15-850(C) Machine Learning Theory

Lecturer: Avrim Blum — Lectures 15 and 16: November 7 and 9, 1995

Scribe: Carl Burch

So far we have been considering a learning model in which the learner plays a passive role. The learner receives examples and processes them. Frequently in real life, though, we have the capability to manipulate the environment to determine what happens under specific conditions. It would be nice to include this concept of manipulation in our model.

1 Membership queries

One could concoct different ways of modeling this type of manipulation, but the easiest for us is to extend the capability allowed before in a simple manner by allowing an algorithm to ask what is called a *membership query*. A membership query is a question of the form, "What is the classification of example x?", where x is chosen by the learning algorithm, to which an oracle answers positive or negative.

A natural form of learning that follows from this is *learning in the mistake-bound model with membership queries*. In this model, in each round the learning algorithm is given the following choice. It may either receive an unlabeled example from the world and make a prediction as in the standard mistake-bound model, or else it may choose to make a membership query. The algorithm is charged for each mistake and for each query. An algorithm is said to learn with membership queries in the mistake-bound model if, no matter what the target concept and the sequence of inputs to the algorithm, the number of mistakes made by the algorithm plus the number of membership queries asked by the algorithm is polynomially bounded by the size of the target concept and the size of an example.

2 Equivalence queries

Another possible query type is an *equivalence query*, which is a question of the form, "Is there an example on which hypothesis h fails?", to which an oracle answers, "There are none," or gives an example x on which $h(x) \neq c(x)$.

The equivalence query can be thought of as another way of viewing the mistake-bound model. Given an algorithm $\mathcal{A}$ that works with equivalence queries, we can construct an algorithm $\mathcal{B}$ learning in the mistake bound model which gets a hypothesis h from $\mathcal{A}$ and tests h on examples, returning to $\mathcal{A}$ the first example on which h fails, whereupon it receives from $\mathcal{A}$ a new value for h. Once $\mathcal{A}$ proposes a correct hypothesis, $\mathcal{B}$ will make no more errors. Algorithm $\mathcal{B}$ will make the same number of mistakes as the number of queries $\mathcal{A}$ asks before achieving the correct hypothesis.

Conversely, given an algorithm $\mathcal{B}$ learning in the mistake bound model, we can construct an algorithm $\mathcal{A}$ learning using equivalence queries if $\mathcal{A}$ has access to $\mathcal{B}$'s internal state.

For this reason *learning with membership queries and equivalence queries* is the same as learning with membership queries in the mistake bound model. Though adding equivalence queries doesn't offer a real gain over using the mistake-bound model, it is sometimes easier to think about, so we will tend to use learning with membership queries and equivalence queries.

3 Learning monotone and general DNF formulas

To investigate the relationship between this active model of learning and the passive model we examined before, we return to our well-worn example of learning DNFs.

In the passive models, learning monotone DNFs is just as difficult as learning general DNFs. This can be seen by the following reduction. Say we have an algorithm $\mathcal{A}$ that learns monotone DNFs only. We can construct an algorithm $\mathcal{B}$ that learns general DNFs in the following way. Say that $\mathcal{B}$ receives an example $x_0 x_1 \ldots x_{n-1}$. Feed to $\mathcal{A}$ the example $x_0 \overline{x_0} x_1 \overline{x_1} \ldots x_{n-1} \overline{x_{n-1}}$. This example allows $\mathcal{A}$ to act on negations of variables in x since they appear positive in the example it receives. The prediction of $\mathcal{B}$ will be the same as $\mathcal{A}$'s prediction.

This reduction fails with queries, since $\mathcal{A}$ may ask something that cannot be transformed back to regular space. In particular, a membership query may ask about an example in which the first and second variables of the strings it receives (corresponding to x_0 and $\overline{x_0}$) are both true.

In fact, learning monotone DNF formulas turns out to be easy to do with membership queries (described below) but learning general DNF formulas with queries has been shown to be as hard as learning general DNF formulas without queries, modulo cryptographic assumptions, by Angluin and Kharitonov [1].

4 Actively learning monotone DNFs

Let's define the flip operation $x \updownarrow i$ on a bit string $x_0 \ldots x_n$ and an integer i as $x_0 \ldots x_{i-1} \overline{x_i} x_{i+1} \ldots x_n$.

The following algorithm will learn monotone DNF formulas:

FIND-MONOTONE-DNF
$h \leftarrow$ **False**;
loop begin
    $x \leftarrow$ equivalence-query(h); (Note: x is a positive example)
    if $x =$ none **then terminate**;
    $\hat{x} \leftarrow x$;
    while for some i, $\hat{x}_i = 1$ and membership-query($\hat{x} \updownarrow i$) returns positive, **do** $\hat{x} \leftarrow \hat{x} \updownarrow i$;
    $T \leftarrow$ conjunction of variables corresponding to bits set to 1 in $\hat{x}$;
    $h \leftarrow$ disjunction of h and T;
end

The internal **while** loop of this procedure we call *walking x toward $\vec{0}$*. More generally, walking x toward z would be identical to the **while** loop occuring above, except that the $x_i = 1$ term would be replaced by $x_i \neq z_i$.

Each equivalence query will return a positive counterexample to h. We can see this by considering an arbitrary example y that h says is positive. This means that it satisfies a term in h. By the construction of h, this term would come from a positive example $\hat{x}$ in which every bit set to 1 in $\hat{x}$ is also set to 1 in y. Since the target function is monotone, the fact that $\hat{x}$ is a true positive example implies that y is also a true positive example. Hence, the only counterexamples are those on which h says negative (but are really positive).

Every term in h is a minterm (i.e., a subset) of some term in the target function c by construction (if it wasn't, then we would have been able to walk the example further towards $\vec{0}$). Therefore, the number of equivalence queries is bounded by the number of terms in c. Moreover, since the number of membership queries for each term in h is at most n, the number of membership queries is bounded by the product of n and the number of terms in c.

5 Learning functions with small DNF and CNF representations

We now turn to a result of Bshouty's that gives an algorithm to learn functions which have both a small DNF representation and a small CNF representation [2]. This class includes those functions with small decision tree representations, since any decision tree can be made into a DNF by ORing together terms corresponding to the positive leaves, or into a CNF by ANDing together clauses corresponding to the negations of the negative leaves.

5.1 Defining Make-DNF

To learn a general DNF formula, we will first define the function Make-DNF(z), in which z is a negative example. This function uses membership and equivalence queries but can only make use of *positive* counterexamples to its equivalence queries. The final algorithm will use this function, and will make use of the negative counterexamples at an outer level. (It will use them to invoke new copies of Make-DNF.)

Make-DNF(z)
$h \leftarrow$ **False**;
loop begin
  $x \leftarrow$ equivalence-query(h);   We will assume that x is a *positive* counterexample.
  $\hat{x} \leftarrow x$;
  while for some i, $\hat{x}_i \neq z_i$ and membership-query($\hat{x} \updownarrow i$) returns positive, **do** $\hat{x} \leftarrow \hat{x} \updownarrow i$;
  $T \leftarrow$ conjunction of literals satisfied by $\hat{x}$ but not by z;
  $h \leftarrow$ disjunction of h and T;
end

As you can see, this algorithm is closely related to the algorithm for learning monotone DNF formulas. What we can show about this algorithm, however, will be less impressive than what we showed earlier.

It is relatively obvious that z satisfies no term in h. In fact, z doesn't even satisfy any literal inside any term in h. The reason is that each term T in h was constructed as a conjunction of literals satisfying a positive example $\hat{x}$ but not satisfying z. (And, the fact that $\hat{x}$ is positive but z is negative means that there is at least one literal satisfying $\hat{x}$ but not z, so T will not be null.)

Less obviously, each term T in h is a subterm of some term T' in the target function c (i.e., each literal in T appears in T'). The reason is as follows. After the **while** loop, $\hat{x}$ is a positive example and therefore satisfies some term T' of c. In addition, for each i such that $x_i \neq z_i$, $\hat{x} \updownarrow i$ is a negative example (otherwise we would have flipped that bit). This means that for all i such that $\hat{x}_i \neq z_i$, which correspond to the literals that are placed into T, we know that $\hat{x} \updownarrow i$ does not satisfy c and so does not satisfy T'. Hence the literal corresponding to bit i of $\hat{x}$ appears in T'. This means that T will be a subterm of T'.

In fact, it is not hard to see that at the time the term T described above is placed into h, there is no *other* term in h that is also a subterm of T'. The reason is that the counterexample x does not satisfy any terms of h (by definition), and as $\hat{x}$ is walked towards z it continues to not satisfy any terms of h (since, as we noted, z does not satisfy any literals of any terms of h). Therefore, the term T' cannot contain any term of h as a subterm (otherwise $\hat{x}$ would satisfy it). What this means is that the number of terms in h will never exceed the number of terms in c. So the number of equivalence queries is bounded by the number of terms in c and the number of membership queries is bounded by the product of n^2 and the number of terms in c.

5.2 Using Make-DNF

Now we can tackle a general DNF formula using several instances of Make-DNF. We can imagine this routine handling the equivalence queries of each of these instances, feeding positive counterexamples to the proper instance of Make-DNF and generating new instances of Make-DNF with negative counterexamples.

Find-DNF
$H \leftarrow \emptyset$;
loop begin
    $x \leftarrow$ equivalence-query(conjunction of hypotheses of each instance of Make-DNF within H);
    if $x =$ none **then terminate**;
    if x is a positive counterexample **then**
        $M \leftarrow$ an instance of Make-DNF in H whose hypothesis x satisfies;
        Use x as a positive counterexample to the equivalence query in M;
    else
        $H \leftarrow H \cup$ Make-DNF(x);
end

Since we have already shown that each instance of MAKE-DNF uses a polynomial number of queries to converge to the correct solution, we need only to show that a polynomial of negative counterexamples might occur. This will occur when the target function has a polyomial-size CNF as well as DNF representation. To show this, we would like to introduce a couple of definitions.

Define subcube(y,z) as the set of all points x such that $(\forall i) y_i = z_i \Rightarrow x_i = y_i$. I.e., it is the set of all points in the subcube having y and z at opposite corners. Also say that a negative example z *covers* a negative example y if every x in subcube(y,z) is a negative example. (With monotone DNFs, $\vec{0}$ covers all negative examples, since given a negative example y each $x \in$ subcube($\vec{0}$,y) has all the zero bits of y plus possibly more.)

Let us say that our oracle gives us a negative counterexample y. This means that no previous negative counterexample z covers y for the following reason. Let z be a negative counterexample, and say y is a negative counterexample received later. Since y is positive according to h, it must satisfy the hypothesis of each instance of MAKE-DNF in H, and in particular it satisfies the hypothesis of MAKE-DNF(z). Let T be one of the terms in this hypothesis satisfied by y. This term is the conjunction of literals satisfied by some positive example x but not by z. Therefore, x lies in subcube(y,z) because y differs from z in every position that x does (that is, in all literals of T).

So, once the algorithm receives a negative example y it will never receive any other negative examples covered by y.

To understand what this means, consider the case of a decision tree. When we receive a counterexample that goes to a negative leaf of the tree, a new instance of MAKE-DNF is spawned. This example covers all the negative examples going to the same leaf, so the number of instances of MAKE-DNF in H is bounded by the number of negative leaves in the decision tree.

More generally, think of writing the complement of the target function, $\bar{c}$, as a DNF formula. A negative example z for c is a positive example of $\bar{c}$ and must satisfy some term T of $\bar{c}$. It is not hard to see that example z covers any other example that also satisfies T. Therefore, the number of copies of MAKE-DNF is at most the number of terms in the DNF expression of $\bar{c}$, or equivalently, the number of clauses in the CNF expression of c. So, the running time of the algorithm is polynomial in the product of the DNF-size and the CNF-size of the target function.

6 Summary

We have described learning with queries and have seen that queries allow for a very simple algorithm to learn monotone DNF formulas (which in the passive model are as hard to learn as general DNF formulas). We have also seen a somewhat more complicated algorithm to learn functions with small (polynomial size) DNF and CNF representations. This latter class includes the class of functions that have polynomial-size decision tree representation.

References

[1] D. Angluin and M. Kharitonov. When won't membership queries help? In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, pages 444–454, New Orleans, May 1991.

[2] N. H. Bshouty. Exact learning via the monotone theory. In *Proceedings of the IEEE Symp. on Foundation of Computer Science*, pages 302–311, Palo Alto, CA., 1993.