Efficient Enumeration of Partial Feature Vectors with Non-Empty Extension

John Berkowitz, Eric Meinhardt

February 2020

1 Definitions and notation

There are four main objects and the connection between them to consider:

1. a finite set of unary predicates ('binary features')
2. a finite set of object types where every type has exactly one unique description in terms of the complete feature set
3. the powerset lattice over the set of objects
4. the bounded semilattice of formulas definable using only conjunctions of literals drawn from the set of unary predicates.

In more detail: consider a finite set of objects $\mathcal{O}$ such that $|\mathcal{O}| = n$. Let $\mathbb{P}(\mathcal{O})$ denote the powerset of $\mathcal{O}$. Note that $|\mathbb{P}(\mathcal{O})| = 2^n$ and that the powerset defines a bounded lattice (where $\leq = \subseteq$) over $\mathcal{O}$, where $\mathcal{O}$ is the global least upper bound and $\emptyset$ is the global greatest lower bound. We will refer to this bounded lattice as 'the powerset lattice (on $\mathcal{O}$)' or as 'the extension lattice' (for reasons clarified shortly).

Suppose all objects are describable using logical formulas containing a finite set of unary predicates ('features') $\mathcal{F}$ where $|\mathcal{F}| = m$, and where every object can be described by exactly one conjunctive clause of m literals, e.g.

$$f_1 \wedge \neg f_2 \wedge \ldots \wedge f_m \tag{1}$$

(By abuse of notation, let $\mathcal{F}(o)$ denote the length-m formula that describes $o \in \mathcal{O}$.) Such formulas can be equivalently thought of in terms of elements of $\mathbb{B}^m$: Boolean vectors associated with some defined ordering over $\mathcal{F}$.

Note that formulas that mention less than m features pick out a (possibly empty) subset of $\mathcal{O}$. (For various reasons (viz. phonology) we are not interested formulas that involve disjunction.) To represent such a formula as a vector v, we need a total ordering of features, plus two bits of information for each feature i: whether the predicate (feature) is mentioned, and if it is mentioned, whether the predicate is present as a positive literal or a negative literal. Let $v_i = +1$ indicate a positive literal, $v_i = -1$ indicate a negative literal, and $v_i = 0$ indicate that no literal mentioning the ith feature is present. We will refer to a vector with no 0s a 'fully-specified feature vector' or 'fsfv'. By further abuse of notation, let $\mathcal{F}(o)$ denote the maximally-specified vector that describes $o \in \mathcal{O}$. (There will be no context where the distinction between conjunctive formulas of literals and vectors is important.)

Let the set of logically possible ternary feature vectors be denoted by V, where $|V| = 3^m$. We will refer to such vectors as 'partially-specified feature vectors', 'partial feature vectors', or simply *pfvs*. Note that they form a *bounded meet semilattice* defined by *specification* where:

- the empty vector $[] = \{0\}^m$ is the global greatest lower bound.
- $u \leq v$ iff $\forall i\ u_i = v_i$ or $u_i = 0$

Intuitively, the meet of $u, v = u \wedge v$ is the vector that contains all the feature value specifications that are common to both u and v. We will refer to the bounded meet semilattice on V as 'the specification semilattice', or simply identify the whole semilattice $(\leq, V)$ with V.

Consider the connection between V and $\mathbb{P}(\mathcal{O})$. The set of objects picked out by a pfv v — its *extension* or *interpretation* — is denoted $[\![v]\!]$. Note that the extension of the empty feature vector — the greatest lower bound of V — is the greatest upper bound of $\mathbb{P}(\mathcal{O})$ (viz. $\mathcal{O}$), and that as you move upwards in the specification semilattice (taking meets), you descend in the extension lattice (taking joins).

The problem: Given a feature set $\mathcal{F}$ and a set of object types $\mathcal{O}$ such that $|\mathcal{O}| << 2^{|\mathcal{F}|}$, how can one efficiently and practically enumerate the set of all partial feature vectors with non-empty extension? (Most logically possible partial feature vectors will point to $\varnothing$ in the extension lattice.)

I actually think I have a nice breadth first search approach to this actually, that is based on doing bfs starting from the all zeros pfv. I implemented it before.

We begin by offering a constructive definition of this set. The n lowest non-empty elements of the extension lattice

$$\{o_1\} \ldots \{o_n\} \tag{2}$$

each contain exactly one element of $\mathcal{O}$. Note that there are exactly n associated vectors in V

$$\mathcal{F}(o_1) \ldots \mathcal{F}(o_n) \tag{3}$$

and these vectors are a subset of the set of highest vectors in the specification seemilattice (viz. all and only the highest vectors that also have non-empty extension). Consider the lower closure of v

$$\downarrow(v) = \{u | u \leq v\} \tag{4}$$

Note that as you descend, the extension of each successive vector is a superset of the last. Accorindgly, $\downarrow(v)$ picks out all pfvs whose extension contains (at the very least) $[\![v]\!]$. The set of all pfvs with non-empty extension can be constructively defined, then, as

$$\bigcup_{o \in \mathcal{O}} \downarrow(\mathcal{F}(o)) \tag{5}$$

From an algorithmic perspective, naive implementation of this definition is grossly inefficient because of the overlap each lower closure has. (Viz. the inefficiency will scale...uh...horrifically with $m = |\mathcal{F}|$ (the specification semilattice will get taller) and less(?) horrifically with $n = |\mathcal{O}|$.)

The challenge: given u, v, and $\downarrow(u)$, how can you efficiently calculate $\downarrow(v)$ — i.e. how can you avoid recomputing as much of $\downarrow(u) \cap \downarrow(v)$ as possible?

2 One idea

Helpful scratch note: consider the meet of $u, v = u \wedge v = glb(u, v)$ and note that

Mumble mumble semilattice homomorphism mumble mumble category theory

$$\downarrow(u \wedge v) = \downarrow(u) \cap \downarrow(v) \tag{6}$$

Related idea: Let $\triangledown : V \times \mathbb{P}([0, m-1]) \rightarrow V$ denote the function that takes a vector and a set of indices, and returns an altered version of the vector with the associated indices unspecified. Naive generation of $\downarrow(v)$ proceeds via

1. Enumerating the set of specified indices of $v = \text{spec}(v) = \{i | v_i \neq 0\}$.
2. Constructing the set of all ways of unspecifying $v = \mathbb{P}(\text{spec}(v))$.
3. Then, $\downarrow(v) = \{\triangledown(v, s) | s \in \mathbb{P}(\text{spec}(v))\}$

If $u \in \downarrow(v)$, then let $\delta(v, u)$ denote the smallest subset s of spec(v) such that

$$\triangledown(v, s) = u \tag{7}$$

Suppose we want to efficiently generate $\downarrow(v)$ and u is a pfv whose lower closure has already been calculated. $u \wedge v$ is easy to calculate; let

$$s = \delta(v, u \wedge v) \tag{8}$$

You can avoid regenerating $\downarrow(u) \cap \downarrow(v) = \downarrow(u \wedge v)$ in the course of generating $\downarrow(v)$ by modifying steps 2-3 above: never unspecify any set of indices s' containing s.

[*FIXME* *need to keep going.*

1. *Given spec v and $s = \delta(v, u \wedge v)$, how do you efficiently generate $\mathbb{P}$(spec$(v)) - s$?*
   (a) *Naive approach: generate $\mathbb{P}$(spec(v)) and then filter.*
   (b) *Any intelligent alternative to this is going to depend on the method of generating $\mathbb{P}$(spec(v)) and (see 2d) when there's a* set *of s's to filter, the structure of that set. Fast generation is going to depend somewhat opaquely on what NumPy does fast.*
      - *Current implementation is based on https://stackoverflow.com/a/42202157.*
      - *There's more to how the output is used than what's described there, but this is one step you might want to change if you weren't going to filter.*
2. *What I've described is about wanting to efficiently generate $\downarrow(v)$ by avoiding recomputation of $\downarrow(v) \cap \downarrow(u)$ where $\downarrow(u)$ has already been computed. Suppose I have a* set *of pfvs U that I've already computed lower closures for. What's the expression for what you want to avoid recomputing and how do you modify the idea above?*
   (a) *Stuff to avoid recomputing $= \bigcup_{u \in U} \downarrow(v) \cap \downarrow(u) = \bigcup_{u \in U} \downarrow(v \wedge u)$.*
   (b) *Does this work out to something nice algebraically?*

(c) *If not, there's still probably a dynamic-programming-flavored way of what $u \in U$ to consider next is best given what you've considered so far in the course of trying to efficiently construct $\downarrow(v)$.*

(d) *Another idea: let U denote the set of fsfvs whose lower closures have been calculated so far. Upon calculating $\downarrow(u)$ and adding u to U, we also calculate $\delta(v, u \wedge v)$ for every remaining $v \in V - U$: we maintain for each 'uncalculated' v information about what regions of its lower closure are already calculated. (There's some seemingly minor further optimization to do here in terms of how you maintain and represent these for each v — e.g. possibly eliminating elements that are subsets/supersets of other ones.)*

   - *There are $O(n^2)$ possible $\delta(\cdot, \cdot)$ values. Each $\delta(\cdot, \cdot)$ value can be represented as a length-m bitmask. Construct an $n \times n \times m$ tensor M, fill it in as you go, and then use NumPy array sorcery to scalably do the filtering in step (1a): if $\mathbb{P}(spec(v))$ has p elements and an associated stack (matrix) P of bitmasks of shape $p \times m$, then taking the dot product of each row of P with each row of $M[index(v), :, :]$ will tell you whether any generated row of P is something you want to ignore. (You also need to then identify + handle all the other rows of P that define supersets of what you just identified.)*

]

3 On efficiently producing $\downarrow(v)$ from u,v, and $\downarrow(u)$

To begin, we consider a sequence of pfvs $\left\{w^t\right\}_{t=0}^{m}$ and a permutation $\pi(i)$ of $\{1, ..., m\}$ with the following properties.

1. $w^0 = u$: The sequence starts at u
2. $w^m = v$: The sequence finishes at v
3. $w_i^t = w_i^{t-1}\ \forall i \neq \pi(t),\ \forall t \in \{1, ..., m\}$. At each $t \geq 0$, the value in position $\pi(t)$ is flipped from it's value in u to it's value in v, until eventually all values have been flipped. We note that if $u_{\pi(t)} = v_{\pi(t)}$ then $w^t = w^{t-1}$

We can efficiently construct $\downarrow(v)$ by sequentially transforming $\downarrow(w^{t-1})$ into $\downarrow(w^t)$ and determining an optimal order of coordinates to flip, π^*.

3.1 Tranforming $\downarrow(w^{t-1})$ into $\downarrow(w^t)$

For the sake of notational brevity, in this section we will fix t and let $i = \pi(t)$. The only difference between w^{t-1} and w^t is at the i^{th} position. This is exactly the difference between the i^{th} coordinates of v and u, which has seven cases:

1. $u_i = v_i$
2. $u_i = 0$ and $v_i = 1$

3. $u_i = 0$ and $v_i = -1$
4. $u_i = 1$ and $v_i = 0$
5. $u_i = -1$ and $v_i = 0$
6. $u_i = 1$ and $v_i = -1$
7. $u_i = -1$ and $v_i = 1$

Practically, the effect of the different cases on lower closure transformation can be divided into four categories.

3.1.1 $u_i = v_i$

In this trivial case, $\downarrow(w^{t-1}) = \downarrow(w^t)$. This takes $O(1)$ operations and $|\downarrow(w^t)| = |\downarrow(w^{t-1})|$.

3.1.2 $u_i = 1$ **or** $u_i = -1$ **and** $v_i = 0$

In this case, $p_i = 0$ or $p_i = u_i (\neq 0)$, $\forall p \in \downarrow(w^{t-1})$. Explicitly, for every $p \in \downarrow(w^{t-1})$ such that $p_i = 0$ there is a $p' \in \downarrow(w^{t-1})$ such that $p'_i = u_i (\neq 0)$ and $p'_j = p_j \; \forall j \neq i$. The same is true vice versa. $\downarrow(w^t) \subseteq \downarrow(w^{t-1})$, specifically $\downarrow(w^t) = \{p | p \in \downarrow(w^{t-1}),\ p_i = 0\}$. Algorithmically, this means filtering out all elements $p \in \downarrow(w^{t-1})$ such that $p_i = u_i (\neq 0)$. This takes $O(|\downarrow(w^{t-1})|)$ operations and $|\downarrow(w^t)| = 0.5 \times |\downarrow(w^{t-1})|$.

3.1.3 $u_i = -v_i \neq 0$

In this case constructing $|\downarrow(w^t)|$ from $|\downarrow(w^{t-1})|$ is simple. $\forall p \in \downarrow(w^{t-1})$ multiply p_i by -1 and add p to $\downarrow(w^t)$. This takes $O(|\downarrow(w^{t-1})|)$ operations and $|\downarrow(w^t)| = |\downarrow(w^{t-1})|$.

3.1.4 $u_i = 0$ **and** $v_i = 1$ **or** $v_i = -1$

In this case, $p_i = 0 \, \forall p \in \downarrow(w^{t-1})$. We note that $\downarrow(w^{t-1}) \subseteq \downarrow(w^t)$. The elements of $\downarrow(w^t) \setminus \downarrow(w^{t-1})$ are formed by taking each $p \in \downarrow(w^{t-1})$ and setting $p_i = v_i$. Algorithmically we make a "copy" of $\downarrow(w^{t-1})$ and for all elements of this copy, set the value of the i^{th} coordinate to v_i. We then add all the elements of this copy into $\downarrow(w^{t-1})$ to produce $\downarrow(w^t)$. This takes $O(|\downarrow(w^{t-1})|)$ operations (provided set addition is constant) and $|\downarrow(w^t)| = 2 \times |\downarrow(w^{t-1})|$.

3.2 The optimal order of flips: π^*

We note that the transformations described above all require $O(|\downarrow(w^{t-1})|)$ operations, with the exception of the case in 3.1.1, which is the identity operation. Thus, in a sequence of transformations each transformation takes time proportional to the size of the lower close set output by the previous transformation. However, the transformation of 3.1.2 *halves* the size of the lower closure set, the transformation of 3.1.3 maintains the size, and the transformation of 3.1.4 *doubles* doubles the size. Thus, the optimal ordering of flips π^* obeys the following structure.

1. All coordinates corresponding to 3.1.1 should be flipped first.
2. Carry out flips and transformations according to the case of 3.1.2. The ordering between instances of this case does not matter.
3. Carry out flips and transformations according to the case of 3.1.3. The ordering between instances of this case does not matter.
4. Carry out flips and transformations according to the case of 3.1.4. The ordering between instances of this case does not matter.

3.3 Zooming out

The last two subsections define a transformation distance $d(u, v)$ between $\downarrow(u)$ and $\downarrow(v)$ for every $u, v \in V$. Ultimately, we want to efficiently generate something equivalent to $\bigcup_{o\in\mathcal{O}} \downarrow(\mathcal{F}(o))$.

Is it a true distance metric? Depends on how you actually define the metric. The transformation cost is not symmetric between u and v unless the number of incidences of 3.1.2 and 3.1.4 are the same.

One idea might be to generate an ordering over the set of objects $o_1, o_2 \ldots o_n$ such that the total

$$\sum_{i=1}^{n-1} d(o_i, o_{i+1}) \tag{9}$$

is minimized. It's not obvious that for a given feature system and set of object types this is actually the shortest way to generate

Goal: prove necessary/sufficient conditions where this is/isn't the case.

$$\bigcup_{o\in\mathcal{O}} \downarrow(\mathcal{F}(o))$$

There might plausibly be distinct (possibly singleton) clusters ('components') of $\mathcal{O}$ where it's better to generate such transformation sequences for some clusters separately.

Related separate thought about average complexity: the last two subsections are defined for arbitrary pairs of pfvs, but in the context of the goal of efficiently generating

$$\bigcup_{o\in\mathcal{O}} \downarrow(\mathcal{F}(o))$$

the u, v pairs where we're interested in transforming $\downarrow(u)$ into $\downarrow(v)$ will typically be fully-specified or close to fully-specified, so cases §3.1.1 and §3.1.3 will be most typical.