Implementation issues in the Fourier Transform algorithm *

Yishay Mansour[†] **Sigal Sahar**[‡]
Computer Science Dept.
Tel-Aviv University
Tel-Aviv, ISRAEL

Abstract

The Fourier transform of functions with boolean inputs has received considerable attention in the last few years in the Computational Learning Theory community, and has come to play an important role in proving many important learnability results. The aim of this work is to demonstrate that the Fourier transform techniques are also a useful and practical algorithm, in addition to having many interesting theoretical properties.

One of the benefits we present is the confidence level the algorithm produces in addition to the predictions. The confidence level measures the likelihood that the prediction is correct.

In order to have the algorithm run-time be reasonable and still produce accurate hypotheses, we had to perform many optimizations. In the paper we discuss the more prominent optimizations, ones that were crucial and without which the performance of the algorithm would severely deteriorate.

1 Introduction

Over the last few years the Fourier transform representation of boolean functions has been an instrumental tool in the computational learning theory community. It has been used mainly to demonstrate the learnability of various classes of functions with respect to the uniform distribution.

The first connection between the Fourier representation and learnability of boolean functions was established in [6]. The main application in [6] was to learn the class AC^0 (constant depth, polynomial size circuits) in quasi-polynomial time (i.e. $O(n^{poly-log(n)})$) with respect to the uniform distribution. The work of [5] has developed a very powerful algorithmic procedure; given a function and a threshold parameter it finds in polynomial time all the Fourier coefficients of the function that are greater than the threshold. Originally the procedure was used to learn decision trees [5], and later it was used for learning polynomial size DNF [8, 2, 4]. The Fourier transform technique applies naturally to the uniform distribution, and, indeed, most of the learnability results based on the Fourier transform are with respect to the uniform distribution. Some of the results were extended to product distribution [1, 3].

A great advantage of the Fourier transform algorithm is that it needs not make any assumptions regarding the function it is learning. Thus, we can apply it to any function and hope to obtain some "large" Fourier coefficients

*This paper is intended for the theory category.

†e-mail: mansour@cs.tau.ac.il

‡e-mail: gales@cs.tau.ac.il

which can then be easily used in order to make a prediction. The prediction function simply computes the sum of the coefficients with the corresponding basis functions and compares the sum to some threshold. (Essentially it can be viewed as a perceptron, where the Fourier coefficients are the weights and the inputs are the values of the appropriate basis functions.) The procedure is also immune to a certain rate of noise, and it will be able to operate even if a fraction of the examples are maliciously misclassified. However, a drawback of the procedure is that it requires to query the target function on various randomly selected inputs.

The aim of this work is to implement this procedure in an efficient way so that, as we shall demonstrate, it is not only a useful theoretical tool but also a practical one. Note that the procedure scales linearly with the number of input variables, conveniently allowing us to run it on a fairly large number of input variables. The experiments are based on learning of various functions such as decision trees, DNF and majority. The results demonstrate that the procedure can approximate a wide variety of different function classes.

When making predictions, it is extremely beneficial to have the prediction algorithm supply an indicator that provides the confidence level the algorithm has in the prediction it just made. Our algorithm provides us naturally with such an indicator. Suppose that in the prediction process we are computing a sum and then comparing it to some threshold, say 0. Intuitively we can use the value of this sum to indicate how confident we are of our prediction. If the sum is near 0 then we are probably not doing much better than guessing, while if it is far from 0 we can claim that we are confident of our prediction. Though this simple logic thus far has not been supported by any theoretical result, our experimental results provide overwhelming evidence that this is indeed the case.

In the process of implementing the Fourier algorithm we made many important optimizations. The optimizations were essential in order to make the algorithm run in a reasonable amount of time. In addition we made enhancements that significantly improve the accuracy of the hypothesis we generate. Choosing the "right" parameters proved to be essential in order to achieve desirable running time while maintaining a reasonable error rate.

One of the most important enhancements we have added is the ability to detect "on the fly" inaccurate approximations (that the algorithm computes throughout its search) and to immediately correct them. This feature considerably improves the performance of the algorithm with a minimal run time cost.

The paper is organized as follows: section 2 defines the Fourier transform and provides a description of the algorithm. Section 3 describes the experiments and their outcome. Section 4 describes various aspects of our implementation. We end with our conclusions in Section 5.

2 Preliminaries

In this section we sketch the ideas behind Fourier transform, its connection to learning and the algorithm that finds the large coefficients. A comprehensive survey of the theoretical results and proofs can be found in [7].

2.1 Fourier transform

Boolean functions of n variables are considered as real valued functions $f : \{0,1\}^n \to \{-1,1\}$. The inner product of two functions is defined by:

$$< g, f >= 2^{-n} \sum_{x \in \{0,1\}^n} f(x)g(x) = E[g \cdot f],$$

where E is the expected value with respect to the uniform distribution.

The basis is defined as follows: for each $z \in \{0,1\}^n$, define the *basis function* $\chi_z(x)$:

$$\chi_z(x_1, \cdots, x_n) = (-1)^{\sum_{i=1}^{n} x_i z_i}$$

Any function of n boolean inputs can be uniquely expressed as a linear combination of the basis functions. For a function f, the z^{th} *Fourier coefficient* of f is denoted by $\hat{f}(z)$, i.e., $f(x) = \sum_{z \in \{0,1\}^n} \hat{f}(z)\chi_z(x)$. The Fourier coefficients are computed by $\hat{f}(z) =< f, \chi_z >$ and we call z the *coefficient-name* of $\hat{f}(z)$.

A *t-sparse function* is a function that has at most t non-zero Fourier coefficients.

```
SUBROUTINE search(α)
        IF TEST[f, α, θ] THEN
                IF |α| = n THEN OUTPUT α
                        ELSE search(α0); search(α1);
```

Figure 1: Subroutine `search`

2.2 Prediction

Our aim is to approximate the target function f by a t-sparse function h. In many cases h will simply include the "large" coefficients of f. In other words, assume that $\Lambda = \{z_1, \ldots, z_m\}$ is the set of z's for which $\hat{f}(z_i)$ is "large". We set $h(x) = \sum a_i \chi_{z_i}(x)$, where a_i is our approximation of $\hat{f}(z_i)$. The hypothesis we generate using this process, $h(x)$, does not have a boolean output. In order to obtain a boolean prediction we use $sign(h(x))$, i.e., output $+1$ if $h(x) \geq 0$ and -1 if $h(x) < 0$.

We would like to bound the error we get from using h to approximate f using the expected error squared, i.e., $E[(f-h)^2]$. It can be shown that bounding the expected error squared bounds the boolean prediction error probability, i.e.,

$$\Pr[f(x) \neq sign(h(x))] \leq E[(f-h)^2].$$

For a given t, the hypothesis h which is t-sparse and minimizes $E[(f-h)^2]$ simply includes the t largest coefficients of f. Note that the more coefficients we include in our approximation and the better we approximate their values, the smaller $E[(f-h)^2]$ is going to be. This provides us with the motivation to find the "large" coefficients.

2.3 Finding the large coefficients

We briefly describe the algorithm that finds the "large" coefficients of a function. The algorithm receives as inputs a function f (a black-box it can query on various inputs) and an interest threshold parameter $\theta > 0$. It outputs a list of coefficient-names that (1) includes all the coefficients-names whose corresponding coefficients are *"large"*, i.e., at least θ, and (2) does not include "too many" coefficient-names. The algorithm runs in polynomial time in both $1/\theta$ and n.

The basic idea of the algorithm is to perform a search in the space of the coefficient-names of the input function f. Throughout the search we maintain a prefix of a coefficient-name, and try to estimate whether *any* of its extensions can be a coefficient-name whose value is "large". The search is performed like any generic search procedure (see Figure (1)). The algorithm commences by calling `search`(λ) where λ is the empty string. On each invocation it computes the predicate $TEST[f, \alpha, \theta]$. If the predicate is true it recursively calls `search`($\alpha 0$) and `search`($\alpha 1$). The interesting part in the search is the implementation of the $TEST[f, \alpha, \theta]$ predicate. (Note that if $TEST$ is very permissive we might reach all the coefficients, in which case our running time will not be polynomial).

Formally, $TEST[f, \alpha, \theta]$ computes whether

$$E_{x \in \{0,1\}^{n-k}} E^2_{y \in \{0,1\}^k}[f(yx)\chi_\alpha(y)] \geq \theta^2, \quad \text{where } k = \|\alpha\|. \tag{1}$$

Define $f_\alpha(x) = \sum_{\beta \in \{0,1\}^{n-k}} \hat{f}(\alpha\beta)\chi_\beta(x)$. It can be shown that the expected value in (1) is exactly the sum of the squares of the coefficients whose prefix is α, i.e.,

$$E_{x \in \{0,1\}^{n-k}} E^2_{y \in \{0,1\}^k}[f(yx)\chi_\alpha(y)] = E_x[f^2_\alpha(x)] = \sum_{\beta \in \{0,1\}^{n-k}} \hat{f}^2(\alpha\beta),$$

This implies that if there exists a coefficient $|\hat{f}(\alpha\beta)| \geq \theta$, then the expected value is at least θ^2. This condition guarantees the correctness of our algorithm, namely that we reach all the "large" coefficients. We would like also to bound the number of recursive calls that `search` performs. We can show that for at most $1/\theta^2$ of the prefixes of size k, $TEST[f, \alpha, \theta] = TRUE$. This bounds the number of recursive calls in our procedure by $O(n/\theta^2)$.

In $TEST$ we would like to compute the expected value, but in order to do so efficiently we settle for an approximation of its value. This can be done as follows: (1) we choose m_1 random $x_i \in \{0,1\}^{n-k}$, (2) we choose

m_2 random $y_{i,j} \in \{0,1\}^k$, (3) we query f on $y_{i,j}x_i$ (which is why we need the query model - to query f on many points with the same prefix x_i) and receive $f(y_{i,j}x_i)$, and (4) compute the estimate as,

$$B_\alpha = \frac{1}{m_1}\sum_{i=1}^{m_1}\left(\frac{1}{m_2}\sum_{j=1}^{m_2} f(y_{i,j}x_i)\chi_\alpha(y_{i,j})\right)^2 .$$

On each invocation of the procedure the parameters we provide are θ, m_1 and m_2.

3 Experiments description

We have implemented the algorithm described in Section 2.3 and then went forth to verify the theoretical findings by running a series of experiments. The parameters of each experiment include, in addition to the function to be learned, θ, m_1 and m_2. We will later include a more detailed discussion of these parameters, but we will briefly introduce their meaning here.

The parameter θ determines the threshold between "small" and "large" coefficients. Thus, θ controls the number of coefficients we will output, and how many recursive calls we will perform.

The parameters m_1 and m_2 determine how accurately we approximate the $TEST$ predicate each time. Should we fail to approximate the $TEST$ predicate accurately, $TEST$ may return faulty, even random, results (as it will, for example, for a ludicrous choice of $m_1 = 1$ and $m_2 = 1$). An inaccurate approximation of the value in $TEST$ (due to an inadequate choice of parameters) may cause the algorithm to skip interesting subtrees (ones that contain "large" coefficients at the leaves), causing the final hypothesis to be less accurate. Moreover, it may also cause the algorithm to traverse subtrees that are "uninteresting" (containing no "large" coefficients at the leaves), leading to a substantial increase in the run-time. An intelligent choice of m_1 and m_2 is therefore indispensable. This issue is discussed in greater detail in Section 4.2.

Figures (3)-(6) present representative results of our experiments in the form of graphs that evaluate the output hypothesis of the algorithm on randomly chosen test points. The target function, f, returns a boolean response, ± 1, while the Fourier Transform hypothesis, h, returns a real response. We therefore present, for each experiment, a graph that includes two curves: the frequency of the values of the Fourier Transform hypothesis, $h(x)$, when $f(x) = +1$, and the second curve for $f(x) = -1$.

If the two curves intersect, their intersection represents the inherent error the algorithm makes. Errors occur in two cases: (1) the algorithm predicts a $+1$ response when the actual response is -1 (the lightly shaded area in Figure (2)), and (2) the algorithm predicts a -1 response, while the true response is $+1$ (the darker shade area in Figure (2)).

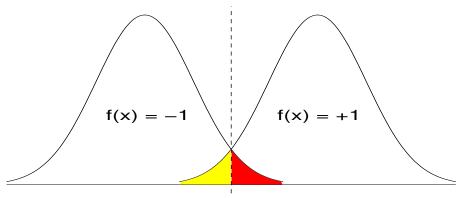

Figure 2: Typical frequency plots and typical errors

3.1 Confidence levels

One of the most consistent and interesting findings was the distribution of the error versus the value of the algorithm's hypothesis: its shape is always that of a bell shaped curve. This feature usually permits us to determine with a high (often 100%) confidence level the result for most of the instances. In fact, it gives us much more than that: it serves as the indicator of how confident we are of our prediction. Since we know the error distribution we can output a certain confidence level for each prediction we make. Let us demonstrate the strength of this technique. Consider the results of the 16-term DNF (Figure (4)). If the algorithm's hypothesis outputs -0.21 (translated into -1 in boolean terms by the **sign** function), we know with an 83% confidence level that the prediction is, indeed, correct. If the algorithm outputs 0.795 as its prediction, we can virtually guarantee that the response is correct. Thus, although the total error level is over 9%, we can tell for each point how confident we are in our prediction. This is an indispensable tool for practical usage of the hypothesis.

3.2 Results

Representative results of the experiments we ran appear in Figures (3)-(6). All the function have 40 or 41 boolean input variables. Figures (3) and (4) include the results for DNF with 8 and 16 terms respectively. Figure (5) contains two graphs portraying the results for fully balanced decision trees of depth 5 and 3, respectively. Figure (6) depicts the majority function.

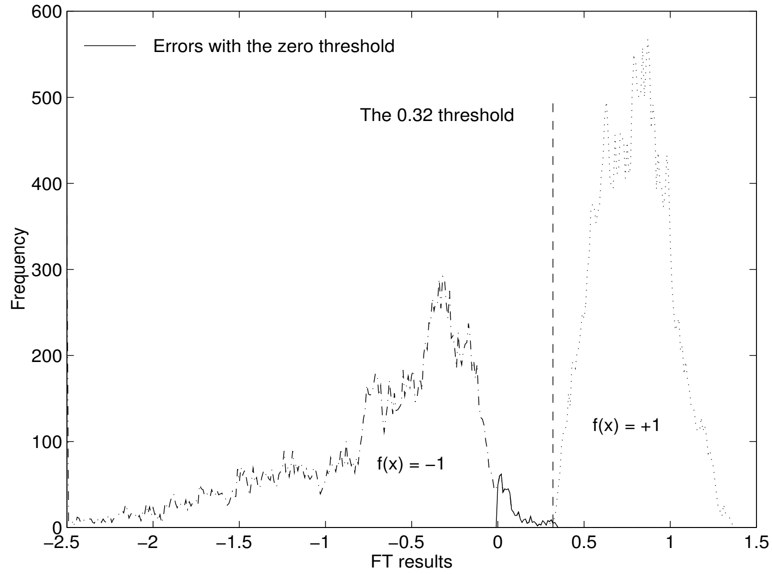

Figure 3: **8 term DNF**. This (randomly generated) DNF of 40 variables returns -1 about 43% of the time. The results shown above are for the values of $\theta = 0.03$, $m_2 = 5600$ and $m_1 = 100$. The hypothesis used 112 non-zero coefficients. The error detected for this function calculated with two thresholds as explained and detailed in Section 4.4 was 1.22% and 0.068%.

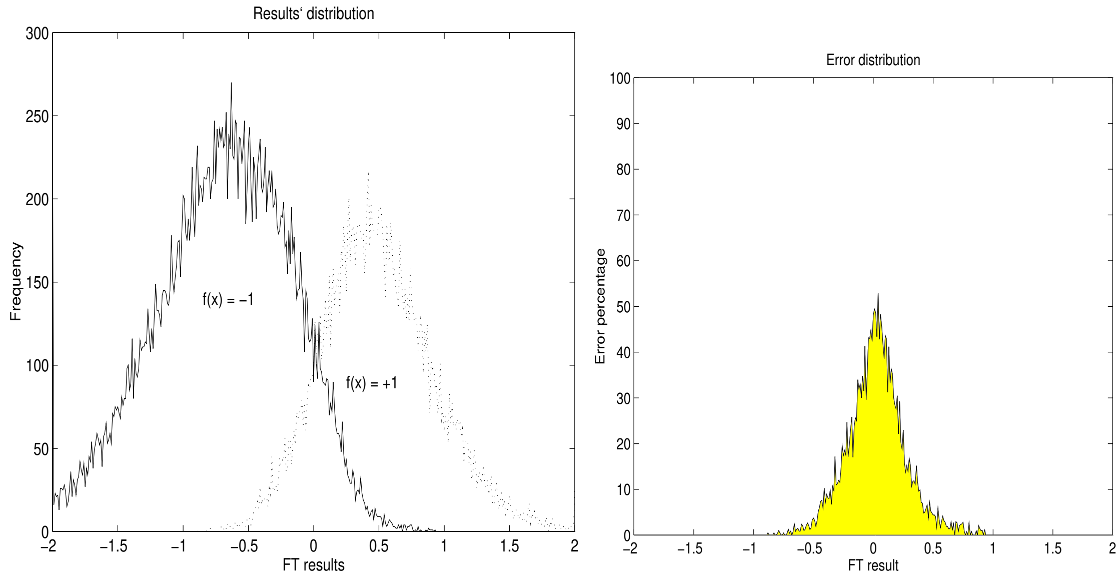

Figure 4: **16 term DNF.** This (randomly generated) DNF of 40 variables returns -1 about 61% of the time. The results shown above are for the values of $\theta = 0.02$, $m_2 = 12500$ and $m_1 = 100$. The hypothesis used 186 non-zero coefficients and a total of 9.628% error was detected.

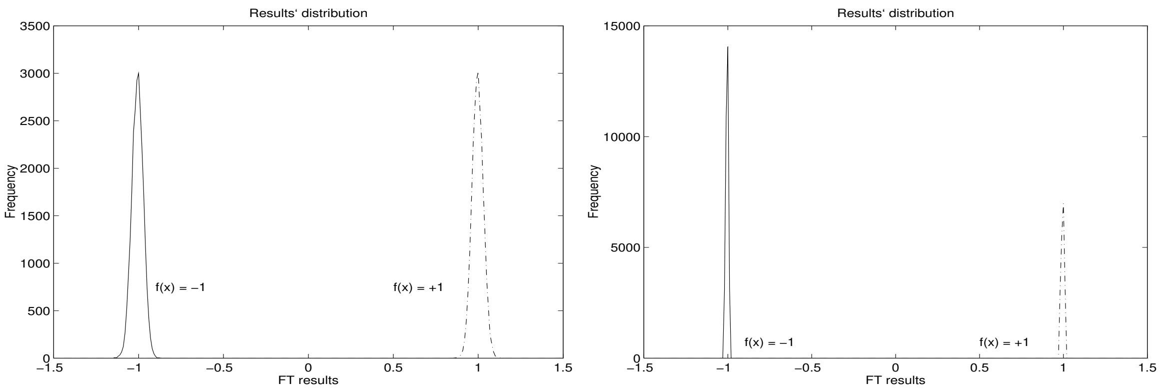

Figure 5: **Decision trees of depth 5 and 3** with 41 variables. The 5-deep (3-deep) decision tree returns -1 about 50%(62.5%) of the time. The results shown above are for values $\theta = 0.03$, $m_1 = 100$ and $m_2 = 5600$ ($\theta = 0.06$, $m_1 = 100$ and $m_2 = 1300$). Both graphs are totally disjoint, signifying 0% error.

4 Implementation enhancements

4.1 Retry

During our experimentations we have noticed that at times our estimate B_α for $E[f_\alpha^2]$ might be inaccurate. An error in this estimate can lead to our aborting the transversal of a subtree that may contain leaves (i.e., coefficients) that should, in fact, be included in the algorithm's hypothesis. We have devised a method to detect many of these errors and correct them in an efficient manner.

The properties of the Fourier transform guarantee that $E[f_\alpha^2] = E[f_{\alpha 0}^2] + E[f_{\alpha 1}^2]$, therefore we expect that $B_\alpha \approx B_{\alpha 0} + B_{\alpha 1}$. Whenever this is not true, we conclude that at least one of our approximations is somewhat lacking, which can lead the algorithm to miss important coefficients at the leaves of the corresponding subtrees. We will remedy the situation by running the `search` procedure again on the respective children, i.e., we will "retry" the node α.

This solution will, of course, increase the probability of finding all the "large" coefficients. However, a brute force implementation may cost us an inordinate amount of time since we will traverse subtrees that we have previously visited. We therefore take care to traverse only subtrees that were, until now, unvisited. One should further note that it is because any discrepancies between the parent and its children are discovered—and corrected—as soon as they appear, that there is no need to retraverse a subtree we have already visited.

In order to demonstrate the usefulness of this approach we bring forth the example of learning the majority function. When learning a majority function of 41 boolean variables using the parameters $m_1 = 100$, $m_2 = 800$ and $\theta = 0.08$, without the retry mechanism we receive the results depicted by the dotted-dashed line in Figure (6) (13.724% error). The shaded area in Figure (6) describes the errors that have occured when we ran the algorithm without retries. The algorithm perceived errors since without the retries it missed eight (of a total of 42) large coefficients. *With* the retries all the correct coefficients were found, and even two additional ones were added, but their values are negligible. The results we received *with* the retries were faultless, that is, the Fourier transform approximation always output the correct result. These results are presented in the dotted curve in Figure (6).

[1]One may notice the somewhat odd distribution of the results in Figure (6): all the results are centered in a few peaks. This distribution is not coincidental. Upon closer examination of the FT of the majority function, one encounters 42 *equal* coefficients: one coefficient for each singleton (a vector of the form 00..010..00) and an additional coefficient for the parity coefficient (the all-ones vector). Since all the coefficients are equal, let us label with a c_{maj} the value of each coefficient. For an input vector with z zeros (and $41 - z$ ones), we will have a contribution of $\pm|(2z - 41) * c_{maj}|$ to the result, where $2z - 41$ is obviously odd. The parity will contribute an additional $\pm c_{maj}$ (depending on whether z is odd or even), so that the total contribution to the final result is an even factor of c_{maj}. Since $c_{maj} = \binom{40}{20}\frac{1}{2^{40}} \sim 0.12$, we have peaks around factors of 0.24. The distribution around the peaks is due to the fact we only approximate each coefficient and get a value close to $\binom{40}{20}\frac{1}{2^{40}}$.

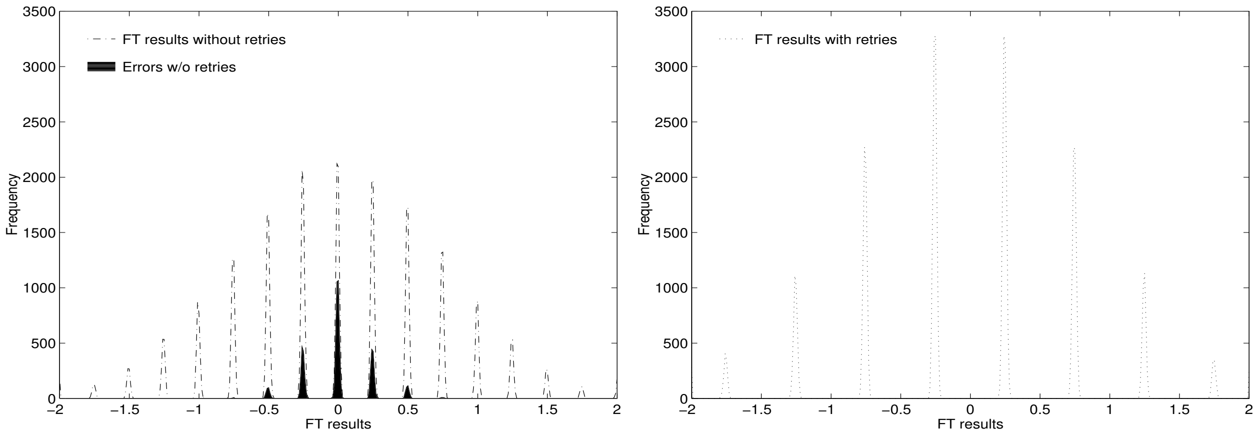

Figure 6: **Majority function.** The Fourier transform results with retries, plotted in a dotted line on the right, show no errors. The Fourier transform results without retires, plotted in the dashed-dotted line on the left, show the errors shaded in black. Note the majority-function characteristic distribution of the results[1].

4.2 Determining the parameters

One of our aims was to determine the values of the different parameters, m_1, m_2 and θ. Recall that in our algorithm we calculate B_α, the approximation of $E_x[f_\alpha^2(x)]$ and that m_1 is the number of times we sample x in order to make this approximation. We sample y randomly m_2 times in order to approximate $f_\alpha(x_i) = E_y[f(yx_i)\chi_\alpha(y)]$, for each x_i. This approximation of $f_\alpha(x_i)$ has a standard deviation of approximately $\frac{1}{\sqrt{m_2}}$. Assume that the true value is β_i, i.e. $\beta_i = f_\alpha(x_i)$, then we expect that the contribution of the i^{th} element to B_α to be $(\beta_i \pm \frac{1}{\sqrt{m_2}})^2 = \beta_i^2 \pm \frac{2\beta_i}{\sqrt{m_2}} + \frac{1}{m_2}$.

The algorithm tests $B_\alpha = \frac{1}{m_1}\sum \beta_i^2 \geq \theta^2$, therefore, in order to ensure a low error, based on the above argument we choose

$$m_2 = \frac{5}{\theta^2}.$$

Choosing the right value for m_2 is of great importance. We have noticed on more than one occasion that to a certain degree, *increasing* the value of m_2 actually *decreases* the overall running time. This is not necessarily obvious at first. One would think that any increase in the number of times we loop in the algorithm would only increase the running time. However, we must not forget that a more accurate value for m_2 means a more accurate approximation of the values we are searching, and therefore less chance of redundant recursive calls. The running time, of course, is linear in the number of recursive calls.

We can see this exemplified in Figure (7) where we see the number of recursive calls increase drastically as m_2 *decreases*. In order to present Figure (7), we learned the same 3 term DNF always using $\theta = 0.05$ and $m_1 * m_2 = 100000$. The trials differ in the specific values chosen in each trail for m_2; for example, when $m_2 = 500$, $m_1 = 200$.

4.3 Special cases

We have noticed that when $k = \|\alpha\|$ is either very small or very large, the values we choose for m_1 and m_2 are sometimes self-defeating. That is to say, when $k \sim n$ (n is the number of variables in the target function), we still loop m_1 ($> 2^{n-k}$) times, though often the looping supplies us with no new information. For example, when $k = n - 4$, in order to calculate the *exact* expected value we only need sample $2^4 = 16$ different values, while ordinarily we would have sampled m_1 random times when *always* $m_1 \gg 16$. Thus, we gain no new information, and only waste valuable time. The same holds for very small values of k, and the corresponding m_2 ($> 2^k$) values.

We therefore add the following feature: for small and large values of k we calculate exactly the expected value thus decreasing the run time and adding accuracy.

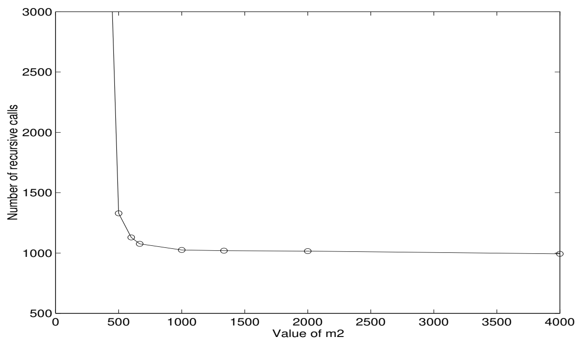

Figure 7: **Determining** m_2. Note that the number of recursive calls grows dramatically as m_2's value decreases. For example, for $m_2 = 400$, the number of recursive calls is $14,433$ compared with only $1,329$ recursive calls for $m_2 = 500$.

4.4 Determining the threshold

Once the list of large coefficients is built and we compute the hypothesis $h(x)$, we still need to determine the threshold, a, to which we compare $h(x)$ (i.e., predict $+1$ iff $h(x) > a$). In the theoretical work it is assumed that $a = 0$, and, indeed, a priori one cannot deduce a better guess. In our experimentations we observed that fixing a different value for a, depending on the hypothesis $h(x)$ we have calculated, can be extremely useful. In order to find the value of a we consider a number of random examples and determine the optimal a with respect to them.

For example,when trying to learn an 8-term DNF with the parameters $m_1 = 100$, $m_2 = 5600$ and $\theta = 0.03$ we received the results depicted in Figure (3). If we use the zero threshold we will receive a total of 1.22% overall error. However, if we choose the threshold to be 0.32, we will get a diminished error of 0.068%.

5 Conclusions

In this work we have implemented the Fourier transform algorithm and showed that it is a useful practical tool, as well as a powerful theoretical technique. We invested a considerable amount of time in optimizing the algorithm, and mentioned in this paper the major points in the optimization process. The algorithm runs in a reasonable amount of time and is able to recover quite successfully various functions.

We have shown that the algorithm naturally derives a confidence parameter. This parameter enables the user in many cases to conclude that the prediction received is accurate with extremely high probability, even if the overall error probability is not negligible.

References

[1] Mihir Bellare. A technique for upper bounding the spectral norm with applications to learning. In 5^{th} *Annual Workshop on Computational Learning Theory*, pages 62–70, July 1992.

[2] Avrim Blum, Merrick Furst, Jeffrey Jackson, Michael Kearns, Yishay Mansour, and Steven Rudich. Weakly learning DNF and characterizing statistical query learning using fourier analysis. In *The 26th Annual ACM Symposium on Theory of Computing*, pages 253 – 262, 1994.

[3] Merrick L. Furst, Jeffrey C. Jackson, and Sean W. Smith. Improved learning of AC^0 functions. In 4^{th} *Annual Workshop on Computational Learning Theory*, pages 317–325, August 1991.

[4] J. Jackson. An efficient membership-query algorithm for learning DNF with respect to the uniform distribution. In *Annual Symposium on Switching and Automata Theory*, pages 42 – 53, 1994.

[5] E. Kushilevitz and Y. Mansour. Learning decision trees using the fourier spectrum. In *Proceedings of the* 23^{rd} *Annual ACM Symposium on Theory of Computing*, pages 455–464, May 1991.

[6] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, fourier transform and learnability. In 30^{th} *Annual Symposium on Foundations of Computer Science, Reseach Triangle Park, NC*, pages 574–579, 1989.

[7] Y. Mansour. Learning Boolean Functions *via* the Fourier Transform. *Advances in Neural Computation*, edited by V.P. Roychodhury and K-Y. Siu and A. Orlitsky, Kluwer Academic Publisheres, 1994.

[8] Yishay Mansour. An $o(n^{\log\log n})$ learning algorihm for DNF under the uniform distribution. In *Workshop on Computational Learning Theory*, pages 53–61, July 1992.