# Modular Learning: Independent Variables and Cartesian-Products

Benjamin Caulfield and Sanjit A. Seshia

University of California, Berkeley CA 94720, USA {bcaulfield,sseshia}@berkeley.edu

**Abstract.** We define and study the problem of modular concept learning, that is, learning a concept that is a cross product of component concepts. If an element's membership in a concept depends solely on it's membership in the components, learning the concept as a whole can be reduced to learning the components. We analyze this problem with respect to different types of oracle interfaces, defining different sets of queries. If a given oracle interface cannot answer questions about the components, learning can be difficult, even when the components are easy to learn with the same type of oracle queries. While learning from superset queries is easy, learning from membership, equivalence, or subset queries is harder. However, we show that these problems become tractable when oracles are given a positive example and are allowed to ask membership queries.

**Keywords:** Inductive Synthesis, Query-Based Learning, Modularity

## 1 Introduction

Inductive synthesis or inductive learning is the synthesis of programs (concepts) from examples or other observations. Inductive synthesis has found application in formal methods, program analysis, software engineering, and related areas, for problems such as invariant generation (e.g. [?]), program synthesis (e.g., [3]), and compositional reasoning (e.g. [?]). Most inductive synthesis follows the query-based 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 [1]. The special nature of query-based learning for formal synthesis, where a program is automatically generated to fit a high-level specification through interaction with oracles, has also been formalized [2].

In spite of this progress, most algorithms for inductive learning/synthesis are monolithic; that is, even if the concept (program) is made up of components, the algorithms seek to learn the entire concept from interaction with an oracle. In contrast, in this paper, we study the setting of *modular concept learning*, where a learning problem is analyzed by breaking it into independent components. If an element's membership in a concept depends solely on its membership in the components, learning the concept as a whole can be reduced to learning the components. We study concepts that are the Cartesian products (i.e., cross-products) of their component concepts. This can be thought of as learning conjunctions in that an element is part of a target concept if and only if its relevant variables are all of the component concepts.

We will focus on the oracle queries given in Table 1.

| 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* | $\#EQ(c)$ | 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 $x \in c^* \backslash c$ |
| Example Query | $EX_{\mathcal{D}}$ | $\#EX(c, \mathcal{D})$ | Samples $x$ from $\mathcal{D}$ and returns $x$ with a label indicating whether $x \in c^*$. |

Table 1: Types of queries studied in this paper.

```
int x = ??;
int y = ??;
int a = f(x);
int b = g(y);
```

$\Phi = \phi_1(a) \wedge \phi_2(b)$

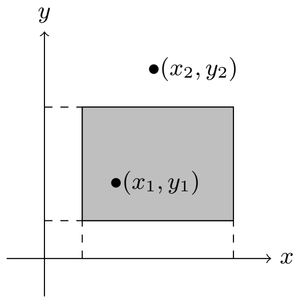

Fig. 1: A simple partial program to be synthesized to satisfy a specification $\Phi$ (left) and the correct set of initial values for $x$ and $y$ (right).

### 1.1 A Motivating Example

To illustrate the learning problem, consider the sketching problem given in Figure 1. Say we wanted to find the set of possible initial values for $x$ and $y$ that can replace the ?? values so that the program satisfies $\Phi$.

Looking at the structure of this program and specification, we can see that the correctness of these two variables are independent of each other. Correct $x$ values are correct independent of $y$ and vice-versa. Therefore, the set of settings will be the cross product of the acceptable settings for each variable. If an oracle can answer queries about correct $x$ or $y$ values separately, then the oracle can simply learn the acceptable values separately and take their Cartesian product.

If the correct values form intervals, the correct settings will look something like the rectangle shown in Figure 2. An algorithm for learning this rectangle can try to simulate learning algorithms for each interval by acting as the oracle for each sublearner. For example, if both sublearners need a positive example, the learner can query the oracle for a positive example. Given the example $(x_1, y_1)$ as shown in the figure, the learner can then pass $x_1$ and $y_1$ to the sublearners as positive examples. However, this does not apply to negative examples, such as $(x_2, y_2)$ in the figure. In this example, $x_2$ is in its target interval, but $y_2$ is not. The learner has no way of knowing which subconcept a negative element fails on. Handling negative counterexamples is one of the main challenges of this paper.

## 2 Notation

In the following proofs, we assume we are given concept classes $C_1, C_2, \dots, C_k$ defined over sets $X_1$, $X_2$, ..., $X_k$. Each $c_i^*$ in each $C_i$ is learnable from algorithm $A_i$ (called *sublearners*) using queries to an oracle that can answer any queries in a set $Q$. This set $Q$ contains the available types of queries and is a subset of the queries shown in Table 1. For example, if $Q := \{\mathit{Mem}, \mathit{EQ}\}$, then each $A_i$ can make membership and equivalence queries to its corresponding oracle.

For each query $q \in Q$, we say algorithm $A_i$ makes $\#q_i$ (or $\#q_i(c_i^*)$) many $q$ queries to the oracle in order to learn concept $c_i^*$, 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 $\#\mathit{Mem}$ many membership queries to learn $c$.

Unless otherwise stated, we will assume any index $i$ or $j$ ranges over the set $\{1 \dots k\}$. We write $\prod S_i$ or $S_1 \times \cdots \times S_k$ to refer to the $k$-ary Cartesian product (i.e., cross-product) of sets $S_i$. We use $S^k$ to refer to $\prod_{i=1}^k S$.

We use vector notation $\boldsymbol{x}$ to refer to a vector of elements $(x_1, \dots, x_k)$, $\boldsymbol{x}[i]$ to refer to $x_i$, and $\boldsymbol{x}[i \leftarrow x_i']$ to refer to $\boldsymbol{x}$ with $x_i'$ replacing value $x_i$ at position $i$. We define $\boxtimes_{i=1}^k C_i := \{\prod c_i \mid c_i \in C_i, i \in \{1, \dots, k\}\}$. We write $\boldsymbol{c}$ or $\prod c_i$ for any element of $\boxtimes_{i=1}^k C_i$ and will often denote $\boldsymbol{c}$ by $(c_1, \dots, c_k)$ in place of $\prod c_i$. The target concept will be represented as $c^*$ or $\prod c_i^*$ which equals $(c_1^*, \dots, c_k^*)$.

The results below answer the following question:

> For different sets of queries $Q$, what is the bound on the number of queries to learn a concept in $\boxtimes C_i$ as a function of each $\#q_i$ for each $q \in Q$?

The proofs in this paper make use of the following simple observation:

**Observation 1** *For sets $S_1, S_2, \dots, S_k$ and $T_1, T_2, \dots, T_k$, assume $\prod S_i \neq \emptyset$. Then $\prod S_i \subseteq \prod T_i$ if and only if $S_i \subseteq T_i$, for all $i$.*

## 3 Simple Lower Bounds

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.

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.*

*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 $x$.

## 4 Learning From Superset Queries

This section introduces arguably the simplest positive result of the paper: when using superset queries, learning cross-products of concepts is as easy as learning the individual concepts.

Like all positive results in this paper, this is accomplished by algorithm that takes an oracle for the cross-product concept $\prod c_i^*$ and simulates the learning process for each sublearner $A_i$ by acting as an oracle for each such sublearner.

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

*Proof.* Algorithm 1 learns $\boxtimes C_i$ by simulating the learning of each $A_i$ on its respective class $C_i$. The algorithm asks each $A_i$ for superset queries $S_i \supseteq c_i^*$, 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 $\sum \#Sup(c_i^*)$ 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 and stops if so. If no concept class contains the empty concept, this check can be skipped. At each step, the algorithm poses query $\prod S_i$ to the oracle. If the oracle returns 'yes' (meaning $\prod S_i \supseteq \prod c_i^*$), 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 $\boldsymbol{x} = (x_1, \ldots, x_k) \in \prod c_i^* \backslash \prod S_i$. There must be at least one $x_i \notin S_i$ (otherwise, $\boldsymbol{x}$ 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 $\boldsymbol{x} \in \prod c_i^*$, we know $x_i \in c_i^*$, so $x_i \in c_i^* \backslash S_i$, 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 $j \neq i$.

```
Result: Learn $\prod C_i$ from Superset Queries
if $\emptyset \in C_i$ for some $i$ then
    Query $\emptyset \supseteq \prod c_i^*$;
    if $\emptyset \supseteq \prod c_i^*$ 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 $\boldsymbol{x} = (x_1, \ldots, x_k)$ 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

## 5 Learning From Membership Queries and One Positive Example

Ideally, learning the cross-product of concepts should be about as easy as learning all the individual concepts. The last section showed this is not the case when learning with equivalence, subset, or membership queries. However, when the learner is given a single positive example and allowed to make membership queries, the number of queries becomes tractable. This is due to the following simple observation.

**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 $x_i \in S_i$.*

So, given a positive example $\boldsymbol{p}$, we can see that $\boldsymbol{p}[j \leftarrow x_j] \in \prod c_i^*$ if and only if $x_j \in c_j^*$. This fact is used to learn using subset or equivalence queries with the addition of membership queries and a positive example. The algorithm is fairly similar for equivalence and subset queries, and is shown as a single algorithm in Algorithm 2.

**Proposition 4.** *If $Q \in \{\{\text{Sub}\}, \{\text{EQ}\}\}$ and a single positive example $\boldsymbol{p} \in \prod c_i^*$ is given, then $\prod c_i^*$ is learnable in $\sum \#q_i$ queries from $Q$ (i.e., subset or equivalence queries) and $k \cdot \sum \#q_i$ membership queries.*

*Proof.* The learning process for either subset or equivalence queries is described in Algorithm 2, with differences marked in comments. In either case, 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 $i \neq j$.

We separately show for each type of query that a correct answer is given to at least one learner $A_i$ for each subset (resp. equivalence) query to the cross-product oracle. Moreover, at most $k$ membership queries are made per subset (resp. equivalence) query, yielding the desired bound.

Subset Queries: For each subset query $\prod S_i \subseteq \prod c_i^*$, the algorithm either returns 'yes' or gives a counterexample $\boldsymbol{x} = (x_1, \ldots, x_k) \in \prod S_i \setminus \prod c_i^*$. 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, $\boldsymbol{x} \notin \prod c_i^*$, so there is an $i$ such that $x_i \notin c_i^*$. By Observation 2 the algorithm can query $\boldsymbol{p}[j \leftarrow x_j]$ for all $j$ until the $x_i \notin c_i^*$ is found.

Equivalence Queries: For each equivalence query $\prod S_i = \prod c_i^*$, the algorithm either returns 'yes', or gives a counterexample $\boldsymbol{x} = (x_1, \ldots, x_k)$. If the algorithm returns 'yes', then a valid target concept is learned. Otherwise, either $\boldsymbol{x} \in \prod S_i \setminus \prod c_i^*$ or $\boldsymbol{x} \in \prod c_i^* \setminus \prod S_i$. In the first case, as with subset queries, the algorithm uses $k$ membership queries to query $\boldsymbol{p}[j \leftarrow x_j]$ for all $j$. Once the $x_i \notin c_i^*$ is found it is given to $A_i$ as a counterexample. In the second case, as with superset queries, the algorithm checks if $x_j \in S_j$ for all $j$ until the $x_i \notin c_i^*$ is found and given to $A_i$.

```
Result: Learn $\prod c_i^* \in \boxtimes C_i$
for $i = 1 \ldots k$ do
    Set $S_i$ to initial query from $A_i$
while Some $A_i$ has not completed do
    Query $\prod S_i$ to oracle;
    if The Oracle returns 'yes' then
        Pass 'yes' to each $A_i$;
        // If Q = {EQ} each sublearner will immediately complete
    else
        Get counterexample $\boldsymbol{x} = (x_1, \ldots, x_k)$;
        if $\boldsymbol{x} \in \prod c_i^* \setminus \prod S_i$ then
            // Only happens if Q = {EQ}
            for $i$ = 1 ... $k$ do
                if $x_i \notin S_i$ then
                    Pass counterexample $x_i$ to $A_i$;
                    Update $S_i$ to new query from $A_i$;
        else
            for $i$ = 1 ... $k$ do
                Query $\boldsymbol{p}[i \leftarrow x_i] \in \prod c_i^*$;
                if $\boldsymbol{p}[i \leftarrow x_i] \notin \prod c_i^*$ and $x_i \in S_i$ then
                    Pass counterexample $x_i$ to $A_i$;
                    Update $S_i$ to new query from $A_i$;
Each $A_i$ returns some $c_i$;
Return $\prod c_i$;
```

**Algorithm 2:** Algorithm for learning from Equivalence (or Subset) Queries, Membership Queries, and One Positive Example

Finally, learning from only membership queries and one positive example if fairly easy.

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

*Proof.* The algorithm 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 $\boldsymbol{p}[i \leftarrow M_i] \in \prod c_i^*$

by Observation 2. Therefore the algorithm is successfully able to simulate the oracle for each $A_i$, yielding a correct hypothesis $c_i$.

## 6 Learning From 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, learning from their cross-product is impossible. 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 the simple algorithm from Proposition 5 for learning from membership queries and a single positive example.

**Proposition 6.** *Algorithm 3 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 $|S_i| > \#Mem_i(c_i^*)$. We will now show this is true for each $i$.

Assume that the sublearner $A_i$ eventually terminates with the correct answer $c_i^*$. Let $\boldsymbol{q_i} := (q_1^i, q_2^i, \dots) \in X_i^*$ be the elements whose membership $A_i$ would query assuming it only received negative answers from an oracle. If $\boldsymbol{q_i}$ is finite, then there is some set $N_i \in C_i$ that $A_i$ outputs after querying all elements in $\boldsymbol{q_i}$ (and receiving negative answers). We will consider the cases when $c_i^* = N_i$ and $c_i^* \neq N_i$

Assume $c_i^* = N_i$: Then by our assumption that $\prod c_i^* \neq \emptyset$, $N_i$ contains some element $n_i$. 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. The algorithm will start with $S_i := \{n_i\}$, so $S_i$ contains an element of $c_i^*$ at the start of the algorithm.

Assume $c_i^* \neq N_i$: By our assumption that $A_i$ eventually terminates, $A_i$ must eventually query some $q_j^i \in c_i^*$ (Otherwise, $A_i$ would only receive negative answers and would output $N_i$). 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.

```
for $i = 1 \ldots k$ do
    if $N_i$ and $n_i$ exist then
        Set $S_i := \{n_i\}$;
    else
        Set $S_i := \{\}$;
Set $j := 0$;
while Some $A_i$ has not terminated do
    for $i$ = 1, ..., $k$ do
        if $A_i$ has not terminated then
            Get query $q^i_j$ from $A_i$ ;
            Pass answer 'no' to $A_i$;
            Set $S_i := S_i \cup \{q^i_j\}$;
    for $\boldsymbol{x} \in \prod S_i$ do
        Query $\boldsymbol{x} \in \prod c^*_i$;
        if $\boldsymbol{x} \in \prod c^*_i$ then
            Run Proposition 5 algorithm using $\boldsymbol{x}$ as a positive example;
    $j := j + 1$;
```

**Algorithm 3:** Algorithm for Learning from Membership Queries Only

## 7 Learning from 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 subset 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 will prove a lower-bound on learning $\mathfrak{C}^k$ from subset queries from an adversarial oracle. This will imply that $\mathfrak{C}^k$ is hard to learn from equivalence queries, since an adversarial equivalence query oracle can give the exact same answers and counterexamples as a subset query oracle.

**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.* (sketch) Algorithm 4 shows the learning algorithm for equivalence queries, and Figure 7 show the decision tree. 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: Learns $\mathfrak{C}$
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 4:** Learning $\mathfrak{C}$ from equivalence queries.

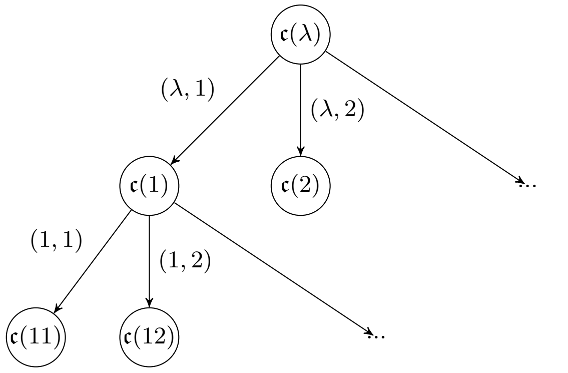

Fig. 2: A tree representing Algorithm 4. Nodes are labelled with the queries made at each step, and edges are labelled with the counterexample given by the oracle.

### 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 $\boldsymbol{x}$ could have the property that $\boldsymbol{x}[i] \in \mathfrak{c}(s_i)$ for all $i \neq j$, but the learner cannot infer the value of this $j$. It must 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.

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 the $k$-ary cross-product $\mathfrak{c}(s_1) \times \cdots \times \mathfrak{c}(w) \times \cdots \times \mathfrak{c}(s_k)$ was justifiably queried to the oracle and received a counterexample $\boldsymbol{x}$ such that $\boldsymbol{x}[i] = (w, a)$.

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

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 $\sum |s_i| = r$

Proving that learning is hard in the worst-case can be thought of as a game between learner and oracle. The oracle can answer queries without first fixing the target concept. It will answer queries so that for any $n$, after less than $k^n$ queries, there is a concept consistent with all given oracle answers that the learning algorithm will not have guessed. The specific behavior of the oracle is defined 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 $\boldsymbol{x}$ such that for all $i$, $\boldsymbol{x}[i] = (s_i, a_i)$, and $a_i$ has not been in any query or counterexample yet seen.
- The oracle never returns 'yes' on any query.

The remainder of this section assumes that queries are answered by the above oracle. An example of answers by the above oracle and the justifiable queries it yields is given below.

*Example 1.* 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, these are the only possible solutions whose sum of string lengths is 2. The graph of justifiable queries is given in Figure 7.1.

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 $w_1, w_2, \ldots, w_k$ where for all $i$, $w_i \leq s_i$, $\prod \mathfrak{c}(w_i)$ 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 $\boldsymbol{x}$ where for all $i$, $\boldsymbol{x}[i] = (s_i, a_i)$ for new $a_i$. Then for all $i$, this query creates the justifiable concept $\prod \mathfrak{c}(s'_j)$, 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 $r + 1$.

We are finally ready to prove the main theorem of this section.

**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.*

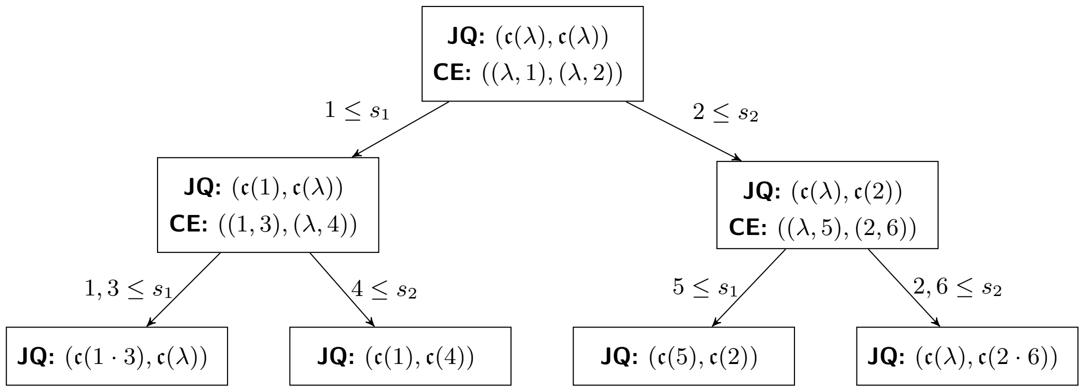
JQ: $(\mathfrak{c}(\lambda), \mathfrak{c}(\lambda))$
CE: $((\lambda, 1), (\lambda, 2))$
$1 \leq s_1$
$2 \leq s_2$
JQ: $(\mathfrak{c}(1), \mathfrak{c}(\lambda))$
CE: $((1, 3), (\lambda, 4))$
JQ: $(\mathfrak{c}(\lambda), \mathfrak{c}(2))$
CE: $((\lambda, 5), (2, 6))$
$1, 3 \leq s_1$
$4 \leq s_2$
$5 \leq s_1$
$2, 6 \leq s_2$
JQ: $(\mathfrak{c}(1 \cdot 3), \mathfrak{c}(\lambda))$
JQ: $(\mathfrak{c}(1), \mathfrak{c}(4))$
JQ: $(\mathfrak{c}(5), \mathfrak{c}(2))$
JQ: $(\mathfrak{c}(\lambda), \mathfrak{c}(2 \cdot 6))$

Fig. 3: The tree of justifiable queries used in Example 1. Each node lists the justifiable query (JQ) and counterexample (CE) given for that query. The edges below each node are labelled with the possible inferences about $s_1$ and $s_2$ that can be drawn from the counterexample.

*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 $\boldsymbol{x}$ be the given counterexample. If for all $i$, $v_i \leq s'_i$, then by construction, there is a $j$ with $\boldsymbol{x}[j] = (v_j, a_j)$ such that $v_j \cdot a_j \leq s'_j$, so $\boldsymbol{x}$ 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 $\boldsymbol{x}$ is a valid counterexample. Therefore, all counterexamples are consistent with $c'$ being correct concept, contradicting the claim that the learner has learned $c$.

## 8 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 $A \uplus B$ and the disjoint union of many sets as $\biguplus A_i$. We define the concept class of disjoint unions as $C_\uplus := \{\biguplus c_i \mid c_i \in C_i\}$.

The algorithm for learning from membership queries is very easy and won't be stated here. Algorithm 5 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 $S_1, \ldots, S_k$ and $T_1, \ldots, T_k$. Then $\biguplus S_i \subseteq \biguplus T_i$ if and only $S_i \subseteq T_i$ for all $i$. Likewise $\biguplus S_i = \biguplus T_i$ if and only if $S_i = T_i$ for all $i$.

We can summarize these results in the following proposition.

**Proposition 10.** *Take any $Q \in \{\{\mathrm{EQ}\}, \{\mathrm{Sub}\}, \{\mathrm{Sup}\}, \{\mathrm{Mem}\}\}$ and assume each concept class $C_i$ is learnable from $\#qi$ many queries. Then there exists an algorithm that can learn the disjoint union of concept classes $\biguplus c_i$ in $\sum \#qi$ many queries.*

```
Result: Learning Disjoint Unions
for $i = 1 \ldots k$ 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 5:** Learning Disjoint Unions

## 9 Efficient PAC-Learning

This sections discusses the problem of PAC-learning the cross products of concept classes.

Previously, van Der Vaart and Weller [4] have shown the following bound on the VC-dimension of cross-products of sets:

$$\mathcal{VC}(\prod C_i) \leq a_1 log(ka_2) \sum \mathcal{VC}(C_i)$$

Here $a_1$ and $a_2$ are constants with $a_1 \approx 2.28$ and $a_2 \approx 3.92$. As always, $k$ is the number of concept classes included in the cross-product.

The VC-dimension gives a bound on the number of labelled examples needed to PAC-learn a concept, but says nothing of the computational complexity of the learning process. This complexity mostly comes from the problem of finding a concept in a concept class that is consistent with a set of labelled examples. We will show that the complexity of learning cross-products of concept classes is a polynomial function of the complexity of learning from each individual concept class.

First, we will describe some necessary background information on PAC-learning.

### 9.1 PAC-learning Background

**Definition 1.** *Let $C$ be a concept class over a space $X$. We say that $C$ is efficiently PAC-learnable if there exists an algorithm $A$ with the following property: For every distribution $\mathcal{D}$ on $X$, every $c \in C$, and every $\epsilon, \delta \in (0,1)$, if algorithm $A$ is given access to $EX(c, \mathcal{D})$ then with probability $1-\delta$, $A$ will return a $c' \in h$ such that $error(c') \leq \epsilon$. $A$ must run in time polynomial in $1/\epsilon$, $1/\delta$, and $size(c)$.*

We will refer to $\epsilon$ as the 'accuracy' parameter and $\delta$ as the 'confidence' parameter. The value of $error(c)$ is the probability that for an $x$ sampled from $\mathcal{D}$ that $c(x) \neq c^*(x)$. PAC-learners have a *sample complexity* function $m_C(\epsilon, \delta) : (0,1)^2 \to \mathbb{N}$. The sample complexity is the number of samples an algorithm must see in order to probably approximately learn a concept with parameters $\epsilon$ and $\delta$.

Given a set $S$ of labelled examples in $X$, we will use $A(S)$ to denote the the concept class the algorithm $A$ returns after seeing set $S$ .

A learner $A$ is an *empirical risk minimizer* if $A(C)$ returns a $c \in C$ that minimizes the number of misclassified examples (i.e., it minimizes $|\{(x,b) \in S \mid c^*(x) \neq b\}|$).

Empirical risk minimizers are closely related to VC dimension and PAC-learnability as shown in the following theorem (Theorem 6.7 from [5])

**Theorem 2.** *If the concept class $C$ has VC dimension $d$, then there is a constant, $b$, such that applying an Empirical Risk Minimizer $A$ to $m_C(\epsilon, \delta)$ samples will PAC-learn in $C$, where*

$$m_C(\epsilon, \delta) \leq b \frac{d \cdot log(1/\epsilon) + log(1/\delta)}{\epsilon}$$

Finally, we will discuss the *growth function.* The growth function describes how many distinct assignments a concept class can make to a given set of elements. More formally, for a concept class $C$ and $m \in \mathbb{N}$, the growth function $G_C(m)$ is defined by:

$$G_C(m) = \max_{x_1, x_2, \ldots, x_m} \Big| \{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C\} \Big|$$

Each $x_i$ in the above equation is taken over all possible elements of $X_i$. The VC-dimension of a class $C$ is the largest number $d$ such that $G_C(d) = 2^d$.

We will use the following bound, a corollary of the Perles-Sauer-Shelah Lemma, to bound the runtime of learning cross-products [5].

**Lemma 1.** *For any concept class $C$ with VC-dimension $d$ and $m > d + 1$:*

$$G_C(m) \leq (em/d)^d$$

## 9.2 PAC-Learning Cross-Products

We now have enough background to describe the strategy for PAC-learning cross-products. We will just describe learning the cross-product of two concepts. As above, assume concept classes $C_1$ and $C_2$ and PAC-learners $A_1$ and $A_2$ are given. We define $A_i(\epsilon, \delta)$ as the runtime of the sublearner $A_i$ to PAC-learn with accuracy parameter $\epsilon$ and confidence parameter $\delta$.

Assume that $C_1$ and $C_2$ have VC-dimension $d_1$ and $d_2$, respectively. We can use the bound from van Der Vaart and Weller to get an upper bound $d$ on the VC-dimension of their cross-product. Assume the algorithm is given an $\epsilon$ and $\delta$ and there is a fixed target concept $c^* = c_1^* \times c_2^*$. Theorem 2 gives a bound on the sample complexity $m_{C_1 \times C_2}(\epsilon, \delta)$. The algorithm will take a sample of labelled examples of size $m_{C_1 \times C_2}(\epsilon, \delta)$. Our goal is to construct an Empirical Risk Minimizer for $C_1 \times C_2$. In our case, $c_1 \in C_1$ and $c_2 \in C_2$. Therefore, for any sample $S$, an Empirical Risk Minimizer will yield a concept in $C_1 \times C_2$ that is consistent with $S$. This algorithm is show in Algorithm 6.

So let $S$ be any such sample the algorithm takes. This set can easily be split into positive examples $S^+$ and negative examples $S^-$, both in $X_1 \times X_2$. The algorithm works by maintaining sets labeled samples $L_1$ and $L_2$ for each dimension. For any $(x_1, x_2) \in S^+$, it holds that $x_1 \in c_1^*$ and $x_2 \in c_2^*$ so $(x_1, \top)$ and $(x_2, \top)$ are added to $L_1$ and $L_2$ respectively. For any $(x_1, x_2) \in S^-$, we know that $x_1 \notin c_1^*$ or $x_2 \notin c_2^*$ (or both), but it is not clear which is true. However, since the goal is only to create an Empirical Risk Minimizer, it is enough to find any concepts $C_1$ and $C_2$ that are consistent with these samples. In other words, we need to find a $c_1 \in C_1$ and a $c_2 \in C_2$ such that for every $(x_1, x_2) \in S^+$, $x_1 \in c_1$ and $x_2 \in c_2$ and for all $(x_1, x_2) \in S^-$, either $x_1 \notin c_1^*$ or

$x_2 \notin c_2^*$. One idea would be to try out all possible assignments to elements in $S^-$ and check if any such assignment fits any possible concepts. This, however, would be exponential in $|S^-|$.

Bounding the size of the growth function can narrow this search. Specifically, let $S_1^- := \{x \mid \exists y, (x, y) \in S^-\}$, let $m = |S_1^-|$ and order the elements of $S_1^-$ by $x_1, x_2, \ldots, x_m$. By the definition of the growth function and Lemma 1:

$$|\{(c(x_1), c(x_2), \ldots, c(x_m)) \mid c \in C_1\}| \leq G_{C_1}(m) \leq (em/d)^d$$

In other words, there are less than $(em/d)^d$ assignments of truth values to elements of $S_1^-$ that are consistent with some concept in $C_1$. If the algorithm can check every $c_1 \in C_1$ consistent with $S^+$ and $S_1^-$, it can then call $A_2$ to see if there is any $c_2 \in C_2$ such that $(c_1 \times c_2)$ assigns true to every element in $S^+$ and false to every element in $S^-$.

Finding these consistent elements of $C_1$ is made easier by the fact that we can check whether partial assignments to $S_1^-$ are consistent with any concept in $C_1$. As mentioned above, it starts by creating the sets $L_1$ and $L_2$ containing all samples in the first and second dimension of $S^+$, respectively. It then iteratively adds labeled samples from $S^-$. At each step, the algorithm chooses one element $(x_1, x_2) \in S^-$ at a time and checks which possible assignments to $x_1$ are consistent with $L_1$. If $(x_1, \bot)$ is consistent, it adds $(x_1, \bot)$ to $L_1$ and calls *RecursiveFindSubconcepts* on $L_1$ and $L_2$. If $(x_1, \top)$ is consistent with $C_1$, then the algorithm adds $(x_1, \top)$ to $L_1$ and $(x_2, \bot)$ to $L_2$ and calls *RecursiveFindSubconcepts*. In either case, if an assignment is not consistent, no recursive call is made. We can summarize these results in the following theorem.

**Theorem 3.** *Let concept classes $C_1$ and $C_2$ have VC-dimension $d_1$ and $d_2$, respectively. There exists a PAC-learner for $C_1 \times C_2$ that can learn any concept using a sample of size $m = ((d_1 + d_2) \cdot log(1/\epsilon) + log(1/\delta))/\epsilon$. The learner requires time $O(m^{d_1}(A_1(1/m, log(\delta)) + A_2(1/m, log(\delta))))$.*

```
Result: Find Subconcepts Consistent with Sample
Input: $S^+$: Set of positive examples in $X_1 \times X_2$
$S^-$: Set of negative examples in $X_1 \times X_2$
$\delta$ : Confidence parameter in $(0, 1)$
FindSubconcepts ($S^+$, $S^-$, $\delta$)
    $\delta' := \delta/(|S^-|G_{C_1}(|S^-|) + G_{C_2}(|S^-|))$;
    $\epsilon' := 1/|S|$;
    // $L_1$: Labelled samples in $X_1$
    $L_1 := \{(x_1, \top) \mid \exists y, (x_1, y) \in S^+\}$;
    // $L_2$: Labelled samples in $X_2$
    $L_2 := \{(x_2, \top) \mid \exists y, (y, x_2) \in S^+\}$;
    $U := S^-$;
    return RecursiveFindSubconcepts($L_1$, $L_2$, $U$, $\epsilon'$, $\delta'$) ;
RecursiveFindSubconcepts ($L_1$, $L_2$, $U$, $\epsilon'$, $\delta'$):
    // Run once all labels are assigned to the first dimension of $U$
    // Attempts to find concept in $C_2$ consistent with all labels given in $L_2$
    if $U = \emptyset$ then
        if $A_2(L_2, \epsilon', \delta')$ then
            return $(A_1(L_1, \epsilon', \delta'), A_2(L_2, \epsilon', \delta'))$;
        else
            $\perp$;
    Get $(x_1, x_2) \in U$;
    $U := U \backslash \{(x_1, x_2)\}$;
    // Attempts to label $x_1$ as false in $c_1^*$
    if $A_1(L_1 \cup \{(x_1, \perp)\}, \epsilon', \delta') \neq \perp$ then
        $c = RecursiveFindSubconcepts(L_1 \cup \{(x_1, \perp)\}, L_2, U, \epsilon', \delta')$;
        if $c \neq \perp$ then
            return c;
    // Attempts to label $x_1$ as true in $c_1^*$
    if $A_1(L_1 \cup \{(x_1, \top)\}, \epsilon', \delta') \neq \perp$ then
        $c = RecursiveFindSubconcepts(L_1 \cup \{(x_1, \top)\}, L_2 \cup \{(x_2, \perp)\}, U, \epsilon', \delta')$;
        if $c \neq \perp$ then
            return c ;
```

**Algorithm 6:** Learning Disjoint Unions

### 9.3 Efficient PAC-learning with Membership Queries

Although polynomial, the complexity of PAC-learning cross-products from a *EX* oracle is fairly expensive. We will show that when a learner is allowed to make membership queries, PAC-learning cross-products becomes much more efficient. This is due to the previously shown technique, which uses membership queries and a single positive example to determine on which dimensions a negatively labelled example fails.

In this case, assuming that $\emptyset \in \boxtimes C_i$, we can ignore the assumption that a positive example is given. If no positive example appears in a large enough labeled sample, the the algorithm can pose $\emptyset$ as the hypothesis.

If $S$ does contain a positive example $\boldsymbol{p}$, then $S$ can be broken down into labeled samples for each dimension $i$. The algorithm initialize the sets of positive and negative examples to $S_i^+ := \{\boldsymbol{x}[i] \mid (\boldsymbol{x}, \top) \in S\}$ and $S_i^- := \{\}$, respectively. For each $(\boldsymbol{x}, \bot) \in S$, a membership queries $\boldsymbol{p}[i \leftarrow \boldsymbol{x}[i]] \in \prod c_i^*$. If so, $\boldsymbol{x}[i]$ is added to $S_i^+$. Otherwise it is added to $S_i^-$. This labelling is correct by Observation 2. The set of labelled examples $S_i := (S_i^+ \times \{\top\}) \cup (S_i^- \times \{\bot\})$ is then passed to the sublearner $A_i$. $A_i$ is run on $S_i$ with accuracy parameter $\epsilon' := \epsilon/k$ and confidence parameter $\delta' := \delta/k$ .

**Proposition 11.** *The algorithm described above PAC-learns from the concept class $\boxtimes C_i$ with accuracy $\epsilon$ and confidence $\delta$. It makes $m_C(\epsilon, \delta)$ queries to* EX*, $k \cdot m_C(\epsilon, \delta)$ membership queries, and has runtime $O(\sum A_i(\epsilon/k, \delta/k))$.*

## 10 Conclusion

The final collection of query complexities for learning cross products is given in Figure 5. All of the bounds are tight, except for the problem of learning with superset queries, membership queries, and one positive example. Additionally, learning disjoint unions from any $Q \in \{\{EQ\}, \{Sub\}, \{Sup\}\}$ requires as many queries as is needed to learn each concept separately. Finally, we have shown that the computational complexity of PAC learning cross-products of concepts is a polynomial function of learning the individual concepts.

| | Only $Q$ | $Q$ with *Mem* and *1Pos* | |
|---|---|---|---|
| $Q \downarrow$ | #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* | $k^{\sum \#EQ_i}$ | $k \sum \#EQ_i$ | $\sum \#EQ_i$ |

Fig. 4: Final collection of query complexities for learning cross products. 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.

## References

1. Dana Angluin. Queries and concept learning. *Machine learning*, 2(4):319–342, 1988.
2. Susmit Jha and Sanjit A Seshia. A theory of formal synthesis via inductive learning. *Acta Informatica*, 54(7):693–726, 2017.
3. Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. *ACM Sigplan Notices*, 41(11):404–415, 2006.

4. Aad Van Der Vaart and Jon A Wellner. A note on bounds for vc dimensions. *Institute of Mathematical Statistics collections*, 5:103, 2009.
5. Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms.* Cambridge university press, 2014.