0368.416701 Sublinear Time Algorithms January 20, 2015

Lecture 12

Lecturer: Ronitt Rubinfeld *Scribe: Nicholas Boffi, Gal Rotem*

1 Estimating the Sum of Powers of Degree One Fourier Coefficients

1.1 Reminders

Recall from the last lecture that, for any Boolean function $f : \{-1,1\}^m \to \{-1,1\}$, we have that

$$f(x) = \sum_{S\subset[n]} \hat{f}(s)\chi_S(x) \tag{1}$$

where the χ_S are given by

$$\chi_S(x) = \prod_{i\in S} x_i \tag{2}$$

and the Fourier coefficients $\hat{f}(S)$ are inner products:

$$\hat{f}(S) = \langle f, \chi_S\rangle = 2^{-n}\sum_x f(x)\chi_S(x). \tag{3}$$

1.2 Estimating Powers

Say we want to estimate the sum of powers of degree one Fourier coefficients. We say that a coefficient is "degree one" if $\hat{f}(S)$ is such that S is a singleton set, i.e, $S = \{i\}$ for some $i \in [n]$. Why might we be interested in doing this? One example is in testing if a function is a dictator, where such sums can be highly indicative that a function is determined by a single bit. Furthermore, the ability to estimate the sums of degree one powers will provide us with an iterative process to estimate sums of any degree. For example, if we can estimate sums of degree two and less, and we can estimate the sums of degree one, then we can subtract the two to find out the sum of degree two terms only. We will show how to perform such a process explicitly later when we prove the upcoming algorithm for degree one coefficients.

First, we proceed with some notation. Denote $\hat{f}(\{i\})$ by $\hat{f}(i)$. Then the question becomes, can we estimate:

$$\sum_{i=1}^{n} \hat{f}(i)^p \tag{4}$$

for some power p?

```
Data: A power $p$
1) Pick $x^{(1)}$, $x^{(2)}$, ... , $x^{(p-1)} \in \{\pm 1\}^n$
2) Pick a "noise vector" $\mu$ such that $\Pr[\mu_i = 1] = \frac{1}{2} + \frac{\eta}{2}$ and $\Pr[\mu_i = -1] = \frac{1}{2} - \frac{\eta}{2}$ for some constant $\eta < 1$.
3) Set $y = f(x^{(1)}) \cdot f(x^{(2)}) \cdot ... \cdot f(x^{(p-1)}) \cdot f(x^{(1)} \odot x^{(2)} \odot ... \odot \mu)$.
4) return y.
```

Algorithm 1: A "random process".

Claim 1 $E[y] = \sum_{S\subset[n]} \eta^{|S|}\hat{f}(S)^p$

Proof Note that we can write

$$E[y] = E[f(x^{(1)}) \cdot f(x^{(2)}) \cdot ... \cdot f(x^{(p-1)}) \cdot f(x^{(1)} \odot x^{(2)} \odot ... \odot \mu)]. \tag{5}$$

We can expand this as

$$E[y] = \sum_{S_1 S_2 ... S_p} \hat{f}(S_1)\hat{f}(S_2)...\hat{f}(S_p) E[\chi_{S_1}(x^{(1)})\chi_{S_2}(x^{(2)})...\chi_{S_{p-1}}(x^{(p-1)})\chi_{S_p}(x^{(1)} \odot x^{(2)} \odot ... \odot \mu)] \tag{6}$$

by writing each $f(x)$ in terms of its Fourier coefficients and using linearity of expectation. Recall now two useful relations from last lecture that $\chi_S(a \odot b) = \chi_S(a)\chi_S(b)$, and that $\chi_A(x)\chi_B(x) = \chi_{A\triangle B}(x)$ where $A\triangle B$ denotes the symmetric difference of A and B, $A\triangle B = (A - B) \cup (B - A)$. Then we have that:

$$E[\chi_{S_1}(x^{(1)})\chi_{S_2}(x^{(2)})...\chi_{S_{p-1}}(x^{(p-1)})\chi_{S_p}(x^{(1)} \odot x^{(2)} \odot ... \odot \mu)] = E[\chi_{S_1\triangle S_p}(x^{(1)})...\chi_{S_{p-1}\triangle S_p}(x^{(p-1)})\chi_S(\mu)] \tag{7}$$

where we have used the first of the two useful relations. Employing the second one to the product, we can now write

$$E[\chi_{S_1\triangle S_p}(x^{(1)})...\chi_{S_{p-1}\triangle S_p}(x^{(p-1)})\chi_S(\mu)] = \prod_{i=1}^{p-1} \left(E[\chi_{S_i\triangle S_p}(x^{(i)})] \right) E[\chi_S(\mu)]. \tag{8}$$

As in the last lecture, $E[\chi_{S_i\triangle S_p}(x^{(i)}) = 1$ for $S_i = S_p$ and 0 otherwise. Also, $E[\chi_S(\mu)] = E\left[\prod_{i\in S} \mu_i\right] = \prod_{i\in S} E[\mu_i] = \prod_{i\in S} \eta = \eta^{|S|}$. Putting this all together, we have:

$$E[y] = \sum_{S\subset[n]} \eta^{|S|} \hat{f}(S)^p. \tag{9}$$

■

Okay, so the claim is true, but why do we care? The point is that $\eta^{|S|}$ decays rapidly as $|S|$ increases. Thus, the *the terms that contribute most to the sum* are the Fourier coefficients of degree one and the Fourier coefficient of degree zero. With a proper choice of η, we can "tune" $E[y]$ so that we can use it for an approximation for degree one Fourier coefficients once we get rid of that pesky coefficient of degree zero. Here's how:

```
Data: A power $p$
Result: An estimation of the sum of Fourier coefficients of degree one to the power p.
1) Estimate $\omega = E[f(x^{(1)}...f(x^{(p)})] = \hat{f}(\emptyset)^p$ to additive $\pm\eta^2$. Note that $x^p = x^{(1)} \odot ... \odot x^{(p-1)} \odot \mu$ with $\eta = 0$ for this step.
2) Use Alg. 1 to estimate $E[y]$ with $y$ as before.
3) return $E[y] - \omega = \sum_{S\subset[n],|S|>0} \eta^{|S|}\hat{f}(S)^p = \gamma$.
```

First, note that $\langle f, \chi_\emptyset \rangle = \hat{f}(\emptyset) = \sum_x f(x)\frac{1}{2^n} = E_x[f(x)]$ by definition.

Claim 2 γ/η *is a good estimate of* $\sum_{|S|=1} \hat{f}(S)^p$.

Proof

$$\sum_{|S|=1} \eta^{|S|}\hat{f}(S)^p = \sum_{|S|>0} \eta^{|S|}\hat{f}(S)^p - \sum_{|S|>1} \eta^{|S|}\hat{f}(S)^p \tag{10}$$

$$\begin{aligned}
&\le \quad \gamma - \eta^2 \sum_{|S|>1} \hat{f}(S)^p && (11)\\
&\le \quad \gamma - \eta^2 \sum_{|S|>1} |\hat{f}(S)^p| && (12)\\
&\le \quad \gamma - \eta^2 \sqrt{\sum_{|S|>1} \hat{f}(S)^2} \cdot \sqrt{\sum_{|S|>1} (\hat{f}(S)^{(p-1)})^2} && (13)\\
&\le \quad \gamma - \eta^2 \cdot (1) \cdot \sqrt{\sum_{|S|>1} \hat{f}(S)^2} && (14)\\
&\le \quad \gamma - \eta^2 \cdot (1) \cdot (1) && (15)\\
&\le \quad \gamma - \eta^2 && (16)
\end{aligned}$$

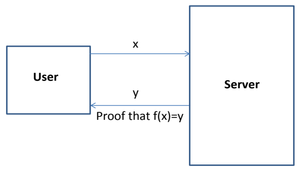
Where we have used that the sum of positive and negative numbers is less than the sum of their absolute values to go from (11) to (12), Cauchy-Schwartz to go from (12) to (13), Boolean Parseval's to reduce the first square root from (13) to (14) and noting that $\hat{f}(S)^{p-1} \le \hat{f}(S)^2$ for $p \ge 2$ to reduce the second square root. Finally, we employ Boolean Parseval's again to go from (14) to (15). Now note that $\sum_{|S|=1} \eta^{|S|}\hat{f}(S)^p = \sum_{|S|=1} \eta\hat{f}(S)^p$ Dividing both sides by η, we have:

$$\sum_{|S|=1} \eta\hat{f}(S)^p \le \frac{\gamma}{\eta} - \eta. \tag{17}$$

■

2 Interactive Proofs

Assume we have some user $\mathcal{U}$ who wants to compute a function f on an input x. Furthermore assume the user is computationally bounded and cannot compute $f(x)$ on his own, but can outsource the computation to a server $\mathcal{S}$. S will return y to the user and claims that $y = f(x)$. To prevent "cheating", $\mathcal{U}$ demands that $\mathcal{S}$ sends both y and some reliable proof that $y = f(x)$.



Example 1 *$\mathcal{U}$ owns a website, and a company $\mathcal{C}$ claims that at least k clicks were made through their website to enter $\mathcal{U}$'s website. We assume that given a "description" x of a click, $\mathcal{U}$ can efficiently (in constant time) verify that x is indeed a valid click.*

$\mathcal{U}$'s goal is to determine if the number of valid clicks is greater than k given the proof provided by $\mathcal{U}$. We ask that $\mathcal{U}$ return "PASS" with high probability in such a case, and if the number of valid clicks is less than $(1-\epsilon)k$ return "FAIL" with high probability.

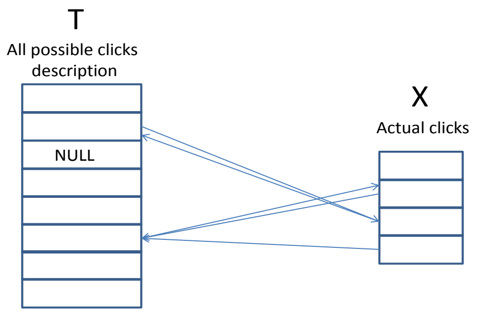
For our first try, we assume that $\mathcal{C}$ sends $\mathcal{U}$ a "big array" X with a description of all k clicks: $X = x_1, x_2, ..., x_k$. Then $\mathcal{U}$ samples $O(1/\epsilon)$ x_i's and checks if there are less then $k\epsilon/2$ invalid click descriptions. If so, he says "PASS", otherwise he returns "FAIL". Note that in this simple protocol, $\mathcal{C}$ can easily fool $\mathcal{U}$ by sending the same description of a click multiple times. Thus we need to also check if there are at most $k\epsilon/2$ duplicate click descriptions.

We need to modify the proof $\mathcal{C}$ sends to $\mathcal{U}$, and we will assume that we don't care how "big" the proof is.

$\mathcal{C}$ will build a table T with all possible clicks description, $t_1, t_2, ..., t_n$ and send both tables T and X, where X is the array $x_1, ...x_k$. $\mathcal{C}$ will also send forward and back pointers both from T to X and from X to T in the following way:

1. For each cell $t_i \in T$, t_i is some click's possible description. If we have such a click in table X, then t_i will hold a pointer to that cell in X. Thus, we have a pointer from t_i to x_j if x_j's click description is exactly t_i. If we don't have a click in X with t_i's description, then t_i will hold a NULL pointer.

2. For each cell in $x_j \in X$, we will hold a back-pointer to the appropriate cell t_i such that t_i is the description of click x_j.



Now $\mathcal{U}$ will verify the proof with the following procedure:

```
1) Repeat $O(1/\epsilon)$ times:
1.1) pick $j \in [k]$ at random
1.2) $l \leftarrow X[j]$
1.3) if $T(l) = x_j$ continue, otherwise **return** $FAIL$
2) **return** $PASS$
```

Algorithm 2: Verifying the proof

The intuition is that if there are any duplicates in X, then T won't "know" on which x_i to point. Thus, we can easily see that if there are $\geq k\epsilon/2$ duplicates in X, then in each round we fail with probability $\geq \epsilon/2$, so in $O(1/\epsilon)$ rounds, we will catch a false proof with constant probability.

Example 2 *Consider the graph problem MAX-CUT. In this problem, we are given a graph, $G = (V, E)$ and we are interested in whether there exists a cut in the graph with at least k crossing edges, this problem is known to be $\mathcal{NP}$-complete.*

We are only interested in verifying a solution: in such a case, some prover, $\mathcal{S}$, declares that he found such a cut. We can verify his proof in sublinear-time using the previous example: this time T serves as a boolean array of size $|V|$, and each cell indicates on which side of the cut the corresponding vertex is in. The table X is of size k as before, and holds all the crossing edges of the cut. A simple verifier can use the following algorithm to check for the proof's correctness:

```
1) Repeat $O(1/\epsilon)$ times:
1.1) pick an edge $(v_i, v_j) \in [k]$ at random
1.2) query $T$ for both vertices $v_i, v_j$
1.3) if $T(v_i) = T(v_j)$ **return** $FAIL$ (they are on the same side of the cut), otherwise- continue
2) **return** $PASS$
```

Algorithm 3: Verifying max-cut

As before, it is clear that with high probability we accept only correct proofs.

Example 3 *Consider the Bin-Packing problem: we are given a positive integer B, a set of n positive elements $x_1, x_2, ..., x_n$ where each $x_i \in [B]$, and k bins of size B. We want to know whether there exists a legal packing of these elements into the k bins, i.e, each element is associated with a bin and the total size of the elements in each bin is less than B. This is also a well-known $\mathcal{NP}$-complete problem.*

As before, we are interested in verifying a solution for this problem in sublinear-time. We want an algorithm that returns "PASS" with probability 1 on all correct proofs, and if at most $(1-\epsilon)n$ of the elements fit, returns "FAIL" with high probability. Consider the following proof: We will have k arrays (one for each bin) $A_1, ..., A_k$ and each array will be of size B. If x_i is of weight w and appears in bin A_j, then A_j will contain w consecutive instances of x_i. In addition, we will have an extra array X of size n such that $X[j]$ indicates the number of bins i in which x_j is packed and the offset m in A_i at which x_j starts to appear w consecutive times.

To verify this proof, the verifier will use the following procedure:

```
1) Repeat $O(1/\epsilon)$ times:
1.1) pick an element $x_i$ of size $w$ , $i \in n$ at random
1.2) query $X[i]$ to get the number of bin $j$, and the offset $m$, for $x_i$.
1.3) verify that $x_i$ appears $w$ consecutive times in $A_j$ starting at index $m$, if it is not - **return**
$FAIL$, else-continue
2) **return** $PASS$
```

Algorithm 4: Verifying bin-packing

A problem may arise: it could be that x_i is "heavy-weighted", and so checking w consecutive indices could be costly. A possible solution could be to use a monotonicity tester, but we won't elaborate any further.