$

Step 5. Use $y_1$, the true label of $e_1$, to determine whether the target parity function includes $x_1$.

Using Observation 1, the label $y_1$ directly indicates whether $x_1$ is included in the target parity:

$$y_1 = 1 \text{ if and only if } x_1 \text{ is included in the target parity}$$

Step 6. Repeat Steps 1 – 5 to obtain basis vectors $e_2, \ldots, e_n$ and determine whether the target parity function includes $x_2, \ldots, x_n$.

To find additional basis vectors we can perform Steps 1 – 5 with a minor modification to Step 2. The basis vector $e_i$ has a 1 only in the $i^{th}$ bit. If $i \leq b$ then this bit is in the first block, and we can perform Step 2 exactly as described above – i.e. successively zero out all blocks except the first block. However, if the $i^{th}$ bit is not in the first block, but is in another block $j$, then Step 2 is modified by zeroing out all blocks except block $j$. The resulting set of vectors will have non-zero bits only in block $j$, and will be uniformly distributed so each has a $(1 - 1/2^b)$ probability of not being $e_i$.

**Run time and sample complexity of BKW**

The BKW run time and sample complexity can be made as small as:

$$2^{O \left( \frac{n}{\log n} \right)}$$

---

2 In fact, it is not even essential in the BKW algorithm that blocks be zeroed out in any particular order.
It is clear from Step 1 that the sample complexity is $O(a^{2b})$ per basis vector, so to process all basis vectors requires $O(a^{2b}n)$ examples.

For run time we analyze the time required at each step in the algorithm. For clarity the run times expressions are not simplified as much as permitted by the notation.

Step 1: Drawing $a^{2b}$ examples takes $O(a^{2b})$ time.

Step 2: Each reduction from an i-sample to an (i+1)-sample takes $O(sn)$, and here each $s \leq a^{2b}$. There are $(a-1)$ reductions, which takes $O(a^{2b}n)$ time.

Step 3: Searching all $b$ bits of the first block for each of the $2^b$ vectors can take up to $O(b^2b)$. Even if the vector happens to be absent from the set and steps 1 – 2 must be repeated, this only increases run time by a constant amount for any reasonable accuracy. The total run time for steps 1 – 3 is $O(a^{2b}n)$.

Step 4: The required number of trials, and so the number of times to repeat steps 1 – 3, is:

$$O\left(2^{2^b}\right)$$

The total time for steps 1 – 4 is $O\left(2^{2^b}a^{2^b}n\right)$

Step 5: Takes constant time.

Step 6: Repeat steps 1 – 5 $n$ times. The total run time for the BKW algorithm is:

$$O\left(2^{2^b}a^{2^b}n^2\right)$$

Let

$$a = \frac{1}{2} \log n; \quad b = \frac{2n}{\log n}$$

$$2^a = \sqrt{n}; \quad 2^b = 2^{\frac{2n}{\log n}}$$

Then the total run time above can be simplified to:

$$2^{O\left(\frac{n}{\log n}\right)}$$

and the sample complexity can likewise be simplified to the same value:

$$O\left(a^{2^b}n\right) = 2^{O\left(\frac{n}{\log n}\right)}$$

Note that the noise rate $\eta$ is not present in these quantities because $\eta$ determines some constant factor for the run time.

Recall from lecture 11 that the ability to learn parity functions efficiently in the presence of RCN would also permit the following:

- breaking cryptographic systems predicated on the assumption that this parity problem is hard
- decoding a random linear code under random errors
- efficiently learning juntas, DTs, DNFs (See [FGKP])

### 2.2 Further Developments

Below are four developments relevant to the BKW algorithm for learning parity functions. Each of the last three have the same time and sample complexity as BKW.
1. [Lyubashevsky] shows an algorithm with sample size \( n^{1+\varepsilon} \) and time \( 2^{O(\frac{n}{\log \log n})} \). While this is an improvement in sample complexity (i.e. polynomial rather than merely sub-exponential), there is not only an increase in run time but also reduced tolerance to error rate \( \eta \).

2. [BKW] further presents results for learning parity functions with RCN under arbitrary (rather than uniform) distributions.

3. [FGKP] show that under the uniform distribution, learning under adversarial noise is not any harder than under RCN.

4. [KMV] show a method for agnostically learning parity functions under arbitrary distributions with adversarial noise, using agnostic boosting to combine weak learners.

3 Learning with Membership Query Oracles

In the Membership Query (MQ) model, the oracle returns the label for an example that is specified by the learner. In other words, the MQ oracle provides the learner with "black box" access to the target function. This ability provides additional power for learning. There are concept classes which cannot be learned efficiently with access to a uniform random example oracle but can be learned efficiently given access to a MQ oracle. More importantly, the MQ model is better for many natural learning problems.

Some types of problems which can be learned efficiently with in the MQ model include:

- r-juntas (Section 3.1 below)
- finding large Fourier coefficients (Section 3.2 below)
- size-s DTs in time poly(n, s,1/\varepsilon) (See [KM] and Section 3.3 below)
- s-term DNFs in time poly(n,s,1/\varepsilon) (See [Jackson] and [BJT])
- Boolean functions with small Fourier L_1 norm (See Section 3.3 below)

3.1 Learning juntas with an MQ oracle

Recall that an r-junta is a Boolean function \( f \) which depends on only \( r \) relevant variables out of \( n \) total variables, and the goal is to learn which are the relevant variables. The MQ model permits juntas to be learned very readily, as the following simple algorithm shows.

1. Pick two examples \( x, y \in \{-1,1\}^n \) which differ in at least one bit position, such that \( f(x) \neq f(y) \).

2. Construct a new example \( z \) such that \( z \) matches \( x \) and \( y \) in the bit positions where \( x \) and \( y \) match, and \( z \) differs from \( x \) in at least one bit position and also differs from \( y \) in at least one other bit position.

3. If \( f(z) = f(x) \), set \( x = z \), otherwise set \( y = z \). Repeat steps 2 and 3 until \( x \) and \( y \) differ in one position and \( f(x) \neq f(y) \). This bit position is a relevant variable.

4. Set the bit position learned in step 3 equal to 0, and (recursively) repeat steps 1 – 4 to find the remaining relevant variables. Also set the bit position learned in step 3 equal to 1, and (recursively) repeat steps 1 – 4 to find the remaining relevant variables.
In Step 1, it may be nontrivial to find a pair of examples that yield different function values. For example, the function \( f \) may be a constant (no relevant variables) or may be well-approximated by a constant if most examples yield the same function value.

Note that if there are \( r \) relevant variables there are \( 2^r \) different ways in which the values of the relevant variables may be set. All settings cannot yield the same function value, so at least one of the \( 2^r \) settings has a function value different than the others. Therefore, if we randomly and uniformly choose examples (and therefore randomly and uniformly choose values for the \( r \) relevant variables), there is at least a \( 1/2^r \) chance that we pick an example with a function value different than the most common function value.

Consequently, choosing \( \text{poly}(2^r) \) random examples provides a high probability that we will select examples with both function values. If all \( \text{poly}(2^r) \) function values are the same, we conclude with high probability that the function \( f \) is constant. Since each example is length \( n \), selecting a random example is performed in time \( O(n) \) so determining whether \( f \) is constant (or finding a pair of examples with different function values) takes time \( \text{poly}(2^r, n) \).

In step 2, we construct \( z \) that is along the path on the Boolean hypercube from \( x \) to \( y \). Let \( P \) be the set of bit positions in which \( x \) and \( y \) differ. The construction guarantees that \( z \) matches \( x \) in some positions in \( P \) and matches \( y \) in the remaining bit positions in \( P \). The new example \( z \) is therefore closer along the Boolean hypercube to \( x \) and to \( y \) than the distance between \( x \) and \( y \). The most efficient way to construct \( z \) is such that \( z \) differs from \( x \) by half of the bit positions in \( P \), and therefore differs from \( y \) by the other half of the bit positions. This places \( z \) at the midpoint along the path from \( x \) to \( y \), and so this step is in general repeated \( \log n \) times.

In step 3, we are effectively moving \( x \) or \( y \) to shorten the distance between between those points on the Boolean hypercube. We repeat until that distance is one, so that the two points are adjacent and thus reveal the relevant variable.

In step 4, note that in fixing the value of a relevant variable in an \( r \)-junta, we transform the problem to that of learning a \( (r-1) \)-junta. Since we perform two recursions for each relevant variable, there are at most \( 2^r \) recursive iterations. Therefore the entire algorithm requires time \( \text{poly}(2^r, n) \) and sample complexity \( \text{poly}(2^r, \log n) \).

### 3.2 Learning Fourier coefficients with an MQ oracle

The MQ model is very useful for identifying "large" Fourier coefficients regardless of where those coefficients are. Recall that a subset \( S \) of the variables \( \{1, 2, \ldots, n\} \) is denoted \( S \subseteq [n] \). Also recall that a function \( f: \{-1, 1\}^n \rightarrow \{-1, 1\} \) has \( 2^n \) Fourier coefficients, one for each of the \( 2^n \) possible subsets \( S \subseteq [n] \). We would like to identify large Fourier coefficients, i.e. identify those \( S \subseteq [n] \) in which

\[
\hat{f}(S) \geq \theta \quad \text{for some } \theta > 0
\]

The basic approach involves partitioning the space of all \( S \subseteq [n] \) according to which of the first \( k \) variables are included in \( S \). For example, for \( k = 2 \), the space of all \( S \subseteq [n] \) can be partitioned as follows:

- \( S \) does not include 1 or 2
- \( S \) includes 1 but not 2
In general for any $k$ the space of all $S$ (and therefore all Fourier coefficients) can be partitioned into $2^k$ sets according to which of the first $k$ variables the subset $S$ includes. Every Fourier coefficient is assigned to one of the $2^k$ sets.

Define for a given $k$ and a given subset $S_1 \subseteq [k]:$

$$f_{k,S_1}(x) = \sum_{T_2 \subseteq [k+1,n]} \hat{f}(S_1 \cup T_2) \chi_{T_2}(x)$$

Here $[k+1, n]$ denotes all possible subsets of $\{k+1, k+2, \ldots, n-1, n\}$, so $T_2$ ranges over all the $2^{n-k}$ possible subsets.

It will be useful to consider $S_1$ as a 'prefix' which indicates which of the first $k$ variables are included in the subset $S$ of $[n]$, and $T_2$ as a 'suffix' which indicates which of the last $(n-k)$ variables are included in $S$. Therefore, $S_1 \cup T_2 = S$. For example, for $k = 2$, the subset $S = \{1,4,5\}$ has prefix $S_1 = \{1\}$ and suffix $T_2 = \{4, 5\}$.

Since $S_1 \cup T_2 \subseteq [n]$, in the above equation, the term

$$\hat{f}(S_1 \cup T_2)$$

is some Fourier coefficient in $[n]$. Since the summation in the above equation takes $T_2$ over all subsets of the final $(n-k)$ variables, $T_2$ is taken over all possible suffixes of $S$. In other words, the above equation iterates over all Fourier coefficients for the parities that begin with prefix $S_1$.

For $k = 0$, since $S_1 \subseteq [k]$, $S_1$ can only be the empty set $\emptyset$ and $T_2 \subseteq [1, n] = [n]$, so

$$f_{0,\emptyset}(x) = \sum_{T_2 \subseteq [1,n]} \hat{f}(T_2) \chi_{T_2}(x) = f(x)$$

Similarly, for $k = n$, then there are no variables in $T_2$, so $T_2 = \emptyset$. Recall from lecture 4 that $\chi_{\emptyset}(x) = 1$, so

$$f_{n,S_1}(x) = \hat{f}(S_1)$$

In other words, when $k = n$ the right hand side evaluates to a specific Fourier coefficient. We will eventually want to investigate values of the function for $k = n$.

**Lemma 3:**

Fix $0 \leq k \leq n$; $S_1 \subseteq [k]$

For any $x \in \{-1, 1\}^n$:

$$f_{k,S_1}(x) = E_{y \in \{-1, 1\}^k} \left[ f(yx) \chi_{S_1}(y) \right]$$

**Proof:**

Fairly straightforward manipulation of the expectation on the right hand side will yield the left hand side. First, apply the standard Fourier expansion of the first term inside the expectation:

$$f(yx) = \sum_{T \subseteq [n]} \hat{f}(T) \chi_T(yx)$$

Let $T = T_1 \cup T_2$, such that

$$T_1 = T \cap [k] \quad \text{T}_1 \text{ is the prefix part of T (the first } k \text{ variables)}$$
$$T_2 = T \cap [k+1, n] \quad \text{T}_2 \text{ is the suffix part of T (the last } n-k \text{ variables)}$$

Recall from lecture 4 the definition:
\[ \chi_S = \prod_{i \in S} x_i \]

so for \( T = T_1 \cup T_2 \) we can decompose the parity on \( T \) as:

\[ \chi_T(yx) = \chi_{T_1}(y) \chi_{T_2}(x) \]

Therefore the Fourier expansion can be rewritten as

\[ f(yx) = \sum_{T_1 \subseteq [k]} \sum_{T_2 \subseteq [k+1,n]} \hat{f}(T_1 \cup T_2) \chi_{T_1}(y) \chi_{T_2}(x) \]

Plugging this value for \( f(yx) \) into the expectation above yields:

\[ E_{y \in \{-1,1\}}[\sum_{T_1} \sum_{T_2} \hat{f}(T_1 \cup T_2) \chi_{T_1}(y) \chi_{T_2}(x) \chi_{S_1}(y)] \]

The terms that don’t depend on \( y \) can be moved outside the expectation, and by linearity of expectation:

\[ = \sum_{T_1} \sum_{T_2} \hat{f}(T_1 \cup T_2) \chi_{T_2}(x) E_{y \in \{-1,1\}}[\chi_{T_1}(y) \chi_{S_1}(y)] \]

Recall from lecture 4 that parity functions are orthonormal, so the expected value above equals zero when \( T_1 \neq S_1 \), and equals one when \( T_1 = S_1 \). This greatly simplifies the expected value to:

\[ = \sum_{T_2 \subseteq [k+1,n]} \hat{f}(S_1 \cup T_2) \chi_{T_2}(x) = f_{k,S_1}(x) \]

In the equation of Lemma 3, the right hand side is an average over all \( 2^k \) settings of the first \( k \) variables. Note also that the quantity in the expectation is a binary value since both \( f \) and \( \chi \) are binary values. We will see that this lemma will allow us to generate random values for \( y \), concatenate some fixed suffix \( x \) to these \( y \)'s, and use the MQ oracle to obtain a corresponding set of values \( f(yx) \). We can then obtain the average value which is the right hand side of

\[ f_{k,S_1}(x) = E_{y \in \{-1,1\}}[f(yx) \chi_{S_1}(y)] \]

which by the equation also provides an estimate for the left hand side. This leads to the following observation.

**Observation 2:**

Given

\[ 0 \leq k \leq n; S_1 \subseteq [k]; x \in \{-1,1\}^{n-k}; \text{MQ Oracle access to } f \]

we can estimate \( \hat{f}_{k,S_1}(x) \) to arbitrary accuracy.

Since

\[ f_{k,S_1}(x) = E_{y \in \{-1,1\}}[f(yx) \chi_{S_1}(y)] \]

and since the product inside the expectation

\[ E_{y \in \{-1,1\}}[f(yx) \chi_{S_1}(y)] \]

is a binary value, we can output a high quality estimate \( v \) with confidence \( \delta \) such that

\[ |v - f_{k,S_1}(x)| \leq y; \quad y > 0 \]

in time \( O(n/\gamma^2 \log 1/\delta) \). Note that the operations are on operations on n-bit strings, and we use
standard Chernoff bounds to take $O(1/\gamma^2)$ samples by $O(1/\gamma^2 \log 1/\delta)$ calls to the MQ oracle.

The following Lemma provides a critical connection to between Fourier coefficients, which we want to discover, and $f_{k,S_1}$, which we can estimate with MQ oracle access.

**Lemma 4:**
Fourier coefficients are related to $f_{k,S_1}(x)$ by the equation

$$E_x[f_{k,S_1}(x)^2] = \sum_{T_2 \subseteq [k+1,n]} \hat{f}^2(S_1 \cup T_2)$$

**Proof:**
Recall from lecture 5 that, for any real valued function $g$ of Boolean inputs $x$, Plancherel's identity states:

$$E_x[g^2(x)] = \sum_{S \subseteq [n]} \hat{g}^2(S)$$

Applying this identity to $f_{k,S_1}(x)$ yields:

$$E_x[f_{k,S_1}(x)^2] = \sum_{T_2 \subseteq [k+1,n]} \hat{f}^2(S_1 \cup T_2)$$

□

Note that we can estimate the left hand side of this equation. From Observation 2, we can efficiently and accurately estimate $f_{k,S_1}$, so we can estimate the expected value of its square. This together with Lemma 4 means that we can estimate the sum of the squares of all Fourier coefficients that start with any desired prefix $S_1$. This will serve as the foundation of an efficient algorithm for locating any large Fourier coefficients which exist.

In the expectation on the left hand side of the above equation, note that the inputs $x$ are $(n-k)$-bit strings in accordance with the equation in Lemma 3. Also note that on the right hand side of the above equation are those Fourier coefficients of the subsets of $[n]$ which match $S_1$ in the first $k$ bits. Equivalently, the summation on the right hand side is over all suffixes that can be concatenated to $S_1$.

Keeping in mind the goal is to find all sets $S \subseteq [n]$ such that $\hat{f}(S) \geq \theta$, we turn to three observations.

**Observation 3.**
At most $\frac{1}{\theta^2}$ sets $S$ have the property $|\hat{f}(S)| \geq \theta$.

Note that if there were more, then the sum of coefficients squared would violate Parseval's identity.

$$\sum_{S \subseteq [n]} \hat{f}^2(S) = 1$$

**Observation 4.**
For any fixed $0 \leq k \leq n$:

At most $\frac{1}{\theta^2}$ of the functions $f_{k,S_1}(x)$ have the property $E_{x \in \{-1,1\}^{n-k}}[f_{k,S_i}(x)^2] \geq \theta^2$. 
From Lemma 4:
\[ E_{x \in \{-1,1\}^{n}}[f_{k, S_1}(x)^2] = \sum_{T_2 \subseteq [k+1, n]} \hat{f}(S_1 \cup T_2)^2 \]

Summing both sides of the above equation over all \( S_1 \), we again obtain by Parseval's identity:
\[ \sum_{S_1} E_{x \in \{-1,1\}^{n}}[f_{k, S_1}(x)^2] = \sum_{S_1} \sum_{T_2 \subseteq [k+1, n]} \hat{f}(S_1 \cup T_2)^2 = \sum_{S \subseteq [n]} \hat{f}(S)^2 = 1 \]

As in Observation 3, if there were more than \( 1/\theta^2 \) such functions, then the sum of coefficients squared would exceed 1.

**Observation 5.**
If \( \hat{f}(S) \geq \theta \), and \( S_1 \) is the prefix of \( S \), then
\[ E[f_{k, S_1}(x)^2] \geq \theta^2 \]
This is true since
\[ E[f_{k, S_1}(x)^2] = \sum_{T \subseteq [k+1, n]} \hat{f}(S_1 \cup T)^2 \]
and the right hand side certainly includes a term \( \hat{f}(S) \geq \theta \) for some \( T \) (the suffix of \( S \)).

Observation 5 means that if there is an \( S \) with 'heavy' Fourier coefficients, the prefix \( S_1 \) of \( S \) can be used in the function \( f_{k, S_1} \) to detect that heavy Fourier coefficient. If
\[ E[f_{k, S_1}(x)^2] \leq \theta^2 \]
then there can be no heavy Fourier coefficients among all \( S \) that start with \( S_1 \). This fact will allow us to conduct a type of binary search to investigate increasingly specific prefixes of \( S \) to locate large Fourier coefficients.

In particular, when we determine that
\[ E[f_{k, S_1}(x)^2] \geq \theta^2 \]
then we continue searching among the Fourier coefficients that begin with \( S_1 \) by searching among even 'longer' prefixes. Lemma 4 limits the number of possibilities, and so limits the number of searches, for any length \( k \) of a prefix \( S_1 \).

We now turn to the algorithm for finding large Fourier coefficients. Assuming we can compute, for any \( k, S_1 \)
\[ E_{x \in \{-1,1\}^{n}}[f_{k, S_1}(x)^2] \]
then we can efficiently identify the exact collection of \( S \subseteq [n] \) such that
\[ |\hat{f}(S)| \geq \theta \] in time \( \text{poly}(n, 1/\theta) \)

**Algorithm for Locating Large Fourier Coefficients:**

We construct a binary tree. Each level of the tree corresponds to a value of \( k \), and each of the \( 2^k \) nodes at a level corresponds to a different value of prefix \( S_1 \). The root corresponds to \( k = 0 \), and the leaves correspond to \( k = n \).

For example, for \( k = 1 \), the two prefixes \( S_1 \) are \( \emptyset \) and \( \{1\} \), and for \( k = 2 \), the four prefixes \( S_1 \) are \( \emptyset \), \( \{1\} \), \( \{2\} \), and \( \{1,2\} \). For any prefix over \( k \) variables there are \( 2^n \) possible suffixes. Therefore, each prefix (i.e. each node) corresponds to \( 2^n \) possible suffixes, and therefore corresponds to \( 2^n \) Fourier
coefficients. Consequently, the $2^n$ leaves correspond to all $2^n$ sets $S$, and to all Fourier coefficients.

Figure 2 – Binary Tree of prefixes and Fourier coefficient weights

Let each node in the tree, which corresponds to a level $k$ and a prefix $S_1$ for that $k$, have value:

$$E_{x \in [-1,1]}[f_{k, S_1}(x)^2]$$

By Lemma 4, the value of a node equals the sum of the squares of all Fourier coefficients for prefix $S_1$.

$$E_{x}[f_{k, S_1}(x)^2] = \sum_{T_2 \subseteq [k+1, n]} \hat{f}^2(S_1 \cup T_2)$$

Note that each node, which corresponds to a prefix $S_1$ over $k$ variables, has two child nodes which correspond to the following two prefixes over $(k+1)$ variables:

- $S_1$
- $S_1 \cup \{k+1\}$

For example, in Figure 2 at level $k = 1$, the left node representing $S_1 = \emptyset$ and so for this node the variable $\{1\}$ is not included in the prefix that node corresponds to. The two children of this node (and in fact all descendants of this node) also correspond to prefixes which do not include $\{1\}$.

In other words, the two prefixes of the two child nodes collectively make up all possible prefixes over $(k+1)$ variables which begin with the prefix $S_1$ over $k$ variables. Therefore, each of the $2^k$ Fourier coefficients represented by a parent node at level $k$ is represented by exactly one of its two child nodes. This means the values of the two child nodes must sum to the value of the parent node.

$$E[f_{k+1,S_1}(x)^2] + E[f_{k+1,S_1 \cup \{k+1\}}(x)^2] = E[f_{k,S_1}(x)^2]$$

Note also that, for any $k$, the sum of all values of all $2^k$ nodes at that level in the tree equals one, by the same reasoning as in Observation 4:

$$\sum_{S_1} E_{x \in [-1,1]}[f_{k,S_1}(x)^2] = 1$$

The value of the root node equals 1. Since the root represents $k = 0$, the root represents all Fourier coefficients for prefix $\emptyset$, i.e. all Fourier coefficients. By Parseval's identity the sum of the squares of all Fourier coefficients sums to 1.

Starting at level $k = 1$, we can estimate the value of each node by Observation 2. If the value of
a node is less than $\theta^2$, then by Observation 5 there must be no heavy Fourier coefficients which begin with the prefix corresponding to that node. Therefore none of the children of that node will have any heavy Fourier coefficients. On the other hand, if the value of a node is at least $\theta^2$, then we search among its children for heavy Fourier coefficients.

Though the tree includes an exponential number of nodes, the number of nodes to evaluate is only polynomial. By Observation 4, at most $1/\theta^2$ nodes at any level $k$ can have a value of at least $\theta^2$. Therefore, once we locate $1/\theta^2$ such nodes we need not evaluate the values of any remaining nodes at a level. Consequently we must evaluate at most $n/\theta^2$ nodes in the entire tree. By the time we reach the leaves at level $k = n$, we evaluate only the leaves that correspond to heavy Fourier coefficients.

The above algorithm does not calculate any node values precisely; it relies on estimates for the node values according to Observation 2. Nevertheless the algorithm still works with this noise by the following theorem.

**[KM] Theorem:**

For any Boolean function, $f: \{-1,1\}^n \rightarrow \{-1,1\}$, given oracle MQ($f$), $\theta > 0$, we can efficiently locate the Fourier coefficients with the property:

$$|\hat{f}(S)| \geq \frac{\theta}{2}$$

and with certainty we will locate all Fourier coefficients with the property

$$|\hat{f}(S)| \geq \theta$$

By this theorem, in time $\text{poly}(n, 1/\theta, \log 1/\delta)$, with probability at least $1 - \delta$, we output a list $\{S_1, S_2, \ldots S_t\}$, where each $S_i \subseteq [n]$, such that for all $i$:

$$|\hat{f}(S_i)| \geq \frac{\theta}{2}$$

and for every $S \subseteq [n]$ such that

$$|\hat{f}(S)| \geq \theta$$

it is certain that $S = S_i$ for some $i$.

Just as in "low degree algorithm", we can estimate low degree coefficients to high accuracy. The algorithm above can be viewed as an extension of the LMN algorithm from lecture 6 for learning low-degree polynomials. According to the KM theorem, given a set $\{S_1, \ldots S_t\}$ of 'not too many' coefficients, the set can be located efficiently when it exists.
3.3 Challenge Problem

The following two statements were asserted without proof and are the final challenge problem offered for the course.

(a) Functions with a small $L_1$ norm have almost all of their Fourier weight on a few coefficients. For Boolean function $f$ with the property:

$$\sum_{S \subseteq [n]} \hat{f}^2(S) \leq L$$

for all $S$ such that

$$\hat{f}(S) \geq \frac{\epsilon}{L}$$

there are at most $L^2/\epsilon$ such $S$, and over all such $S$

$$\sum_{S} \hat{f}^2(S) \geq 1 - \epsilon$$

(b) If $f$ is a Boolean function that is computed by an $s$-leaf decision tree, then

$$\sum_{S \subseteq [n]} |\hat{f}(S)| \leq s$$

(Note that this allows us to efficiently learn polynomial-size DTs using a MQ oracle.)