LEARNING CARTESIAN-PRODUCTS OF LEARNABLE SETS

1. Introduction

Traditional machine learning focuses on learning patterns from sets of (sometimes labelled) examples. This is useful for learning approximations of concepts. However, for logical structures such as automata or propositional formulas, slight changes can result in very different behaviors. It is not generally possible to exactly learn these structures from just labelled examples.

Therefore, researchers have introduced the active learning model, where the learner is allowed to make queries about the target concept to an oracle. Using the correct set of oracles can result in the polynomial time learnability of otherwise unlearnable sets []

Recently, exact active learning has been applied to formal synthesis, where a program is automatically generated to fit a high-level specification []. This has been particularly useful in Counter-Example Guided Inductive Synthesis []

2. Important Notation

In the following proofs, we assume we are given concept classes $C_1, C_2, \ldots, C_k$ defined over sets X_1, X_2, $\ldots$, X_k. Each c_i in each C_i is learnable from algorithm A_i using queries to an oracle that can answer any queries in a set Q. For each query $q \in Q$, we say algorithm A_i makes $\#q(c)$ many q queries to the oracle in order to learn concept c, dropping the index i when necessary . We replace the term $\#q$ with a more specific term when the type of query is specified. For example, an algorithm A might make $\#Mem(c)$ many membership queries to learn c.

Q is almost always a singleton. Should we just call it a query instead of a set?

Unless otherwise stated, we will assume any index i or j ranges over the set $\{1 \ldots k\}$. We write $\prod S_i$ to refer to the cartesian-product of sets S_i. Note that this is the k-ary cartesian

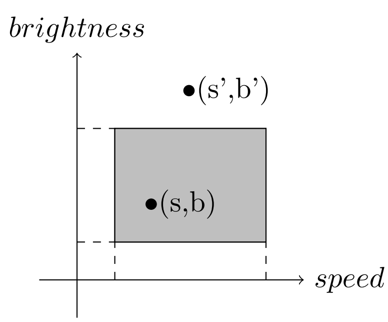

Figure 1. The correct range of lighthouse behaviors.

Query Name	Symbol	Complexity	Oracle Definition
Single Positive Query	$1Pos$	$\#1Pos(c)$	Return a fixed $x \in c^*$
Positive Query	Pos	$\#Pos(c)$	Return an $x \in c^*$ that has not yet been given as a positive example (if one exists)
Membership Query	Mem	$\#Mem(c)$	Given string s, return true iff $s \in c^*$
Equivalence Query	EQ	$\#EQ$	Given $c \in C$, return true if $c = c^*$ otherwise return $x \in (c \backslash c^*) \cup (c^* \backslash c)$
Subset Query	Sub	$\#Sub(c)$	Given $c \in C$, return 'true' if $c \subseteq c^*$ otherwise return some $x \in c \backslash c^*$
Superset Query	Sup	$\#Sup(c)$	Given $c \in C$, return 'true' if $c \supseteq c^*$ otherwise return some $x \in c^* \backslash c$

product, and is not simply repeated applications of the binary cartesian product. So for example $\prod_{i=1}^{3} S_i$ equals $\{(s_1, s_2, s_3) \mid s_1 \in S_1, s_2 \in S_2, s_3 \in S_3\}$ and not $\{((s_1, s_2), s_3) \mid s_1 \in S_1, s_2 \in S_2, s_3 \in S_3\}$ We use S^k to refer to $\prod_{i=1}^{k} S$.

We use vector notation $\vec{x}$ to refer to a vector of elements $(x_1, \ldots, x_k)$, $\vec{x}[i]$ to refer to x_i, and $\vec{x}[i \leftarrow x_i']$ to refer to $\vec{x}$ with x_i' replacing value x_i at position i. We define $\prod C_i := \{\prod c_i \mid c_i \in C_i, i \in \{1, \ldots, k\}\}$. We write $\vec{c}$ for any element of $\prod C_i$ and will often denote $\vec{c}$ by $(c_1, \ldots, c_k)$ in place of $\prod c_i$.

The results below answer the following question: For what set of queries Q does the learnability of each C_i imply the learnability of $\prod C_i$ and how does the number of queries to learn $\prod C_i$ increase as a function of each $\#q_i(c_i)$ for each $q \in Q$?

The proofs in this paper make use of the following simple observations

Observation 1. *For sets $S_1, S_2, \ldots, S_k$ and $T_1, T_2, \ldots, T_k$, we have $\prod S_i \subseteq \prod T_i$ if and only if $S_i \subseteq T_i$ for all i or $\prod S_i = \emptyset$.*

Observation 2. *Fix sets $S_1, S_2, \ldots, S_k$, points $x_1, x_2, \ldots, x_k$ and an index i. If $x_j \in S_j$ for all $j \neq i$, then $(x_1, x_2, \ldots, x_k) \in \prod S_i$ if and only if $x_i \in S_i$.*

3. Negative Results

This section introduces some fairly simple lower bounds.

We will start with a lower-bound on learnability from positive examples.

Proposition 1. *There exist concepts C_1 and C_2 that are each learnable from constantly many positive queries, such that $C_1 \times C_2$ is not learnable from any number of positive queries.*

Proof. Let $C_1 := \{\{a\}, \{a, b\}\}$ and set $C_2 := \{\mathbb{N}, \mathbb{Z} \backslash \mathbb{N}\}$. To learn the set in C_1, pose two positive queries to the oracle, and return $\{a, b\}$ if and only if both a and b are given as positive examples. To learn C_2, pose one positive query to the oracle and return $\mathbb{N}$ if and only if the positive example is in $\mathbb{N}$. An adversarial oracle for $C_1 \times C_2$ could give positive examples only in the set $\{a\} \times \mathbb{N}$. Each new example is technically distinct from previous examples, but there is no way to distinguish between the sets $\{a\} \times \mathbb{N}$ and $\{a, b\} \times \mathbb{N}$ from these examples. $\square$

Now we will show lower bounds on learnability from *EQ*, *Sub*, and *Mem*. We will see later that this lower bound is tight when learning from membership queries, but not equivalence or subset queries.

Proposition 2. *There exists a concept C that is learnable from $\#q$ many queries posed to $Q \subseteq \{\mathrm{Mem}, \mathrm{EQ}, \mathrm{Sub}\}$ such that learning C^k requires $(\#q)^k$ many queries.*

Should I explicitly handle infinite and finite cases separately? Should I include bigO notation on the infinite case?

Proof. Let $C = \{\{j\} \mid j \in \{0 \ldots m\}\}$.

We can learn C in m membership, subset, or equivalence queries by querying $j \in c^*$, $\{j\} \subseteq c^*$, or $\{j\} = c^*$, respectively.

However, a learning algorithm for C^k requires more than m^k queries. To see this, note that C^k contains all singletons in a space of size $(m+1)^k$.

So for each subset query $\{x\} \subseteq c^*$, if $\{j\} \neq c^*$, the oracle will return j as a counterexample, giving no new information. Likewise, for each equivalence query $\{j\} = c^*$, if $\{j\} \neq c^*$, the oracle can return j as a counterexample. Therefore, any learning algorithm must query $x \in c^*$, $\{x\} \subseteq c^*$, or $\{x\} = c^*$ for $(m+1)^k - 1$ values of x $\square$

4. Positive Results

Proposition 3. *If $Q = \{\mathrm{Sup}\}$, then there is an algorithm that learns any concept $\vec{c}^* = c_1^* \times \cdots \times c_k^* \in \prod C_i$ in $\sum \#\mathrm{Sup}(c_i^*)$ queries.*

Proof. Algorithm 1 learns $\prod C_i$ by simulating the learning of each A_i on its respective class C_i. The algorithm asks each A_i for superset queries S_i, queries the product $\prod S_i$ to the oracle, and then uses the answer to answer at least one query to some A_i. Since at least one A_i receives an answer for each oracle query, at most $\sum \#\mathit{Sup}(c_i^*)$ queries must be made in total.

We will now show that each oracle query results in at least one answer to an A_i query (and that the answer is correct). The oracle first checks if the target concept is empty, if not it proceeds as normal. At each step, the algorithm poses query $\prod S_i$ to the oracle. If the oracle returns 'yes' (meaning $\prod S_i \supseteq \vec{c}^*$), then $S_i \supseteq c_i^*$ for each i by Observation 1, so the oracle answers 'yes' to each A_i. If the oracle returns 'no', it will give a counterexample $\vec{x} = (x_1, \ldots, x_k) \in \vec{c}^* \setminus \prod S_i$. There must be at least one $x_i \notin S_i$ (otherwise, $\vec{x}$ would be in $\prod S_i$). So the algorithm checks $x_j \in S_j$ for all x_j until an $x_i \notin S_i$ is found. Since $\vec{x} \in \vec{c}^*$, we know $x_i \in c_i^*$, so $x_i \in c_i^* \setminus S_i$, so the oracle can pass x_i as a counterexample to A_i.

Note that once A_i has output a correct hypothesis c_i, S_i will always equal c_i, so counterexamples must be taken from some $j \neq i$. $\square$

Result: Learn $\prod C_i$ from Superset Queries
if $\emptyset \in C_i$ *for some* i **then**
  Query $\emptyset \supseteq \vec{c}^*$;
  if $\emptyset \supseteq \vec{c}^*$ **then**
    return $\emptyset$
for $i = 1 \ldots k$ **do**
  Set S_i to initial subset query from A_i
while *Some* A_i *has not completed* **do**
  Query $\prod S_i$ to oracle;
  if $\prod S_i \supseteq c^*$ **then**
    Answer $S_i \supseteq c_i^*$ to each A_i;
    Update each S_i to new query;
  else
    Get counterexample $\vec{x} = (x_1, \ldots, x_k)$ **for** $i = 1 \ldots k$ **do**
      if $x_i \notin S_i$ **then**
        Pass counterexample x_i to A_i;
        Update S_i to new query;
  for $i = 1 \ldots k$ **do**
    if A_i *outputs* c_i **then**
      Set $S_i := c_i$;
return $\prod c_i$;

Algorithm 1: Algorithm for learning from Subset Queries

combine learning algorithm for sub, mem, and eq w/ one positive query

Proposition 4. *If* $Q = \{\mathrm{Mem}\}$ *and a single positive example* $\vec{p} \in \vec{c}^*$ *is given, then* $\vec{c}^*$ *is learnable in* $k \cdot \sum \#\mathrm{Mem}_i(c_i^*)$ *membership queries.*

Proof. Algorithm 2 learns by simulating each A_i in sequence, moving on to A_{i+1} once A_i returns a hypothesis c_i. For any membership query M_i made by A_i, $M_i \in c_i^*$ if and only if $\vec{p}[i \leftarrow M_i] \in \vec{c}^*$ by Observation 2. Therefore the algorithm is successfully able to simulate the oracle for each A_i, yielding a correct hypothesis c_i. $\square$

Result: Learn $\prod C_i$ from Membership Queries and One Positive Example
Get positive example $\vec{p}$;
for $i = 1 \ldots k$ **do**
  while *A_i has not returned a hypothesis c_i* **do**
    Get membership query M_i from A_i;
    Query $\vec{p}[i \leftarrow M_i]$ to oracle;
    if *Oracle returns 'yes'* **then**
      Pass answer $M_i \in c_i^*$ to A_i;
    else
      Pass answer $M_i \notin c_i^*$ to A_i;
return $\prod c_i$;

Algorithm 2: Algorithm for learning from Membership Queries and One Positive Example

Proposition 5. *If $Q = \{\text{Sub}\}$ and a single positive example $\vec{p} \in \vec{c}^*$ is given, then $\vec{c}^*$ is learnable in $\sum \#\text{Sub}_i(c_i^*)$ subset queries and $k \cdot \sum \#\text{Sub}_i(c_i^*)$ membership queries.*

Proof. The learning process is described in Algorithm **??**. For each subset query $\prod S_i \subseteq \vec{c}^*$, the algorithm either returns 'yes' or gives a counterexample $\vec{x} = (x_1, \ldots, x_k) \in \prod S_i \backslash \vec{c}^*$. If the algorithm returns 'yes', then by Observation 1 $S_i \subseteq c_i^*$ for all i, so the algorithm can return 'yes' to each A_i. Otherwise, $\vec{x} \notin \vec{c}^*$, so there is an i such that $x_i \notin c_i^*$. By Observation 2 the algorithm can query $\vec{p}[j \leftarrow x_j]$ for all j until the $x_i \notin c_i^*$ is found.

Once the correct c_j is found for any j, S_j will equal c_j for all future queries, so any counterexamples must fail on an $i \neq j$.

Each subset query results in a correct answer being given to at least one learner A_i and at most k membership queries are made per subset query, yielding the desired bound on queries. □

```
Result: Learn ∏ C_i from Equivalence (or Subset) Queries, Membership Queries, and
        One Positive Example
for i = 1 ... k do
 | Set S_i to initial equivalence query from A_i
Query ∏ S_i to oracle;
if Q = {EQ} and ∏ S_i = c* then
 | return ∏ S_i ;
else
 | Get counterexample x = (x_1, ..., x_k);
 | if Q = {EQ} and x ∈ c* \ ∏ S_i then
 |  | for i = 1 ... k do
 |  |  | if x_i ∉ S_i then
 |  |  |  | Pass counterexample x_i to A_i;
 |  |  |  | Update S_i to new query from A_i;
 | else
 |  | for i = 1 ... k do
 |  |  | Query p[i ← x_i] ∈ c*;
 |  |  | if p[i ← x_i] ∉ c* and x_i ∈ S_i then
 |  |  |  | Pass counterexample x_i to A_i;
 |  |  |  | Update S_i to new query from A_i;
```

Algorithm 3: Algorithm for learning from Equivalence (or Subset) Queries, Membership Queries, and One Positive Example

Finally, we study the case when $Q = \{EQ\}$, as described in Algorithm 3. This algorithm works as a synthesis of the learning algorithms for Supersets and Subsets. When a negative example is given, the algorithm runs as in Algorithm **??** for handling subset queries. When a positive example is given, the algorithm runs as in Algorithm 1 for handling superset queries.

5. Disjoint Union

This section discusses learning disjoint unions of concept classes. This is generally much easier than learning cross-products of classes, since counterexamples belong to a single dimension in the disjoint union. This problem uses the same notation as the cross-product case, but we denote the disjoint union of two sets as $A \cupdot B$ and the disjoint union of many sets as $\bigcupdot A_i$. We define the concept class of disjoint unions as $C_{\cupdot} := \{\bigcupdot c_i \mid c_i \in C_i\}$.

The algorithm for learning from membership queries is very easy and won't be stated here. Algorithm 4 shows the learning procedure for when $Q \in \{\{Sub\}, \{Sup\}, \{EQ\}\}$. The correctness of this algorithm follows from the following simple facts. Assume we have sets $S_1, \dots, S_k$ and $T_1, \dots, T_k$. Then $\bigcupdot S_i \subseteq \bigcupdot T_i$ if and only $S_i \subseteq T_i$ for all i. Likewise $\bigcupdot S_i = \bigcupdot T_i$ if and only if $S_i = T_i$ for all i.

Result: Learning Disjoint Unions
for $i = 1 \dots k$ **do**
  Set S_i to initial query from A_i
while *Some A_i has not terminated* **do**
  Query $\biguplus S_i$ to oracle;
  if *Oracle returns 'yes'* **then**
    Pass 'yes' to each A_i;
    Get updated S_i from each A_i;
  else
    Get counterexample $x_i \in X_i$ for some i;
    Pass x_i as counterexample to A_i;
    Get updated S_i from each A_i;
return $\biguplus S_i$;

Algorithm 4: Learning Disjoint Unions

6. Learning with Only Membership Queries

We have seen that learning with membership queries can be made significantly easier if a single positive example is given. In this section we describe a learning algorithm using membership queries when no positive example is given. This algorithm makes $O(max_i\{\#Mem_i(c_i)\}^k)$ queries, matching the lower bound given in a previous section.

For this algorithm to work, we need to assume that $\emptyset \notin C_i$ for all i. If not, there is no way to distinguish between an empty and non-empty concept. For example consider the classes $C_1 = \{\{1\}, \emptyset\}$ and $C_2 = \{\{j\} \mid j \in \mathbb{N}\}$. It is easy to know when we have learned the correct class in C_1 or in C_2 using membership queries. However, for any finite number of membership queries, there is no way to distinguish between the sets $\emptyset$ and $\{(1, j)\}$ for some j that has yet to be queried.

The main idea behind this algorithm is that learning from membership queries is easy once a single positive example is found. So the algorithm runs until a positive example is found from each concept or until all concepts are learned. If a positive example is found, the learner can then run Algorithm 3 for learning from membership queries and a single positive example.

Proposition 6. *Algorithm 5 will terminate after making $O(max_i\{\#\mathrm{Mem}_i(c_i)\}^k)$ queries.*

Proof. The algorithm works by constructing sets S_i of elements and querying all possible elements of $\prod S_i$. We will get our bound of $O(max_i\{\#Mem_i(c_i)\}^k)$ by showing the algorithm will find a positive example once $|S_i| > max_i\{\#Mem_i(c_i)\}$ for all i. Since the algorithm queries all possible elements of $\prod S_i$, it is sufficient to prove that S_i will contain an element of c_i once $|S_i| > \#Mem_i(c_i)$.

Assume that each learner eventually terminates. Let $\vec{q^i} = q^i_1, q^i_2, \dots$ be the membership queries A_i makes assuming it only receives negative answers from an oracle. If $\vec{q^i}$ is finite, then there is some set $N_i \in C_i$ that A_i outputs after querying all points in $\vec{q^i}$ (and receiving negative answers). If N_i is non-empty let n_i be some element in N_i. Note that although sampling elements from a set might be expensive in general, this is only done for N_i and can therefore be hard-coded into the learning algorithm. If $c_i = N_i$, then by our assumption that $\vec{c}^* \neq \emptyset$, N_i contains some n_i. So S_i contains an element of c_i at the start of the algorithm. If $c_i \neq N_i$, by our assumption that A_i eventually terminates, A_i must eventually query some $q^i_j \in c_i$. So after j steps, S_i contains some element of c_i. Since $j < \#Mem_i(c_i)$, we have that S_i contains a positive example once $|S_i| > \#Mem_i(c_i)$, completing the proof. $\square$

```
Result: Learning with Membership Queries Only
for i = 1...k do
    if N_i and n_i exist then
        Set S_i := {n_i};
    else
        Set S_i := {};
Set j = 0;
while True do
    for i = {1, ..., k} do
        if |q^i| ≥ j then
            S_i := S_i ∪ {q^i_j};
    for x ∈ ∏ S_i do
        Query x ∈ c*;
        if x ∈ c* then
            Run Algorithm 2 using x as a positive example;
    j := j + 1;
```

Algorithm 5: Algorithm for Learning from Membership Queries Only

7. Learning Cartesian Products with Equivalence or Subset Queries is Hard

The previous section showed that learning cross products of membership queries requires at most $O(max_i\{\#Mem_i(c_i)\}^k)$ membership queries. A natural next question is whether this can be done for equivalence and subset queries. In this section, we answer that question in the negative. We will construct a class $\mathfrak{C}$ that can be learned from n equivalence or membership queries but which requires at least k^n queries to learn $\mathfrak{C}^k$.

We define $\mathfrak{C}$ to be the set $\{\mathfrak{c}(s) \mid s \in \mathbb{N}^*\}$, where $\mathfrak{c}(s)$ is defined as follows:

$$\mathfrak{c}(\lambda) = \{\lambda\} \times \mathbb{N}$$
$$\mathfrak{c}(s) = (\{s\} \times \mathbb{N}) \cup \mathfrak{c}_{sub}(s)$$
$$\mathfrak{c}_{sub}(sa) = (\{s\} \times (\mathbb{N}\backslash\{a\})) \cup \mathfrak{c}_{sub}(s)$$

For example, $\mathfrak{c}(12) = (\{12\} \times \mathbb{N}) \cup (\{1\} \times (\mathbb{N}\backslash\{2\})) \cup (\{\lambda\} \times (\mathbb{N}\backslash\{1\}))$.

An important part of this construction is that for any two strings $s, s' \in \mathbb{N}$, we have that $\mathfrak{c}(s) \subseteq \mathfrak{c}(s')$ if and only if $s = s'$. This implies that a subset query will return true if and only if the true concept has been found. Moreover, an adversarial oracle can always give a negative example for an equivalence query, meaning that oracle can give the same counterexample if a subset query were posed. So we will show that $\mathfrak{C}$ is learnable from equivalence queries, implying that it is learnable from subset queries.

We we prove a lower-bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. An adversarial equivalence query oracle can give the exact same answers and counterexamples, implying that $\mathfrak{C}^k$ is hard to learn from equivalence queries.

Proposition 7. *There exist algorithms for learning from equivalence queries or subset queries such that any concept $\mathfrak{c}(s) \in \mathfrak{C}$ can be learned from $|s|$ queries.*

Proof. (proof sketch) Algorithm 6 shows the learning algorithm for equivalence queries. As mentioned above, this algorithm is essentially the same for learning from subset queries. When learning $\mathfrak{c}(s)$ for any $s \in \mathbb{N}^*$, the algorithm will construct s by learning at least one new element of s per query. Each new query to the oracle is constructed from a string that is a substring of s If a positive counterexample is given, this can only yield a longer substring of s. □

Result:
Set $s = \lambda$;
while *True* **do**
  Query $\mathfrak{c}(s)$ to Oracle **if** *Oracle returns 'yes'* **then**
    return $\mathfrak{c}(s)$
  if *Oracle returns* $(s', m) \in c^* \backslash \mathfrak{c}(s)$ **then**
    Set $s = s'$;
  if *Oracle returns* $(s, m) \in \mathfrak{c}(s) \backslash c^*$ **then**
    Set $s = sm$;

Algorithm 6: Learning $\mathfrak{C}$ from equivalence queries.

7.1. **Showing $\mathfrak{C}^k$ is Hard to Learn.** It is easy to learn $\mathfrak{C}$, since each new counterexample gives one more element in the target string s. When learning a concept, $\prod \mathfrak{c}(s_i)$, it is not clear which dimension a given counterexample applies to. Specifically A given counterexample $\vec{x}$ could have the property that $\vec{x}[i] \in \mathfrak{c}(s_i)$ for all $i \neq j$, but the learner cannot infer the value of this j. It will then proceed considering all possible values of j, requiring exponentially more queries for longer s_i. This subsection will formalize this notion to prove an exponential lower bound on learning $\mathfrak{C}^k$. First, we need a couple definitions.

is this clear?

A concept $\prod \mathfrak{c}(s_i)$ is *justifiable* if one of the following holds:

- For all i, $s_i = \lambda$
- There is an i and an $a \in \mathbb{N}$ and $w \in \mathbb{N}^*$ such that $s_i = wa$, and $\mathfrak{c}(s_1) \times \cdots \times \mathfrak{c}(w) \times \cdots \times \mathfrak{c}(s_k)$ was justifiably queried to the oracle and received a counterexample $\vec{x}$ such that $\vec{x}[i] = (w, a)$.

need flat cross product

A concept is *justifiably queried* if it was queried to the oracle when it was justifiable.

The adversarial oracle works as follows:

- It will always answer the same query with the same counterexample.
- Given any query $\prod \mathfrak{c}(s_i) \subseteq c^*$, the oracle will return a counterexample $\vec{x}$ such that for all i, $\vec{x}[i] = (s_i, a_i)$, and a_i has not been in any query or counterexample yet seen.

Consider the following example when $k = 2$. First, the learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$ to the oracle and receives a counter-example $((\lambda, 1), (\lambda, 2))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$. The learner queries $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$ and receives counterexample $((1, 3), (\lambda, 4))$. The learner queries $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$ and receives counterexample $((\lambda, 5), (2, 6))$. The justifiable concepts are now $(\mathfrak{c}(1), \mathfrak{c}(4))$, $(\mathfrak{c}(1{\cdot}3), \mathfrak{c}(\lambda))$, $(\mathfrak{c}(5), \mathfrak{c}(2))$ and $(\mathfrak{c}(\lambda), \mathfrak{c}(2{\cdot}6))$. At this point, the only possible solutions constructible from strings of length 1 are $(\mathfrak{c}(1), \mathfrak{c}(4))$ and $(\mathfrak{c}(5), \mathfrak{c}(2))$.

For any strings $s, s' \in \mathbb{N}^*$, we write $s \leq s'$ if s is a substring of s', and we write $s < s'$ if $s \leq s'$ and $s \neq s'$. We say that the *sum of string lengths* of a concept $\prod \mathfrak{c}(s_i)$ is of size r if $\sum |s_i| = r$

The following simple proposition can be proven by induction on sum of string lengths.

Proposition 8. *Let $\prod \mathfrak{c}(s_i)$ be a justifiable concept. Then for all $w_i \leq s_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle.*

Proposition 9. *If all justified concepts $\prod \mathfrak{c}(s_i)$ with sum of string lengths equal to r have been queried, then there are k^{r+1} justified queries whose sum of string lengths equals $r + 1$*

Proof. This proof follows by induction on r. When $r = 0$, the concept $\prod \mathfrak{c}(\lambda)$ is justifiable.For induction, assume that there are k^r justifiable queries with sum of string lengths equal to r. By construction, the oracle will always chose counterexamples with as-yet unseen values in $\mathbb{N}$. So querying each concept $\prod \mathfrak{c}(s_i)$ will yield a counterexample $\vec{x}$ where for all i, $\vec{x}[i] = (s_i, a_i)$ for new a_i. Then for all i, this query creates the justifiable concept $\prod \mathfrak{c}(s'_i)$, where $s'_j = s_j$ for all $j \neq i$ and $s'_i = \mathfrak{c}(s_i \cdot a_i)$. Thus there are k^{r+1} justifiable concepts with sum of string lengths equal to $r + 1$. □

Theorem 1. *Any algorithm learning $\mathfrak{C}^k$ from subset (or equivalence) queries requires at least k^r queries to learn a concept $\prod \mathfrak{c}(s_i)$, whose sum of string lengths is r. Equivalently, the algorithm takes $k^{\sum \# q_i}$ subset (or equivalence) queries.*

explain how to assume the oracle will never return "true" on a query.

Proof. Assume for contradiction that an algorithm can learn with less than k^r queries and let this algorithm converge on some concept $c = \prod \mathfrak{c}(s_i)$ after less than k^r queries. Since less than k^r queries were made to learn c, by Proposition 9, there must be some justifiable concept $c' = \prod \mathfrak{c}(s'_i)$ with sum of string lengths less than or equal to r that has not yet been queried. By proposition 8 we can assume without loss of generality that for all $w_i \leq s'_i$, $\prod \mathfrak{c}(w_i)$ has been queried to the oracle. We will show that c' is consistent with all given oracle answers, contradicting the claim that c is the correct concept. Let $c_v = \prod \mathfrak{c} v_i$ be any concept queried to the oracle, and let $\vec{x}$ be the given counterexample. If for all i, $v_i \leq s'_i$,

	Only Q	Q with Mem and $1Pos$	
	#q	$\#Mem$	#q
Pos	Not Possible	Not Possible	Not Possible
Sup	$\sum \#Sup$	0	$\sum \#Sup$
Mem	$(max_i\{\#Mem_i\})^k$	$\sum \#Mem_i$	$\sum \#Mem_i$
Sub	$k^{\sum \#Sub_i}$	$k \sum \#Sub_i$	$\sum \#Sub_i$
EQ	$k^{\sum \#EQ_i}$	$k \sum \#EQ_i$	$\sum \#EQ_i$

FIGURE 2. Final collection of runtimes. The rows represents the set Q of queries needed to learn each C_i. The columns determine whether the cross product is learned from queries in just Q or $Q \cup \{Mem, 1Pos\}$. In the latter case, the column is separated to track the number of membership queries and queries in Q that are needed.

then by construction, there is an i with $\vec{x}[i] = (v_i, a_i)$ such that $v_i \cdot a_i \leq s_i'$, so $\vec{x}$ is a valid counterexample. Otherwise, there is an i such that $v_i \not\leq s_i'$. So $\{v_i\} \times \mathbb{N} \cap \mathfrak{c}(s_i') = \emptyset$, so $\vec{x}$ is a valid counterexample. Therefore, all counterexamples are consistent with c' being correct concept, contradicting the claim that the learner has learned c. $\square$