

Surprising Pattern Mining in Hypergraph as a Form of Reasoning

Nil Geisweiller¹[0000–1111–2222–3333] and Ben Goertzel^{2,3}[1111–2222–3333–4444]

SingularityNET

Abstract. In this paper we introduce a pattern miner algorithm alongside a definition of surprisingness. The algorithm is framed as a reasoning process, implemented on top the Unified Rule Engine, OpenCog’s rewriting system for carrying mathematical and common sense reasoning. Doing so confer advantages of transparency and flexibility of control. Some initial experiments are conducted on the SUMO ontology.

Keywords: Pattern Miner · Surprisingness · Reasoning.

1 Introduction

Finding patterns is a way of discovering knowledge and learning about data and the world they represent. These patterns can then be used in various ways like constructing predictive models to help to act in this world [REF Shujing work], or to passed on to human expertise. Pattern mining [REF] usually design classes of algorithms capturing *syntactic* features [REF itemset, etc], such as subtree mining [REF], where patterns are subtrees within a data base of trees, and each subtree represents a concept containing all the trees compatible with that subtree. This is both a limit and a strength. Limit because they cannot express arbitrary abstractions, and strength because for that reason they can be relatively efficient. Moreover even purely syntactic pattern miners can go a long way if much of the semantic knowledge is turned into syntax. For instance if the data contains

`human(John)`

`human \Rightarrow mortal`

a purely syntactic pattern miner will not be able to take into account the implicit datum

`mortal(John)`

unless a step of inference is previously taken to make it visible.

Another shortcoming of pattern mining is the volume of patterns it tend to produce. For that reason it can be useful to rank the patterns according to some measure of interestingness [REF].

1.1 Contribution

In this paper we introduce a pattern miner algorithm to find syntactic patterns in hypergraph database alongside a measure of surprisingness. Both are implemented in the OpenCog framework [REF], on top of the *Unified Rule Engine*, URE for short, OpenCog’s reasoning engine. Framing pattern mining as reasoning provides the following advantages:

1. It enables hybridization between syntactic and semantic pattern mining.
2. It offers more transparency. Produced knowledge can be reasoned upon. Reasoning steps taken during mining can be represented as data for subsequent mining as well.
3. It enables meta-learning. By leveraging the URE’s dynamic inference control mechanism [REF] pattern mining itself can be improved.

The last point, although already important by itself, goes further than that. One the motivations to have a pattern miner in OpenCog is to mine inference traces to speed-up reasoning in general. By framing not only pattern mining but more generally learning as reasoning we hope to kickstart a virtuous self-improving cycle. Towards that end, MOSES [REF], OpenCog’s evolutionary program learner, is currently in the process of being ported to the URE as well.

Framing learning as reasoning is not without any drawback as more transparency typically comes with greater computational penalty. However by carefully partitioning transparent reasoning versus opaque computations we hope that an adequate balance between efficiency and open-endedness can be achieved. For instance in the case of evolutionary programming, decisions pertaining to which regions of the program space to explore is best represented as reasoning, given its importance and cost. While more systematic computations such as evaluating the fitness of a candidate can be left as opaque efficient computation. One may draw here a loose analogy with the distinction between conscious and unconscious processes.

1.2 Outline

In Section [REF] a pattern mining algorithm over hypergraphs is presented. Section [REF] is dedicated to frame such algorithm as reasoning. In Section [REF] a measure of Surprisingness is introduced. Preliminary experiments are conducted in Section [REF] on the SUMO ontology. And the conclusion is given in Section [REF].

2 Pattern Mining in Hypergraph Database

2.1 AtomSpace: Hypegraph Database

OpenCog’s primary knowledge repository structure is a labeled hypergraph called the AtomSpace [REF]. It is particularly suited for representing a combination of

symbolic (logical, declarative) knowledge and subsymbolic (probabilistic, connectionistic) knowledge. In the OpenCog terminology, edges are called *links*, vertices are called *nodes*, and *atoms* are either links or nodes. For example one may express that cars are vehicle with

```
(Inheritance (Concept "car") (Concept "vehicle"))
```

if one wishes to express the other way around, how much vehicles are cars, then one can label the inheritance link with a *truth value*

```
(Inheritance (stv 0.4 0.8) (Concept "vehicle") (Concept "car"))
```

where 0.4 represents a probability and 0.8 represents a confidence.

Storing knowledge as hypergraph rather than collections of formulae allows to rapidly retrieve atoms and how they relate to other atoms. For instance given the node (Concept "car") one can rapidly query the links pointing to it, here two, one expressing that cars are vehicle, the other expressing that some vehicles are cars.

2.2 Pattern Matching

OpenCog comes with a *pattern matcher*, a component that can query the atom-space, similar in spirit to SQL, but different in many aspects. For instance queries are themselves programs represented as atoms in the atom-space. This insures reflexivity where queries can be queried or produced by queries. Here's an example of such query

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$Y") (Variable "$Z"))))
```

which fetch any instance of transitivity of inheritance in the AtomSpace. For instance if the AtomSpace contains

```
(Inheritance (Concept "cat") (Concept "mammal"))
(Inheritance (Concept "mammal") (Concept "animal"))
(Inheritance (Concept "square") (Concept "shape"))
```

it will retrieve

```
(Set
  (List (Concept "cat") (Concept "mammal") (Concept "animal")))
```

where `cat`, `mammal` and `animal` are associated to variable `$X`, `$Y` and `$Z` according to the prefix order of the query, but `square` and `shape` will not be retrieved because they do not exhibit transitivity in the data. The construct `Set` represents a set of atoms (here tuples).

The construct `Get` means retrieve. The construct `Present` means that the arguments are patterns to be conjunctively matched against the data present

in the atomspace. We also call **Present** arguments *clauses*, and say that the pattern is a *conjunction of clauses*.

In addition, the pattern matcher can rewrite. For instance a transitivity rule could be implemented with

```
(Bind
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$Y") (Variable "$Z"))))
  (Inheritance (Variable "$X") (Variable "$Z")))
```

The pattern matcher provides the building blocks for the OpenCog's reasoning engine. In fact the URE is, for the most part, pattern matching + unification.

2.3 Pattern Mining as Inverse of Pattern Matching

The pattern miner solves the inverse problem of pattern matcher. That is provided data, it attempts to find queries that would retrieve a certain *minimum* number of matches. This number is called the *support* in the pattern mining terminology.

It is worth mentioning that the pattern matcher has many more constructs than **Get**, **Present** and **Bind**; for declaring types, expressing preconditions, and coding pretty much any computation one may want to express. However the pattern miner only supports a subset of constructs due to the inherent complexity of dealing with such expressiveness.

2.4 Pattern Mining High Level Algorithm

Before showing how to express pattern mining as reasoning, let us explain the algorithm itself.

Our pattern mining algorithm operates like most pattern mining algorithms [2] by searching the space of patterns while pruning the parts that do not have enough support. It typically starts from the most abstract one, the *top* pattern, then constructs specializations of it, only retaining those that have enough support, repeat the specialization on the obtained patterns and so on. The apriori property [1] guaranties that no pattern with enough support will be missed based on the fact that patterns without enough support cannot have specializations with enough support.

Semi-formally, given a database \mathcal{D} , a minimal support S and an initialize collection \mathcal{C} of patterns with their counts equal to or above S , the mining algorithm is as follows

1. select a pattern P from \mathcal{C} ,
2. find all *shallow specializations* \mathcal{E} of P with support equal to or above S ,
3. remove P from \mathcal{C} and add the new patterns \mathcal{E} to \mathcal{C} ,
4. repeat till a termination criterion is met.

The pattern collection \mathcal{C} is usually initialized with the top pattern

```
(Get
  (Present
    (Variable "$X")))
```

that matches the entire database, and from which all subsequent patterns are derived. A shallow specialization is a specialization such that the expansion is only a level deep. For instance, if \mathcal{D} is the 3 inheritances links of Section [REF] (cat is a mammal, a mammal is an animal and square is a shape), a shallow specialization of the top pattern could be

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))))
```

that would still match the entire database, thus have a support of 3. A subsequent shallow specialization of it could be

```
(Get
  (Present
    (Inheritance (Concept "cat") (Variable "$Y"))))
```

that would only match

```
(Inheritance (Concept "cat") (Concept "mammal"))
```

thus have a support of 1.

In practice the algorithm is complemented by heuristics to avoid an exhaustive search, but that is the core of it.

3 Framing Pattern Mining as Reasoning

The hard part of the algorithm above is step 1, selecting which pattern to expand, which determines how the space is being searched. When pattern mining is framed as reasoning such decision corresponds to a *premise selection*.

Let us formalize the type of propositions we need to prove in order to search the space of patterns. Given a database \mathcal{D} and a minimum support S we want is the following theorems

$$S \leq \text{support}(P, \mathcal{D})$$

expressing that pattern P has enough support with respect to the data base \mathcal{D} , or to simplify

$$\text{minsup}(P, S, \mathcal{D})$$

where minsup is defined as

$$\text{minsup}(P, S, \mathcal{D}) := S \leq \text{support}(P, \mathcal{D})$$

Then the primary inference rule we need is

$$\text{minsup}(Q, S, \mathcal{D}) \wedge \text{spec}(Q, P) \vdash \text{minsup}(P, S, \mathcal{D})$$

expressing that if Q has enough support, and Q is a specialization of P , then P has enough support, essentially formalizing the apriori property. We can either apply such rule in a forward way, from left to right, or in a backward way, from right to left. If we search from more abstract to more specialized then we want to use it in a backward way. Meaning the reasoning engine needs to choose P and then construct a specialization of it, Q . In practice we actually have rewritten that rule in a backward way that we apply it in a forward for efficiency reasons, although the underlying idea is the same, which is why in this context select P amounts to selecting a premise of the $\text{minsup}(P, S, \mathcal{D})$. The exact definition of spec is left out, it is merely a variation of the subtree relationship while accounting for variables. As one can see a pattern such as

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))))
```

is different than

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$X"))))
```

where $\$Y$ has been replace by $\$X$, which requires some additional care.

Other *heuristic* rules can be used to infer knowledge about minsup . They are heuristics because unlike the apriori property, they do not guaranty completeness, but can speed-up the search by eliminating large portions of the search space. For instance the following rule

$$\text{minsup}(P, S, \mathcal{D}) \wedge \text{minsup}(Q, S, \mathcal{D}) \wedge R(P \otimes Q) \vdash \text{minsup}(P \otimes Q, S, \mathcal{D})$$

expresses that if P and Q have enough support, and a certain combination of P and Q has a certain property R , then such combination has enough support. Such rule is used to combine multiple conjunctions of patterns, that if for instance given P and Q both being

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))))
```

One can combine them to form

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$Y") (Variable "$Z"))))
```

or

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$X") (Variable "$Z"))))
```

etc. The property R here is that both clauses must share at least one joint variable and the combination must have its support above or equal to the minimum threshold.

4 Surprisingness

Even with the help of the apriori property and additional heuristics to prune the search, the volume of found patterns can still be overwhelming. For that it can be helpful to assign to the patterns a measure of *interestingness*. Such notion is broad and can mean various things in various contexts. We will restrict our attention on the sub-notion of *surprisingness*. Although still broad one can define surprisingness as what is *contrary to expectations*. Such notion is important as if some pattern contradicts expectations it means we have uncovered an incorrectness or incompleteness in our model of the world.

Just like for pattern mining, surprisingness can be framed as reasoning. There are many ways to formalize such notion. We suggest that in its most general sense, surprisingness may be the difference of outcome between different inferences over the same conjecture. Of course in most logical systems, if consistent, different inferences will produce the same result. However in para-consistent systems, such as PLN [5] for *Probabilistic Logic Network*, OpenCog's logic for common sense reasoning, different inference outcomes on the same conjecture are possible. In particular PLN allows a conjecture to be believed with various degrees of truth, ranging from total ignorance to absolute certainty. Thus PLN is particularly well suited for such definition of surprisingness. More specifically we define surprisingness as the *distance of truth values between different inferences over the same conjecture*. In PLN a truth value is a second order distribution, probabilities over probabilities, as explained in Chapter 4 of [5]. Second order distributions are good at capturing uncertainties. Total ignorance is represented by a flat distribution (Bayesian prior), or possibly a slightly concave one (Jeffreys prior [6]), and absolute certainty by a Dirac delta function.

Such definition of surprisingness has the merit of encompassing a wide variety of cases; like the surprisingness of finding a proof contradicting human intuition. For instance the outcome of Euclid's proof of the infinity of prime numbers might contradict the intuition of a beginner upon observation that prime numbers rapidly rarefy as numbers grow. It also encompasses the surprisingness of observing an unexpected event, or the surprisingness of discovering a pattern in seemingly random data. All these cases can be framed as ways of constructing different types of inferences and finding contradictions between them. For instance in the case of discovering a pattern in a data set, one inference could

calculate the empirical probability based on the data, while an other inference could calculate a probability estimate based on the independence of the variables and their marginal probabilities.

To be closer to the human notion of surprisingness one may refine this definition by having one type of inference obtained from the current model of the world from which rapid (and usually uncertain) conclusions can be derived, and the other type of inference provided by the world itself, either via observations, in the case of an experienced reality, or via crisp and long chains of deductions in the case of a mathematical reality.

The distance measure to use to compare conjecture outcomes remains to be defined. Since our truth values are distributions the Jensen-Shannon distance [4], also suggested as surprisingness measure in [7] and [3], could be used. The advantage of such distance is that it accounts well for uncertainty. If for instance a pattern is discovered in a small data set displaying high levels of dependencies between variables (thus surprising relative to an independence assumption of these variables), the surprisingness measure should consider the possibility that it might be a fluke since the data set is small. Fortunately, the smaller the data set, the flatter the second order distributions representing the empirical and the estimated truth values of the pattern, automatically reducing the Jensen-Shannon distance.

Likewise one can imagine the following coin tossing experiments. In the first experiment a coin is tossed 3 times, a probability p_1 of head is calculated, then the coin is tossed 3 more times, a second probability p_2 of head is calculated. p_1 and p_2 might be very different, but it should not be surprising given the low number of observations. On the contrary, in the second experiment the coin is tossed a billion times, p_1 is calculated, then another billion times, p_2 is calculated. Here even tiny differences between p_1 and p_2 should be surprising. In both cases, by representing p_1 and p_2 as second order distributions rather than mere probabilities, the Jensen-Shannon distance properly accounts for the uncertainty.

From now on, for sake of brevity, when speaking of *probability* we actually mean *second order probability distribution*. When the probability is estimated by a Bernoulli process (which is the case in most of the remaining uses) the second order probability distribution can easily be obtained by considering a Beta-distribution [REF].

4.1 Independence-based Surprisingness

Here we explore a simple form of surprisingness based on the independence of the clauses within a pattern called I-Surprisingness for Independence-Surprisingness. For instance

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$Y") (Variable "$Z"))))
```

has two clauses


```
(Inheritance (Variable "$X") (Variable "$Y"))
```

and

```
(Inheritance (Variable "$Y") (Variable "$Z"))
```

If each clause is considered independently, that is the distribution of values taken by the variable tuples $(\$X, \$Y)$ appearing in the first clause is independent from the distribution of values taken by the variable tuples $(\$Y, \$Z)$ in the second clause, one can simply use the product of the two distributions to obtain an estimate of the distribution of their conjunctions. However the presence of joint variables, here $\$Y$, makes it obviously wrong. Instead the connections need to be taken into account. To do that we use the fact that a pattern of connected clauses is equivalent to a pattern of disconnected clauses combined with a condition of equality between the joint variables. For instance

```
(Get
  (Present
    (Inheritance (Variable "$X") (Variable "$Y"))
    (Inheritance (Variable "$Y") (Variable "$Z"))))
```

is equivalent to

```
(Get
  (And
    (Present
      (Inheritance (Variable "$X") (Variable "$Y1"))
      (Inheritance (Variable "$Y2") (Variable "$Z"))))
    (Equal (Variable "$Y1") (Variable "$Y2"))))
```

where the joint variables, here $\$Y$, have been replaced by variable occurrences in each clause, $\$Y1$ and $\$Y2$. The construct `Equal` tests the equality of values, and the construct `And` is the logical conjunction. So the whole query means, find values matching these two pattern, such that each value associated to $\$Y1$ is equal to the value associated to $\$Y2$. Then we can express the probability estimate as the product of the probabilities of the clauses, as if they were independent, times the probability of having the values of joint variables equal. The formula estimating the probability of equality is not detailed here due to lack of place but basically it involves some limited form of static analysis to estimate the number of value a particular variable occurrence can take. More detail is provided in the comment of the code [REF].

In the end we obtain the following inference rule for calculating I-Surprisingness

$$\text{minsup}(P, S, \mathcal{D}) \vdash \text{isurp}(P, \mathcal{D}, f)$$

where `isurp` is a predicate relating a pattern P and its database \mathcal{D} to its I-Surprisingness f , the Shannon-Jensen divergence between the empirical (second order) probability of P and its (second order) probability estimate, as described above. In practice the calculation of f is done in C++ for efficiency. That is

the URE calls that rule after running the pattern miner, which itself calls the C++ code. This provides another example of how learning can be partitioned into transparent reasoning and opaque computation.

Of course after discovering a few surprising patterns based on such independence assumption, the data should be expected to contain such dependencies, and subsequent estimates should take that into account when calculating of the surprisingness. For instance once the empirical probability of a pattern known, if such a pattern was surprising such knowledge should be retained and utilized, so that any other pattern that have its probability estimate rapidly derivable from previously surprising patterns, and such that this estimate is good (the distance between the empirical probability and the estimate is small) should not be considered surprising.

For that the current I-Surprisingness rule needs to be decomposed so that the estimate can be calculated in a separate inference, the empirical probability in a separate inference, and the two compared in a separated Jensen-Shannon distance step.

5 Conclusion

It is clear from that experiment that the next step is to enable some form of open-ended (yet reasonably fast) means of inferences as to take into account more semantic knowledge, and thus .

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. Proceedings of the 20th International Conference on Very Large Data Bases (1994)
2. Chi, Y., Muntz, R., Nijssen, S., N. Kok, J.: Frequent subtree mining - an overview. *Fundam. Inform.* **66**, 161–198 (01 2005)
3. Derezinski, M., Rohanimanesh, K., Hydrie, A.: Discovering surprising documents with context-aware word representations. Proceedings of the 23rd International Conference on Intelligent User Interfaces, IUI 2018, Tokyo, Japan, March 07-11, 2018 pp. 31–35 (2018)
4. Endres, D., Schindelin, J.: A new metric for probability distributions. *Information Theory, IEEE Transactions on* **49**, 1858 – 1860 (08 2003)
5. Goertzel, B., Ikle, M., Goertzel, I.F., Heljakka, A.: Probabilistic Logic Networks. Springer US (2009)
6. Jeffreys, H.: An Invariant Form for the Prior Probability in Estimation Problems. *Proceedings of the Royal Society of London Series A* **186**, 453–461 (1946)
7. Pienta, R., Lin, Z., Kahng, M., Vreeken, J., Talukdar, P.P., Abello, J., Parameswaran, G., Chau, D.H.P.: Adaptivenav: Discovering locally interesting and surprising nodes in large graphs. *IEEE VIS Conference (Poster)* (2015)