An Inferential Approach to Mining Surprising Patterns in Hypergraphs

Nil Geisweiller and Ben Goertzel

SingularityNET Foundation, The Netherlands
{nil,ben}@singularitynet.io

Abstract. A novel pattern mining algorithm and a novel formal definition of surprisingness are introduced, both framed in the context of formal reasoning. Hypergraphs are used to represent the data in which patterns are mined, the patterns themselves, and the control rules for the pattern miner. The implementation of these tools in the OpenCog framework, as part of a broader multi-algorithm approach to AGI, is described.

Keywords: Pattern Miner · Surprisingness · Reasoning · Hypergraphs.

1 Introduction

Pattern recognition is broadly recognized as a key aspect of general intelligence, as well as of many varieties of specialized intelligence. Finding patterns is a way of discovering knowledge and learning about the data the world they represents. These patterns can then be used in various ways like constructing predictive models to help agents act in this world [9], or they can be passed on to human expertise. General intelligence can be envisioned, among other ways, as the process of an autonomous agent recognizing patterns in itself and its environment, including patterns regarding which of its actions tend to achieve which goals in which contexts [5].

The scope of pattern recognition algorithms in AI and allied disciplines is very broad, including many specialized algorithms aimed at recognizing patterns in particular sorts of data such as visual data, auditory data or genomic data. Among more general-purpose approaches to pattern recognition, so-called "pattern mining" plays a prominent role. Mining here refers to the process of systematically searching a body of data to find a large number of patterns satisfying certain criteria. Most pattern mining algorithms are greedy in operation, meaning they start by finding simple patterns and then try to combine these to guide their search for more complex patterns, and iterate this approach a few times.

Pattern mining algorithms tend to work at the *syntactic* level, such as subtree mining [2], where patterns are subtrees within a database 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

$$\texttt{human(John)}$$

$$\texttt{human} \Rightarrow \texttt{mortal}$$

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

$$\texttt{mortal(John)}$$

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

Another shortcoming of pattern mining is the volume of patterns it tends to produce. For that reason it can be useful to rank the patterns found according to some measure of interestingness [12]. One can also use pattern mining in combination with other pattern recognition techniques, e.g. evolutionary programming or logical inference.

Here we present a novel approach to pattern mining that combines semantic with syntactic understanding of patterns, and that uses a sophisticated measure of pattern surprisingness to filter the combinatorial explosion of patterns. The surprisingness measure and the semantic aspect of patterns are handled via embedding the pattern mining process in an inference engine, operating on a highly general hypergraph-based knowledge representation.

1.1 Contribution

We introduce a pattern miner algorithm designed to find patterns in hypergraph database, alongside a measure of surprisingness designed for use with this pattern miner. Both are implemented on the OpenCog framework [6], on top of the *Unified Rule Engine*, URE for short, the reasoning engine of OpenCog.

Framing pattern mining as reasoning provides the following advantages:

1. Enable hybridizations between syntactic and semantic pattern mining.
2. Allow to handle the full notion of surprisingness, as will be further shown.
3. Offer more transparency. Produced knowledge can be reasoned upon. Reasoning steps selected during mining can be represented as data for subsequent mining and reasoning.
4. Enable meta-learning by leveraging URE's inference control mechanism.

The last point, although already important as it stands, goes further than it may at first seem. One of the motivations to have a pattern miner in OpenCog is to mine inference traces, discover control rules out of them, and apply these control rules 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-improvement cycle. Towards that end more components of OpenCog, such as MOSES [8], an evolutionary program learner, are currently in the process of being ported to the URE as well.

Framing learning as reasoning is not without drawbacks, as more transparency comes at a greater computational cost. However by carefully partitioning transparent/costly versus opaque/efficient 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 what regions of the program space to explore is best processed as reasoning, given the importance and the cost of such operation. While more systematic operations such as evaluating the fitness of a candidate can be left as opaque. One may even draw a speculative analogy here with the distinction between conscious and unconscious processes.

1.2 Outline

In Section 2 a pattern mining algorithm over hypergraphs is presented; it is framed as reasoning in Section 3. In Section 4 a definition of surprisingness is provided, and a more specialized implementation is derived from it. Then, in Section 5 an example of how it can be framed as reasoning is presented, both for the specialized and abstract definitions of surprisingness.

2 Pattern Mining in Hypergraph Database

2.1 AtomSpace: Hypergraph Database

Let us first rapidly recall what is the AtomSpace [6], the hypergraph knowledge store with which we shall work here. The AtomSpace is the OpenCog AGI framework's primary data storage solution. It is a labeled hypergraph particularly suited for representing symbolic knowledge, but is also capable of representing sub-symbolic knowledge (probabilities, tensors, etc), and most importantly combinations of the two. In the OpenCog terminology, edges of that hypergraph are called *links*, vertices are called *nodes*, and *atoms* are either links or nodes.

For example one may express that cars are vehicles with

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

`Inheritance` is a link connecting two concept nodes, `car` and `vehicle`. If one wishes to express the other way around, how much vehicles are cars, then one can attach the inheritance 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 query atoms and how they relate to other atoms. For instance given the node `(Concept "car")` one can 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 AtomSpace, similar in spirit to SQL, but different in several aspects. For instance queries are themselves programs represented as atoms in the AtomSpace. This insures reflexivity where queries can be queried or produced by queries.

Here's an example of such a query

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

which fetches 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 retrieves

```
(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` are not retrieved because they do not exhibit transitivity. The construct `Set` represents a set of atoms, and `List` in this context represents tuples of values. 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 reasoning engine. In fact the URE is, for the most part, pattern matching + unification. The collection of atoms that can be executed in OpenCog, to query the atomspace, reason or such, forms a language called *Atomese*.

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 [1, 2].

It is worth mentioning that the pattern matcher has many more constructs than `Get`, `Present` and `Bind`; for declaring types, expressing preconditions, and implementing 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 High Level Algorithm of the Pattern Miner

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 a greedy algorithm of searching the space of frequent patterns while pruning the parts that do not have enough support. It typically starts from the most abstract one, the *top* pattern, constructing specializations of it and only retain those that have enough support, then repeat. 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.

Let us semi-formalize the above. Given a database $\mathcal{D}$, a minimal support S and an initialize collection $\mathcal{C}$ of patterns with enough support, the mining algorithm operates as follows

1. Select a pattern P from $\mathcal{C}$.
2. Produce a *shallow specialization* Q of P with support equal to or above S.
3. Add Q to $\mathcal{C}$, remove P if all its shallow specializations have been produced.
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 whole database, and from which all subsequent patterns are specialized. 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 Subsection 2.2 (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"))))
```

which would match all inheritance links, thus have a support of 3. A subsequent shallow specialization of it could be

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

which would only match

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

and have a support of 1. So if the minimum support S is 2, this one would be discarded. In practice the algorithm is complemented by heuristics to avoid exhaustive search, but that is the crux of it.

3 Framing Pattern Mining as Reasoning

The hardest part of the algorithm above is step 1, selecting which pattern to expand; this has the biggest impact on how the space is explored. When pattern mining is framed as reasoning such decision corresponds to a *premise or conclusion selection.*

Let us formalize the type of propositions we need to prove in order to search the space of patterns. For sake of conciseness we will use a hybridization between mathematics and Atomese, it being understood that all can be formalized in Atomese. Given a database $\mathcal{D}$ and a minimum support S we want to instantiate and prove the following theorems

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

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

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

The primary inference rule we need is (given in Gentzen style),

$$\frac{\texttt{minsup}(Q, S, \mathcal{D}) \qquad \texttt{spec}(Q, P)}{\texttt{minsup}(P, S, \mathcal{D})}\,(\mathrm{AP})$$

expressing that if Q has enough support, and Q is a specialization of P, then P has enough support, essentially formalizing the apriori property (AP). We can either apply such rule in a forward way, top-down, or in a backward way, bottom-up. If we search from more abstract to more specialized we want to use it in a backward way. Meaning the reasoning engine needs to choose P (*conclusion selection* from $\texttt{minsup}(P, S, \mathcal{D})$) and then construct a specialization Q. In practice that rule is actually written backward so that choosing P amounts to a *premise selection*, but is presented here this way for expository purpose. The definition of `spec` is left out, but it is merely a variation of the subtree relationship while accounting for variables (the same variable may occur at different places in the pattern).

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

$$\frac{\texttt{minsup}(P, S, \mathcal{D}) \qquad \texttt{minsup}(Q, S, \mathcal{D}) \qquad R(P \otimes Q)}{\texttt{minsup}(P \otimes Q, S, \mathcal{D})} \text{ (CE)}$$

expresses that if P and Q have enough support, and a certain combination $P \otimes Q$ has a certain property R, then such combination has enough support. Such rule can be used to build the conjunction of multiple patterns. For instance given P and Q both equal to

```
(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"))))
```

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 mined patterns can still be overwhelming. For that it is helpful to assign to the patterns a measure of *interestingness*. This is a broad notion and we will restrict our attention to the sub-notion of *surprisingness*, that can be defined as what is *contrary to expectations*.

Just like for pattern mining, surprisingness can be framed as reasoning. They are many ways to formalize it. We tentatively suggest that in its most general sense, surprisingness may be the considered as the difference of outcome between different inferences over the same conjecture.

Of course in most conventional logical systems, if consistent, different inferences will produce the same result. However in para-consistent systems, such as PLN for *Probabilistic Logic Network* [4], OpenCog's logic for common sense reasoning, conflicting outcomes are possible. In particular PLN allows a proprosition to be believed with various degrees of truth, ranging from total ignorance to absolute certainty. Thus PLN is 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, Chapter 4 of [4]. 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 [7]), 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 database, one inference could calculate the empirical probability based on the data, while an other inference could calculate a probability estimate based on variable independences.

The distance measure to use to compare conjecture outcomes remains to be defined. Since our truth values are distributions the *Jensen-Shannon Distance*, JSD for short [3], suggested as surprisingness measure in [11], 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), 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, reducing the JSD.

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 seems to adequatly accounts for the uncertainty.

A slight refinement of our definition of surprisingness, probably closer to human intuition, can be obtained by fixing one type of inference provided by the current model of the world from which rapid (and usually uncertain) conclusions can be derived, and the other type of inference implied by the world itself, either via observations, in the case of an experiential reality, or via crisp and long chains of deductions in the case of a mathematical reality.

4.1 Independence-based Surprisingness

Here we explore a limited form of surprisingness based on the independence of the variables involved in the clauses of a pattern, called I-Surprisingness for Independence-based 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 probabilities to obtain an probability estimate of their conjunctions. However the presence of joint variables, here `$Y`, makes this calculation 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. 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 the joint variables equal. The formula estimating the probability of equality is not detailed here, but according to our preliminary experiments, a decent estimate can be obtained by performing some syntactic analysis to take into account specializations between clauses relative to their joint variables.

5 I-Surprisingness Framed as Reasoning and Beyond

The proposition to infer in order to calculate surprisingness is defined as

$$\mathtt{surp}(P, \mathcal{D}, s)$$

where surp is a predicate relating the pattern P and the database $\mathcal{D}$ to its surprisingness s, defined as

$$s := \mathtt{dst}(\mathtt{emp}(P, \mathcal{D}), \mathtt{est}(P, \mathcal{D}))$$

where dst is the Jensen-Shannon distance, emp is the empirical second order distribution of P, and est its estimate.

The calculation of $\mathtt{emp}(P, \mathcal{D})$ is easily handled by a *direct evaluation* rule that uses the support of P and the size of $\mathcal{D}$ to obtain the parameters of the beta-binomial-distribution describing its second order probability. However, the mean by which the estimate is calculated is let unspecified. This is up to the reasoning engine to find an inference path to calculate it. Below is an example of inference tree to calculate surp based on I-Surprisingness

$$\dfrac{P \qquad \mathcal{D} \qquad \dfrac{\dfrac{P \qquad \mathcal{D}}{\mathtt{emp}(P, \mathcal{D})}\,(\mathrm{DE}) \qquad \dfrac{P \qquad \mathcal{D}}{\mathtt{est}(P, \mathcal{D})}\,(\mathrm{IS})}{\mathtt{dst}(\mathtt{emp}(P, \mathcal{D}), \mathtt{est}(P, \mathcal{D}))}\,(\mathrm{JSD})}{\mathtt{surp}(P, \mathcal{D}, \mathtt{dst}(\mathtt{emp}(P, \mathcal{D}), \mathtt{est}(P, \mathcal{D})))}\,(\mathrm{S})$$

where

- (S) is a rule to construct the surp predicate,
- (JSD) is a rule to calculate the Jensen-Shannon Distance,
- (DE) is the direct evaluation rule to calculate the empirical second order probability of P according to $\mathcal{D}$,
- (IS) is a rule to calculate the estimate of P based on I-Surprisingness described in Section 4.1.

That inference tree uses a single rule (IS) to calculate the estimate. Most rules are complex, such as (JSD), and actually have the heavy part of the calculation coded in C++ for maximum efficiency. So all that the URE must do is put together such inference tree, which can be done reasonably well given how much complexity is encapsulated in the rules.

As of today we have only implemented (IS) for the estimate. In general, however, we want to have more rules, and ultimately enough so that the estimate can be inferred in an open-ended way. In such scenario, the inference tree would look very similar to the one above, with the difference that the (IS) rule would be replaced by a combination of other rules. Such approach naturally leads to a dynamic surprisingness measure. Indeed, inferring that some pattern is I-Surprising requires to infer its empirical probability, and this knowledge can be further utilized to infer estimates of related patterns. For instance, if say an

I-Surprising pattern is discovered about pets and food. A pattern about cats and food might also be measured as I-Surprising, however the fact that cat inherits pet may lead to constructing an inference that estimates the combination of cat and food based on the combination of pet and food, possibly leading to a much better estimate, and thus decreasing the surprisingness of that pattern.

6 Discussion

The ideas presented above have been implemented as open source C++ code in the OpenCog framework, and have been evaluated on some initial test datasets, including a set of logical relationships drawn from the SUMO ontology [10]. The results of this empirical experimentation are omitted here for space reasons and will be posted online as supplementary information. These early experiments provide tentative validation of the sensibleness of the approach presented: using inference on a hypergraph based representation to carry out pattern mining that weaves together semantics and syntax and is directed toward a sophisticated version of surprisingness rather than simpler objective functions like frequency.

Future work will explore applications to a variety of practical datasets, including empirical data and logs from an inference engine; and richer integration of these methods with more powerful but more expensive techniques such as predicate logic inference and evolutionary learning.

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. Endres, D., Schindelin, J.: A new metric for probability distributions. Information Theory, IEEE Transactions on **49**, 1858 – 1860 (08 2003)
4. Goertzel, B., Ikle, M., Goertzel, I.F., Heljakka, A.: Probabilistic Logic Networks. Springer US (2009)
5. Goertzel, B., Pennachin, C., Geisweiller, N.: Engineering General Intelligence, Part 1: A Path to Advanced Agi Via Embodied Learning and Cognitive Synergy. Atlantis Press (2014)
6. Goertzel, B., Pennachin, C., Geisweiller, N.: Engineering General Intelligence, Part 2: The CogPrime Architecture for Integrative, Embodied AGI. Atlantis Press (2014)
7. 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)
8. Looks, M., Sc, B., Missouri, S.L., Louis, S.: Abstract competent program evolution by moshe looks (2006)
9. O'Neill, J., Goertzel, B., Ke, S., Lian, R., Sadeghi, K., Shiu, S., Wang, D., Yu, G.: Pattern mining for general intelligence: The fishgram algorithm for frequent and interesting subhypergraph mining. In: Bach, J., Goertzel, B., Iklé, M. (eds.) Artificial General Intelligence. pp. 189–198. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

10. Pease, A.: Ontology: A practical guide. Articulate Software Press, Angwin, CA (01 2011)
11. 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)
12. Vreeken, J., Tatti, N.: Interesting Patterns, pp. 105–134. Springer International Publishing, Cham (2014)