Learning Coalition-Based Interactions in Networked Social Systems

Abhijin Adiga[1] **Chris J. Kuhlman**[1] **Madhav V. Marathe**[1,2] **S. S. Ravi**[1,3]
Daniel J. Rosenkrantz[1,3] **Richard E. Stearns**[1,3] **Anil Vullikanti**[1,2]

[1]Biocomplexity Institute and Initiative, University Virginia, Charlottesville, VA 22904
[2]Computer Science Dept., University Virginia, Charlottesville, VA 22904
[3]Computer Science Dept., University at Albany – SUNY, Albany, NY 12222

{abhijin, cjk8gx, marathe}@virginia.edu, ssravi0@gmail.com, drosenkrantz@gmail.com,
thestearns2@gmail.com, vskumar@virginia.edu

Abstract

Using a discrete dynamical system model for a networked social system, we consider the problem of learning a class of local interaction functions in such networks. Our focus is on learning local functions which are based on pairwise disjoint coalitions formed from the neighborhood of each node. Our work considers both active query and PAC learning models. We establish bounds on the number of queries needed to learn the local functions under both models. We also establish a complexity result regarding efficient consistent learners for such functions. Our experimental results on synthetic and real social networks demonstrate how the number of queries depends on the structure of the underlying network and number of coalitions.

1 Introduction

Motivation. Learning the nature of interactions in networked physical and social systems is a challenging problem (see e.g., (Laubenbacher and Stigler 2004; Romero, Meeder, and Kleinberg 2011; González-Bailón et al. 2011)). We use a graphical dynamical systems model, called a synchronous dynamical system (SyDS) (see e.g., (Barrett et al. 2006)) to represent these networked systems. Such a system consists of an undirected graph $G(V, E)$, where the nodes represent entities (agents) and the edges represent pairwise interactions. (Formal definitions are provided in Section 2.) Each node v has a time varying state value (assumed to be Boolean) and a local function f_v which determines the next state of the node using the current states of v and its neighbors. The SyDS model assumes that nodes compute and update their state values synchronously. The graph and the local functions determine the dynamics of the system.

The problem of understanding the nature of interactions in a networked system can be formulated as that of inferring the local functions in a SyDS model of the system. We consider inference through interactions with the system where a user may specify each query in the form of a configuration (i.e., the current state values of nodes) and the system provides the successor configuration, i.e., states of the nodes at the next time instant (Adiga et al. 2018; He et al. 2016). We also consider inference under the Probably Approximately Correct (PAC) learning framework where configuration–successor pairs are independently drawn from an unknown distribution (see similar work in (Narasimhan, Parkes, and Singer 2015; He et al. 2016)). Under both models, we assume that the network is known.

The great majority of prior work focuses on each agent's individual behavior, where an agent treats each neighbor as an autonomous influencer. However, in several situations, an agent is influenced by groups formed by its neighbors. Here, our focus is on learning a form of interaction based on pairwise disjoint **coalitions** formed by the neighbors of an agent. The motivation for this model comes from the work reported in (Ugander et al. 2012; Laubenbacher and Stigler 2004; Colón-Reyes et al. 2006). The model studied in (Ugander et al. 2012) uses a social network and considers the connected components formed by the one-hop neighbors of a node v. Since the connected components are node disjoint, so are the coalitions. The experimental evidence presented in (Ugander et al. 2012) shows that coalitions are indeed operative in social networked systems. In particular, the results in this reference point out that people consider coalitions of their neighbors in deciding whether to join Facebook. Thus, our model of non-overlapping coalitions has direct relevance to social systems. The model studied in (Laubenbacher and Stigler 2004; Colón-Reyes et al. 2006) considers polynomial interaction functions, where each polynomial is a sum of monomials (products of variables where the degree of each variable is at most 1). The monomials can be thought of as coalitions. For Boolean functions, sums of monomials correspond to monotone functions in disjunctive normal form (DNF); such functions are in the sum of products form where no variable appears negated. Each product term in a DNF represents a coalition. Since coalitions considered in models of social systems generally do not overlap (Branzei, Dimitrov, and Tijs 2005; Ugander et al. 2012), we have the additional requirement that the coalitions must partition the set of inputs. We call such functions **partitioned monotone DNF** (PM-DNF) functions. As an example, the Boolean function of five variables defined by $f(x_1, x_2, x_3, x_4, x_5) = x_1\, x_5 + x_2\, x_3 + x_4$ consists of three pairwise disjoint coalitions. The interpretation is that the function takes on the value 1 iff least one of the coalitions is **unanimous**, i.e., all the variables in that coalition have the value 1. Thus, in a social system with PM-DNF functions, a node changes to 1 at time $\tau + 1$ iff there is at least one unanimous coalition among its inputs at time τ.

Our work considers the problem of learning PM-DNF functions under the active query model of (Adiga et al. 2018) and the PAC learning model (Valiant 1984). Two extreme cases of the PM-DNF model are well studied: (i) the Boolean OR function where every coalition has exactly one neighbor (which corresponds to the simple contagion model of (Granovetter 1978)) and (ii) Boolean AND function where all the neighbors form a single coalition (which corresponds to a particular type of complex contagion model of (Centola and Macy 2007)). Our model is a generalization of these two extreme cases. Also, to the best of our knowledge, this is the first work that addresses learning functions that depend on groups of neighbors, rather than individual neighbors. Such a dynamical system can also be viewed as a model for diffusion on hypergraphs (Zhu et al. 2018).

Summary of results.
1. Bounds under the active query model. We present an algorithm that can learn any PM-DNF function with q inputs using $O(q \log q)$ membership queries[1] under the adaptive mode (where a query may depend on the answers to the previous queries). We also show that in the worst-case, $\Omega(q \log q)$ queries are required under the adaptive mode to infer such a function. In addition, we show that $O(\chi \Delta \log \Delta)$ adaptive queries are sufficient to infer all the local functions of a SyDS where χ and Δ represent the number of colors needed to color G^2 (the square graph[2] of G) and the maximum node degree of G respectively.
2. PAC model upper bound. For any fixed values of the parameters ϵ and δ, we show that for learning the PM-DNF functions at all the nodes of a SyDS, an upper bound on the sample complexity is $O\big((2m + n) \log(\Delta + 1)\big)$, where $m = |E|$ and Δ is the maximum node degree.
3. Complexity of efficient PAC learning. We show that the class of PM-DNF functions with two or more product terms is not efficiently PAC learnable unless **NP** = **RP**. (The corresponding problem for one product term is efficiently solvable since a PM-DNF function with one product term is just the AND function.)
4. An algorithm for learning under the PAC model. To cope with the intractability result mentioned in Item 3 above, we present an integer linear programming (ILP)-based algorithm for determining whether there is a PM-DNF function that is consistent with all the given examples (definition in Section 4). This algorithm can be used to construct a PAC learning algorithm for PM-DNF functions in practice.
5. Experimental results. We present experimental results for generating query sets under the adaptive model for both synthetic and real social networks. The number of queries required depends on the structure of the graph and number of blocks (i.e., coalitions). For example, in the case of scale-free networks, the number of queries required is much less than the theoretical upper bound established in this paper. Under the PAC model, we analyze a single local function with regard to sample distribution, size of the input and number of blocks. Interestingly, the ILP-based algorithm exhibits better performance when the number of blocks is large.

For space reasons, proofs for many of the results are omitted; they appear in (Adiga et al. 2019a).

Related work. Many researchers have addressed the problem of learning components of physical and social systems (see e.g., (Adiga et al. 2018; He et al. 2016; Laubenbacher and Stigler 2004; Romero, Meeder, and Kleinberg 2011; González-Bailón et al. 2011)). As mentioned earlier, the coalition-based interaction model was motivated by the work in (Ugander et al. 2012; Colón-Reyes et al. 2006). The problem of learning Boolean DNF functions has received attention in the learning theory literature under membership query and PAC learning models. For example, bounds on the number of membership queries for learning monotone DNFs are proven in (Abasi, Bshouty, and Mazzawi 2014). In their work, the product terms may not partition the set of variables. The problem of learning discrete distributions over $\{0, 1\}^n$ is considered in (Kearns et al. 1994); some of their results use circuits that compute monotone DNF (but not PM-DNF) functions. Other learning problems for DNF functions have been considered in several papers (e.g., (Angluin and Slonim 1994; Liśkiewicz, Lutter, and Reischuk 2017; Servedio 2004)). The topic of active learning has also been explored in the context of sensor networks (e.g., (Castro and Nowak 2007)).

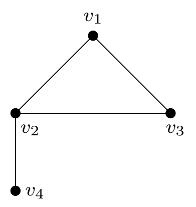
To our knowledge, the problem of learning PM-DNF functions for networked systems has not been addressed in the literature. In particular, the adaptive query techniques presented in (Adiga et al. 2018) for learning symmetric functions (and threshold functions which are a subclass of symmetric functions) cannot be applied to PM-DNF functions since the latter is not a subclass of the former. For example, the PM-DNF function $f(x_1, x_2, x_3, x_4) = x_1\, x_2 + x_3\, x_4$ is not a symmetric function since $f(1, 0, 1, 0) = 0 \neq f(1, 1, 0, 0)$; hence, it is also not a threshold function. Learning threshold functions under the PAC model is considered in (Adiga et al. 2019b); they present a complexity result for efficient consistent learners for threshold functions similar to our result for PM-DNF functions. However, our complexity result is not implied by the one in (Adiga et al. 2019b).

2 Definitions and Problem Formulations

Model for networked social systems. Following (Barrett et al. 2006), we use a formalism called a **synchronous dynamical system** (SyDS), to model a networked social system. Let $\mathbb{B}$ denote the Boolean domain {0,1}. A SyDS $\mathcal{S}$ over $\mathbb{B}$ is a pair $\mathcal{S} = (G, \mathcal{F})$, where (i) $G(V, E)$, an undirected graph with $n = |V|$ nodes, represents the underlying graph of the SyDS, and (ii) $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$ is a collection of functions, with f_i denoting the **local function** at node v_i, $1 \leq i \leq n$. At any time, each node of G has a state value from $\mathbb{B}$. The inputs to function f_i are the states

[1]A membership query specifies an input to a Boolean function and the response is the value of the function.

[2]The square $G^2(V, E^2)$ of a graph $G(V, E)$ has the edge $\{u, v\}$ whenever there is a path of length ≤ 2 between u and v in G.



Node	Local function
v_1	$s_1 + s_2\, s_3$
v_2	$s_1 + s_2 + s_3\, s_4$
v_3	$s_1\, s_2 + s_3$
v_4	$s_2 + s_4$

Figure 1: An example of a PM-DNF-SyDS. The local functions (which are all PM-DNF functions) are shown in the table on the right. Variable s_i represents the state of node v_i, $1 \leq i \leq 4$.

of the nodes in the **closed neighborhood** of v_i (i.e., node v_i and the neighbors of v_i in G). For each input, the output of function f_i gives the next state of v_i. In a SyDS, all nodes compute and update their next state *synchronously* (i.e., in parallel). At any time τ, if $s_i^{\tau} \in \mathbb{B}$ is the state of node v_i ($1 \leq i \leq n$), the **configuration** C of the SyDS is the n-vector $(s_1^{\tau}, s_2^{\tau}, \ldots, s_n^{\tau})$. The system evolves in discrete time steps by repeated application of $\mathcal{F}$. If C and C' denote two successive configurations of a SyDS, then C' is the **successor** of C.

Partitioned monotone DNF functions. In this paper, each local function f_i is based on coalitions formed by the closed neighborhood of node v_i, $1 \leq i \leq n$. Our focus is on one class of such Boolean functions, called **partitioned monotone DNF** (PM-DNF) functions.

Definition 1. A Boolean function f is a PM-DNF iff it has a disjunctive normal form (DNF) (i.e., sum of products) representation satisfying the following two properties: (i) all the variables appear unnegated in f (i.e., f is monotone) and (ii) the collection of product terms (also referred to as **blocks** or **coalitions**) partitions the set of inputs to f; i.e., each input appears in *exactly one* block.

Example 1. Suppose we have five Boolean variables, denoted by x_1, x_2, x_3, x_4 and x_5. One example of a PM-DNF function is $f_1(x_1, x_2, x_3, x_4, x_5) = x_1\, x_3\, x_5 + x_2\, x_4$, which has two product terms (coalitions). Note that the OR function $f_3(x_1, x_2, x_3, x_4, x_5) = x_1 + x_2 + x_3 + x_4 + x_5$ (five coalitions) and the AND function $f_4(x_1, x_2, x_3, x_4, x_5) = x_1\, x_2\, x_3\, x_4\, x_5$ (one coalition) are PM-DNF functions. On the other hand, $f_5(x_1, x_2, x_3, x_4, x_5) = x_1\, x_2 + x_4\, x_5$ is not a PM-DNF function since x_3 doesn't appear in any of the product terms. Likewise, $f_6(x_1, x_2, x_3, x_4, x_5) = x_1\, x_2 + x_3\, \overline{x_4} + x_5$ is not a PM-DNF function since x_4 is negated.

For simplicity, we use the abbreviation PM-DNF-SyDS to denote a SyDS in which every local function is a PM-DNF function. We now present an example of such a SyDS.

Example 2. The graph of a PM-DNF-SyDS is shown in Figure 1. Suppose the initial configuration is $(1, 0, 0, 0)$; that is, node v_1 is in state 1 and nodes v_2, v_3 and v_4 are in state 0. It can be seen that the system goes through the following sequence of configurations during the next two time steps: $(1, 0, 0, 0) \longrightarrow (1, 1, 0, 0) \longrightarrow (1, 1, 1, 1)$. From the configuration $(1, 1, 1, 1)$, no further state changes occur. Such a configuration is a **fixed point** for this system.

Active query model. This query model for SyDSs was proposed in (Adiga et al. 2018). Under this model, each query, which we call a **successor query**, specifies a configuration C; the response to the query is the configuration C', the successor of C. One can think of C as specifying an input to each local function and the response C' as specifying the value of each local function for the input specified by C.

For expository purposes, we consider learning each local function separately. Thus, to learn an unknown PM-DNF function f, a query specifies an assignment α of values to the inputs of f; the response to the query is the Boolean value $f(\alpha)$. In the learning theory literature, such queries are called **membership queries** (see e.g., (Angluin and Slonim 1994)). Since our goal is to use as few membership queries as possible, we will use the **adaptive** query mode considered in (Adiga et al. 2018). In this mode, membership queries are generated one at a time; a query may depend on the responses for previous queries. We also consider learning PM-DNF functions under the PAC model; we refer the reader to (Antony and Biggs 1992; Kearns and Vazirani 1994) for the relevant definitions.

Positive and negative examples. For an unknown PM-DNF f, each example η given to a PAC learner is a pair (α, β), where α is an assignment of $\{0,1\}$ values to the inputs of f and $\beta \in \{0, 1\}$ is the value $f(\alpha)$ of the function. These are *positive* examples. We need not consider negative examples here since a negative example of the form (α, β), that is, "β is not the output of f for input α", is equivalent to the positive example $(\alpha, \overline{\beta})$.

The concept class of PM-DNF functions is **PAC learnable** by a learner L using the hypothesis space H if for any target concept c, values ϵ and δ such that $0 < \epsilon, \delta < 1/2$, and distribution $\mathcal{D}$ over the instance space, L outputs with a probability of at least $1 - \delta$, a hypothesis $h \in H$ such that $\text{error}_{\mathcal{D}}(h) \leq \epsilon$. The **sample complexity** of a learner, denoted by $\mathcal{M}(\epsilon, \delta)$, is the number of examples needed by the learner to output an appropriate hypothesis h. We will use the following well-known upper bound (Haussler 1988) on $\mathcal{M}(\epsilon, \delta)$ based on the size of the hypothesis space H:

$$\mathcal{M}(\epsilon, \delta) \leq \frac{1}{\epsilon}\big(\log|H| + \log(1/\delta)\big). \tag{1}$$

3 Bounds Under the Active Query Model

Lower bound. We establish the lower bound by pointing out that any algorithm that uses membership queries under the adaptive mode can be viewed as a decision tree, like the one used to establish a lower bound on comparison-based sorting algorithms (Cormen et al. 2009). A proof of the following theorem appears in (Adiga et al. 2019a).

Theorem 1. *Every algorithm that uses membership queries under the adaptive mode to learn a PM-DNF function with q inputs must use $\Omega(q \log(q))$ queries in the worst-case.*

Upper bound: A query generation algorithm to learn a PM-DNF function. We now discuss our algorithm for generating membership queries under the adaptive mode to learn an unknown PM-DNF function f with q inputs, denoted by x_1, x_2, ..., x_q. For each block of f, the variable with the smallest index will be referred to as the **key variable** for that block. For example, if one of the blocks is $x_2\, x_7\, x_9$, then the key variable for that block is x_2. For a set of blocks, the **key block** for that set is the block with the largest key variable. For a set of blocks, we define the **superkey variable** for that set of blocks to be the key variable of the key block in the set of blocks.

The algorithm consists of a loop that identifies the blocks of f, one block at a time. For each iteration of this loop, we refer to the already discovered blocks as the **known blocks**, and the remaining blocks as the **unknown blocks**. We refer to the variables in the known blocks as **allocated variables**, and the variables in the unknown blocks as **unallocated variables**. The unallocated variables are sorted by their index, lowest index first. The list of unallocated variables can be considered to be divided into two parts; a left part consisting of **primary unallocated variables**, and a right part (possibly empty) consisting of **secondary unallocated variables**. As the algorithm proceeds, the primary unallocated variables are unallocated variables that are potentially the key variable of some unknown block, whereas secondary unallocated variables are unallocated variables that the responses to previously issued queries have shown are not the key variable of any block.

Initially all the blocks are unknown, and all the variables are primary unallocated variables. At each iteration of the loop, the algorithm finds the key block among the currently unknown blocks. The algorithm does this by first finding the superkey variable for the set of unknown blocks, thereby identifying the key variable of the key unknown block. The current iteration of the loop then proceeds by finding all the remaining variables in the key unknown block, one variable at a time. Once all the variables of the key unknown block have been found, that block is now known, and the status of its variables changes to **allocated**. After changing the status of this block, if all the variables are allocated (i.e., belong to known blocks), then all the blocks of f have been identified, so the algorithm is finished. Otherwise, the algorithm reiterates the loop, to discover the new key unknown block.

Each iteration of the loop consists of two major substeps; the details of these substeps are provided below. Recall that each membership query specifies an assignment α of $\{0,1\}$ values to the variables, and the response to the query is the value $f(\alpha)$. The algorithm uses two types of membership queries: **superkey queries** and **block queries**. Substep 1 of each loop iteration uses superkey queries to find the superkey variable of the unknown blocks. Once this superkey variable is identified, Substep 2 uses block queries to find the other members of the key block.

Substep 1: Finding the superkey variable of the unknown blocks. The algorithm uses a binary search over the primary unallocated variables, using superkey queries to guide the search. The binary search maintains a list L of **candidate variables**, each of which is a primary unallocated variable, and might potentially be the superkey. List L initially consists of all the primary unallocated variables, since the superkey is one of these variables.

If list L of candidate variables contains only one variable, say variable x_k, then the binary search is over, and variable x_k is the superkey. Otherwise, the binary search to find the superkey proceeds as follows. Let x_j be the $\lceil |L|/2 \rceil^{\text{th}}$ variable on list L. Let α^j be the assignment to the q variables where a given variable x_i is 1 iff x_i is unallocated and $i > j$. The algorithm issues α^j as a query, which we refer to as a *superkey query*.

Suppose $f(\alpha^j) = 0$. Then the unallocated variables to the right of x_j do not contain a complete block, so none of these variables can be the key of any unknown block. So, the status of each primary unallocated variable x_i such that $i > j$ is changed to **secondary**. Also, each candidate variable x_i on list L such that $i > j$ is deleted from L.

Suppose $f(\alpha^j) = 1$. Then the unallocated variables to the right of x_j contain a complete block, so the superkey is a candidate variable x_i such that $i > j$. So, each candidate variable x_i on list L such that $i \leq j$ is deleted from L.

In this manner, the search for the superkey variable is recursively continued on the left or right half of list L, depending on the value of $f(\alpha^j)$, until L contains just one variable. Note that each query reduces the size of L by a factor of 2. (More precisely, if L and L' denote respectively the list before and after the list shortening, then $|L'| = \lceil |L|/2 \rceil$). Thus, the number of queries used to find the superkey variable is at most $\lceil \log(q) \rceil$.

Substep 2: Finding the other variables in the key block. Substep 2 uses a loop that searches for the other variables in the key block, one variable at a time. At the beginning of each iteration of this loop, a nonempty set of key block members (including the superkey) have already been found. We refer to this set of variables as **identified key block members**. The iteration begins by issuing a *block query* α wherein a given variable is 1 iff it is an identified key block member. If $f(\alpha) = 1$, then the identified key block members form the complete key block. The key block is now known, so the status of its members is changed to allocated, and Substep 2 is complete.

If $f(\alpha) = 0$, then the key block contains at least one additional member, and a binary search is used to find the additional member with the lowest index. The binary search maintains a list L of **candidate variables**, each of which is a secondary unallocated variable, and which might potentially be the next member of the key block. List L initially consists of all the secondary unallocated variables to the right of the last member added to the key block, since the next member of the key block is one of these variables. If list L of candidate variables contains only one variable, say variable x_j, then the binary search is over, and variable x_j is the next member of the key block. Variable x_j is now the newest identified key block member, and another iteration of the main loop for Substep 2 begins.

Otherwise, if list L of candidate variables contains more than one variable, the binary search to find the next member of the key block proceeds as follows. Let x_j be the $\lceil |L|/2 \rceil^{\text{th}}$ variable on list L. Let α^j be the assignment to the q variables where a given variable x_i is 1 iff either x_i is an identified key block member or x_i is unallocated and $i > j$. The algorithm issues block query α^j.

Suppose $f(\alpha^j) = 0$. Then the key block has a variable x_i such that $i \leq j$ and x_i is on list L. Thus, each candidate variable x_i on list L such that $i > j$ is deleted from L.

Suppose $f(\alpha^j) = 1$. Then the next member of the key block is a variable x_i such that $i > j$ and x_i is on list L. Thus, each candidate variable x_i on list L such that $i \leq j$ is deleted from L.

In this manner, the search for the next member of the key block is recursively continued on the left or right half of list L, depending on the value of $f(\alpha_j)$, until L contains just one variable. Since each query reduces the size of L by a factor of 2, the number of queries used to find the next member of the key block is at most $\lceil \log(q) \rceil$.

Overall, the algorithm uses at most $1 + \lceil \log(q) \rceil$ queries per variable. Thus, an upper bound on the number of queries is $q(1 + \lceil \log(q) \rceil) = O(q \log(q))$. Thus, we have:

Theorem 2. *A PM-DNF function f with q inputs can be learned using at most $q(1 + \lceil \log(q) \rceil) = O(q \log(q))$ adaptive membership queries.* ■

Inferring all local functions. The following theorem provides an upper bound on the number of queries needed to learn all local functions of a PM-DNF-SyDS. A proof of the theorem appears in (Adiga et al. 2019a).

Theorem 3. *For a PM-DNF-SyDS with underlying graph G, $O(\chi(G^2)\, \Delta \log(\Delta))$ successor queries are sufficient to infer all the local functions. Here, Δ is the maximum node degree in G and $\chi(G^2)$ is the minimum number of colors needed for a valid node coloring of G^2.*

4 Results Under the PAC Learning Model

4.1 Upper bound on the number of queries

We begin with an upper bound on the sample complexity to learn a PM-DNF function under the PAC model. A proof of the following result appears in (Adiga et al. 2019a).

Proposition 1. *Let $\epsilon, \delta > 0$ be fixed. The asymptotic sample complexity $\mathcal{M}(\epsilon, \delta)$ for PAC learning all the PM-DNF local functions for a given graph $G(V, E)$ is $\mathcal{M}(\epsilon, \delta) = O\big((2m + n)\log(\Delta + 1)\big)$, where $m = |E|$ and Δ is the maximum degree of G.*

4.2 A complexity result for efficient PAC learning

We will show that a class of PM-DNF functions is not efficiently PAC learnable unless **NP** = **RP**. To do this, we need to introduce the notion of consistency of a PM-DNF function with respect to a set of examples.

Consistent hypothesis. Given a set $\mathcal{E}$ of examples, we say that a hypothesis (i.e., a PM-DNF function) f is **consistent** with respect to $\mathcal{E}$ if for each example $(\alpha, \beta) \in \mathcal{E}$, $f(\alpha) = \beta$. As is well known in the learning theory literature (see e.g., (Kearns and Vazirani 1994)), algorithms for obtaining consistent hypotheses are useful in constructing PAC learning algorithms.

We now present our complexity result for the class of PM-DNF functions with two or more product terms. (As stated in Section 1, the case of a PM-DNF function with one product term is trivial.) To prove the result, we use the following problem which is known to be **NP**-complete (Garey and Johnson 1979).

Hypergraph 2-Colorability (H2C): Given a set $U = \{u_1, u_2, \ldots, u_q\}$ and a collection $Y = \{Y_1, Y_2, \ldots, Y_k\}$ of subsets of U (i.e., the hyperedges) with $|Y_j| \geq 2$, $1 \leq j \leq k$, can the elements in U be colored with two colors so that no hyperedge in Y is monochromatic (i.e., each subset in Y contains at least one element of each color)?

Theorem 4. *If **NP** $\neq$ **RP**, the class of PM-DNF functions with two or more product terms is not efficiently PAC learnable.*

Proof (idea). We use a reduction from H2C to show that if there is an efficient PAC learning algorithm for PM-DNF functions with two or more blocks, then there is an **RP**-time algorithm for H2C, contradicting the assumption that **NP** $\neq$ **RP**. Details appear in (Adiga et al. 2019a). ■

We note that Theorem 4 holds for the case of *proper* learning where the hypothesis class and the concept class are the same, namely the class of PM-DNF functions. Whether the result can be extended to the representation-independent setting (see, e.g., (Warmuth 1989)) is left for future work.

4.3 An ILP-based PAC learning algorithm

As is well known, if a hypothesis h (which in this case is a PM-DNF function) that is consistent with all the given examples can be constructed, then the number of examples used to learn h is within a constant factor of the minimum sample complexity needed to learn the hypothesis class (Blumer et al. 1989). Therefore, we focus on developing an algorithm for a consistent learner. We consider the following problem which we call **Consistent Learning of Partitioned Monotone DNF** functions (CL-PMDNF).

Given: A set $\mathcal{E}$ of examples for an unknown PM-DNF function f with q inputs given by $X = \{x_1, x_2, \ldots, x_q\}$; $\mathcal{E}$ is partitioned into $\mathcal{E}_0$ and $\mathcal{E}_1$, where $\mathcal{E}_0$ ($\mathcal{E}_1$) is the set of examples in which the function value is 0 (1); integer $k \leq q$.
Requirement: Determine whether the variable set X can be partitioned into exactly k blocks, with each block forming a product term of the function, so that the resulting function is consistent with $\mathcal{E}$. If so, find one such partition.

The above formulation assumes that we know the number of blocks. This can be done without loss of generality since we can try the values 1, 2, ..., q for the number of blocks.

Let B_1, B_2, ..., B_k denote the blocks (product terms) of the unknown PM-DNF function. To develop our ILP formulation for CL-PMDNF, let z_{ij} be an indicator variable which has the value 1 if variable x_i is in Block B_j and 0 otherwise, $1 \leq i \leq q$ and $1 \leq j \leq k$. We now explain the constraints in our ILP.

The following two sets of constraints enforce the following requirements: (i) each variable appears in exactly one block and (ii) each block is nonempty (since we must have exactly k blocks).

$$\sum_{j=1}^{k} z_{ij} = 1, 1 \leq i \leq q; \quad \sum_{i=1}^{q} z_{ij} \geq 1, 1 \leq j \leq k\,.$$

Consider any example $\eta_p = (\alpha_p, 0)$ in $\mathcal{E}_0$. Let $S_p \subseteq X$ be the set of variables that have the value 0 in the input assignment α_p. Since the value of the function is 0, each block must have at least one of the variables from S_p. This gives rise to the following set of constraints:

$$\sum_{x_i \in S_p} z_{ij} \geq 1, \quad 1 \leq j \leq k.$$

Consider any example $\eta_r = (\alpha_r, 1)$ of $\mathcal{E}_1$. Let $S_r \subseteq X$ be the set of variables that have the value 0 in the input assignment α_r. Since the value of the function is 1, there is at least one block which does *not* have any of the variables from S_r. To capture this constraint, we introduce k auxiliary $\{0,1\}$ variables, denoted by $b_{r,1}$, $b_{r,2}$, …, $b_{r,k}$, and the following constraints. (Note that each example in $\mathcal{E}_1$ gives rise to a distinct set of auxiliary variables.)

$$b_{r,j} \geq z_{ij},\ \forall i \in S_r; \quad \sum_{j=1}^{k} b_{r,j} \leq k-1\,.$$

It can be verified that the last two sets of constraints together imply that there is a block B_j that does not contain any of the variables in S_r.

Thus, the CL-PMDNF problem is represented by the set of constraints given above along with the following constraints on the variables: (i) $z_{ij} \in \{0,1\}$, for $1 \leq i \leq q$, $1 \leq j \leq k$ and (ii) $b_{r,j} \in \{0,1\}$ for each example η_r in $\mathcal{E}_1$ and $1 \leq j \leq k$. There is a PM-DNF function with k blocks that is consistent with $\mathcal{E}$ iff there is a feasible solution to the above set of constraints.

A PAC learning algorithm from the ILP formulation. Our PAC learning algorithm for PM-DNF functions constructs the ILP from the given set $\mathcal{E}$ of examples and each possible value of k (the number of blocks) and outputs one such function when there is a feasible solution to the ILP.

5 Experimental Results

In this section, we evaluate the algorithms developed in the previous sections and compare the results to the derived bounds. In the case of active query, key aspects that we address are how network and local function structures affect the number of queries. We consider both the size (number of nodes as well as edge density) and structure (regular, scale-free, etc.) of the network. For local functions, we consider different numbers of blocks. For PAC learning, our focus is on the difference between the structures of the inferred partition and the true partition with respect to the number of examples sampled, example distribution, and the number of blocks. We used both synthetic and mined networks from the web for the experiments. As shown in Table 1, five mined networks and three classes of synthetic networks were used. There are five graph instances for each of the random regular (RR) and Barabási-Albert (BA) synthetic (scale-free) networks for a specified edge density.

5.1 Active query

We applied the greedy coloring algorithm in (Kosowski and Manuszewski 2004) to obtain a coloring of the square of the network. The number of colors $C(G^2)$ is shown in Table 1. In each experiment, a PM-DNF function was chosen randomly for each node. Let parameter b denote the maximum

Network	**Properties**			
	n	d_{ave}	Δ	$C(G^2)$
CitHep	34401	24.46	846	847
CoAstro	17903	22	504	505
Jazz	198	27.69	100	109
NRV	769	11.84	20	35
WikiVote	7115	28.32	1065	1082
Star	1001	1.998	1000	1001
RR[a]	1000	10,50,100	10,50,100	$34, 367, 994$
BA[a]	1000	$10, 50, 100$	100,261,374	111,379,780

[a] 5 replicates and Δ, d_{ave} (in case of BA) and $C(G^2)$ values are approximate.

Table 1: Table of networks used in our experiments and their properties. Parameters n, d_{ave} and Δ are the number of nodes, average degree and maximum degree respectively. $C(G^2)$ is the number of colors used to color the square graph G^2 by the greedy coloring scheme.

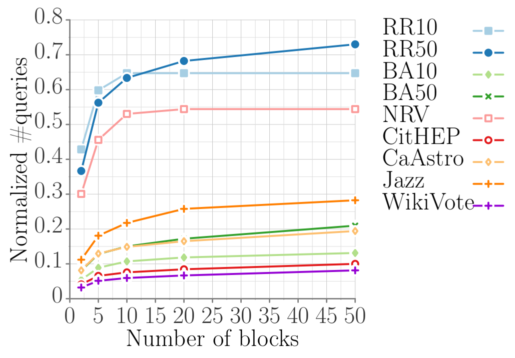


Figure 2: Performance of the active query algorithm on synthetic and mined networks. The Y-axis shows the ratio of the number of queries used to the upper bound given by Theorem 3. The number of queries is averaged over results from 100 repetitions of the experiments. The standard deviation is less than 0.01.

number of blocks possible. The local function was generated using the following iterative process. Blocks were indexed $\{1, 2, \ldots, b\}$. In each iteration, the block index was cyclically incremented. We chose a node uniformly at random without replacement and assigned the block index corresponding to that iteration. For example, suppose $q = 5$ and $b = 3$. Then, there are 5 iterations and the block ID assignment happens in the following order: $[1, 2, 3, 1, 2]$. In the 4th iteration, for example, there are 2 inputs without block ID (since we are sampling without replacement). One of them is chosen randomly and assigned block ID 1. Given this, the algorithm was evaluated using five different values of b (namely 2, 5, 10, 20, and 50) for each network. Each experiment was repeated 100 times.

Effect of network structure. In Figure 2, we note that for a majority of the networks, the number of queries (#queries) required is $< 30\%$ of the upper bound. This is mainly due to the skewed degree distribution of most networks except for random regular networks. Consider the maximum degree of a node in each color class. For scale-free networks,

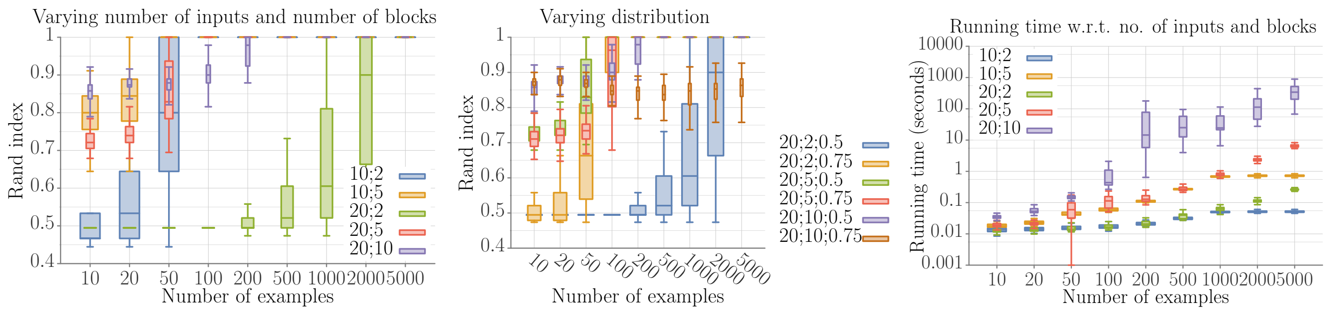


Figure 3: Performance of the ILP algorithm. Each curve in the first and third figures corresponds to $(q; k)$: the number of inputs and true size of the block. In both cases, the results are shown for uniform distribution with $p = 0.5$. The Rand index is averaged over 100 repetitions. In the second figure, each curve corresponds to a distinct (q, k, p). In all cases, the block size parameter for the ILP was set to k.

most color classes have a small maximum degree; therefore, few queries are needed to determine the local functions of vertices in these classes. Interestingly, there is no clear correlation between edge density and #queries. For example, the NRV (New River Valley Friendship) network is much smaller in size, average degree and maximum degree when compared to CoAstro (Coauthorship Astrophysics) network. Yet, when compared with the upper bound, the #queries required is much higher for the NRV network.

Effect of the number of blocks b. As the number of blocks increases, the #queries required also increases. This is because most queries are required to discover the beginning of a block. Also, when $b > \Delta$ (the maximum degree), we observe a plateau due to saturation, as all the partitions contain only blocks of size one. For RR10, RR50 and NRV, there is a sharp increase in #queries between $b = 2$ and 10 as in every color class, $b \log_2 \Delta$ additional queries are required. However, for other networks, because of the skewed degree distribution which leads to many nodes with small degrees, saturation occurs at much lower values of b.

5.2 PAC Model

In the PAC model we restricted our attention to a single local function. The objective is to evaluate the ILP-based algorithm with regard to sample distribution, number of inputs and true block size. The true PM-DNF and the inferred PM-DNF were compared using *Rand index* (Rand 1971). Rand index for two partitions X and Y of a set is $\frac{a+b}{a+b+c+d}$ where a, b, c, and d are respectively the number of pairs of elements (x, y) from the set such that x and y are in (i) the same subset in X and in Y, (ii) different subsets in X and in Y, (iii) same subset in X but different subsets in Y, and (iv) different subsets in X but same subset in Y. The examples were sampled from a uniform distribution over configurations. Each element is set to state 1 independently with the same probability p. The values of p considered were $0.25, 0.5$ and 0.75. Also, we considered input sizes $q = 10$ and 20 and block sizes $k = 2, 5$ and 10. Each experiment was repeated 10 times. We assumed that the inference algorithm has knowledge of the number of blocks in the true partition. Given the number of inputs and number of blocks, we used a similar method as in the active query case to construct the PM-DNF function.

The results in Figure 3 show a rapid increase in the quality of inference with relatively small increments in the number of queries for the case of uniform distribution (Figures 3(a) and (b)). As expected, the greater the number of inputs, the greater is the number of queries required. Interestingly, unlike the active query case, fewer samples are required to infer the local function as the number of blocks increases. This can be explained as follows. Under the uniform distribution, the probability that all elements of a block of size ℓ are 1 is p^{ℓ}; this gives a greater chance of a block being discovered. This is also the reason why as p increases, the chance of discovering a block is higher. However, when p is too high (as in $p = 0.75$), there is a higher chance that a block is hidden in a bigger set of nodes in every example, thus leading to a lower Rand index. Also, we note that as the number of examples is increased, there is an increase in the variance of the Rand index before it is 1 for all repetitions. This variance is higher when the number of blocks is much less than the number of inputs. The running time (Figure 3(c)) steadily increases with the number of examples and blocks.

6 Future Work

It is of interest to extend our results to other types of coalition-based functions; for example, we may require that for the function to have the value 1, at least $k \geq 2$ coalitions must be unanimous. Our complexity result for learning PM-DNF functions is for the case of proper learning where the hypothesis class and the concept class are the same. It is of interest to consider the learning problem under the representation-independent setting. Developing other learning algorithms that can scale to large networks is another direction for future work.

Acknowledgments. We thank the reviewers for carefully reading the manuscript and providing very helpful comments. This work has been partially supported by DTRA CNIMS (Contract HDTRA1-11-D-0016-0001), NSF Grant IIS-1908530, NSF Grant OAC-1916805, NSF CRISP 2.0 Grant 1832587, NSF DIBBS Grant ACI-1443054, NSF BIG DATA Grant IIS-1633028, and NSF EAGER Grant CMMI-1745207. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwith-

standing any copyright annotation thereon.

References

Abasi, H.; Bshouty, N. H.; and Mazzawi, H. 2014. On exact learning monotone DNF from membership queries. *CoRR* abs/1405.0792:1–16.

Adiga, A.; Kuhlman, C. J.; Marathe, M. V.; Ravi, S. S.; Rosenkrantz, D. J.; and Stearns, R. E. 2018. Learning the behavior of a dynamical system via a "20 questions" approach. In *Thirty second AAAI Conference on Artificial Intelligence*, 4630–4637.

Adiga, A.; Kuhlman, C. J.; Marathe, M.; Ravi, S. S.; Rosenkrantz, D. J.; Stearns, R. E.; and Vullikanti, A. 2019a. Learning coalition-based interactions in networked social systems. Technical report, Biocomplexity Institute and Initiative, University of Virginia, Charlottesville, VA.

Adiga, A.; Kuhlman, C. J.; Marathe, M.; Ravi, S. S.; and Vullikanti, A. 2019b. PAC learnability of node functions in networked dynamical systems. In *Proc. ICML 2019*, 82–91.

Angluin, D., and Slonim, D. K. 1994. Randomly fallible teachers: Learning monotone DNF with an incomplete membership oracle. *Machine Learning* 14(1):7–26.

Antony, M., and Biggs, N. 1992. *Computational Learning Theory*. Cambridge, UK: Cambridge University Press.

Barrett, C. L.; Hunt, H. B.; Marathe, M. V.; Ravi, S.; Rosenkrantz, D. J.; and Stearns, R. E. 2006. Complexity of reachability problems for finite discrete dynamical systems. *Journal of Computer and System Sciences* 72(8):1317–1345.

Blumer, A.; Ehrenfeucht, A.; Haussler, D.; and Warmuth, M. K. 1989. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM (JACM)* 36(4):929–965.

Branzei, R.; Dimitrov, D.; and Tijs, S. 2005. *Models in cooperative game theory*. Springer.

Castro, R., and Nowak, R. 2007. Active learning and sampling. In *Proc. Foundations and Applications of Sensor Management*, 177–200.

Centola, D., and Macy, M. 2007. Complex contagions and the weakness of long ties. *American Journal of Sociology* 113(3):702–734.

Colón-Reyes, O.; Jarrah, A. S.; Laubenbacher, R. C.; and Sturmfels, B. 2006. Monomial dynamical systems over finite fields. *Complex Systems* 16(4).

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*. Cambridge, MA: MIT Press and McGraw-Hill, Second edition.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-completeness*. San Francisco: W. H. Freeman & Co.

González-Bailón, S.; Borge-Holthoefer, J.; Rivero, A.; and Moreno, Y. 2011. The dynamics of protest recruitment through an online network. *Scientific Reports* 1:7 pages.

Granovetter, M. 1978. Threshold models of collective behavior. *American Journal of Sociology* 1420–1443.

Haussler, D. 1988. Quantifying inductive bias: AI learning algorithms and valiant's learning framework. *Artificial intelligence* 36(2):177–221.

He, X.; Xu, K.; Kempe, D.; and Liu, Y. 2016. Learning influence functions from incomplete observations. In *Advances in Neural Information Processing Systems*, 2073–2081.

Kearns, M. J., and Vazirani, V. V. 1994. *An Introduction to Computational Learning Theory*. Cambridge, MA: MIT Press.

Kearns, M.; Mansour, Y.; Ron, D.; Rubinfeld, R.; Schapire, R.; and Sellie, L. 1994. On the learnability of discrete distributions. In *Proc. ACM STOC*, 273–282.

Kosowski, A., and Manuszewski, K. 2004. Classical coloring of graphs. *Contemporary Mathematics* 352:1–20.

Laubenbacher, R., and Stigler, B. 2004. A computational algebra approach to the reverse engineering of gene regulatory networks. *J. Theoretical Biology* 229:523–537.

Liśkiewicz, M.; Lutter, M.; and Reischuk, R. 2017. Proper learning of k-term DNF formulas from satisfying assignments. *Electronic Colloquium on Computational Complexity (ECCC)* 24:114.

Narasimhan, H.; Parkes, D. C.; and Singer, Y. 2015. Learnability of influence in networks. In *Advances in Neural Information Processing Systems*, 3186–3194.

Rand, W. M. 1971. Objective criteria for the evaluation of clustering methods. *J. American Statistical Association* 66(336):846 850.

Romero, D.; Meeder, B.; and Kleinberg, J. 2011. Differences in the mechanics of information diffusion across topics: Idioms, political hashtags, and complex contagion on twitter. In *Proceedings of the 20th international conference on World wide web*, 695–704. ACM.

Servedio, R. A. 2004. On learning monotone DNF under product distributions. *Inf. Comput.* 193(1):57–74.

Ugander, J.; Backstrom, L.; Marlow, C.; and Kleinberg, J. 2012. Structural diversity in social contagion. *Proceedings of the National Academy of Sciences* 109(16):5962–5966.

Valiant, L. G. 1984. A theory of the learnable. *Communications of the ACM* 18(11):1134–1142.

Warmuth, M. K. 1989. Towards representation independence in PAC learning. In *Proc. International Workshop on Analogical and Inductive Inference (AII'89)*, 78–103.

Zhu, J.; Zhu, J.; Ghosh, S.; Wu, W.; and Yuan, J. 2018. Social influence maximization in hypergraph in social networks. *IEEE Transactions on Network Science and Engineering*.