


Chapter 1



Related Work

1.1 Statistics of Formal Systems

The core problem of assigning “interestingness” to logical formulae is the application of statistical reasoning to the discrete, semantically-rich domain of formal systems. This problem has been tackled from various directions  or a variety of reasons; here we summarise those contributions which **seem of** particular importance for theory exploration.

1.1.1 Probability of Statements

An important property of a logical formula (in general, and in particular with regards to interestingness) is its truth value. Although we may be able to determine some truth values exactly, e.g. via deduction, it may only be possible (or more efficient) to *approximate* truth values. One straightforward approximation is to adopt *probabilities*, where we assign probability 1 to formulae which are known to be true (e.g. axioms), 0 to formulae known to be false, and intermediate values to those which are not known.

This can be useful for “knowledge ases” of non-logical axioms (e.g. the observed colours of Hempel’s ravens  and allows the probability of universal statements to be inferred from non-exhaustive cases [8]. However, inferring consistent probabilities requires *logical omniscience*: knowledge of all the logical consequences of the chosen axioms. This is unfortunate, since it is precisely what we are trying to avoid by approximating with probabilities.

Avoiding this requirement for logical omniscience has given rise to the field of *bounded rationality*, where reasoning (deduction, induction, etc.) is subject to some computational constraints. For example, the “logical inductor” of [2] avoids assigning particular probabilities to formal statements, and instead produces a *sequence* of successive approximate probabilities, each computable with finite (though doubly-exponential!) resources. These approximations are only guaranteed to be consistent in the limit.

The probability of a statement can affect its interestingness in a few ways. Statements whose probability is very close to 0 or 1 may be considered less interesting, since they are essentially known (e.g. easily derivable or refutable via known facts). Those closer to $\frac{1}{2}$ may be considered more interesting; although in a bounded rationality context this may simply indicate the requirement for some routine (but lengthy) computation: for example stating that the n th bit of π is a zero, for some large n .

1.1.2 Learning From Structured Data

One major difficulty with formal mathematics as a domain in which to apply statistical machine learning is the use of *structure* to encode information in objects. In particular, *trees* appear in many places: from inductive datatypes, to recursive function definitions; from theorem statements, to proof objects. Such nested structures may extend to arbitrary depth, which makes them difficult to represent with a fixed number of features, as is expected by most machine learning algorithms. Here we review a selection of solutions to this problem, and compare their distinguishing properties.

Feature Extraction

Feature extraction is a common pre-processing step for machine learning (ML). Rather than feeding “raw” data straight into our ML algorithm, we only learn a sample of *relevant* details, known as *features*. This has two benefits:

- *Feature vectors* (ordered sets of features) are chosen to be more compact than the data they’re extracted from: feature extraction is *lossy compression*. This reduces the size of the ML problem, improving efficiency (e.g. running time).
- We avoid learning irrelevant details such as the encoding system used, improving *data* efficiency (the number of samples required to spot a pattern).
- All of our feature vectors will be the same size, i.e. they will all have length (or *dimension*) d . Many machine learning algorithms only work with inputs of a uniform size; feature extraction allows us to use these algorithms in domains where the size of each input is not known, may vary or may even be unbounded. For example, element-wise comparison of feature vectors is trivial (compare the i th elements for $1 \leq i \leq d$); for expressions this is not so straightforward, as their nesting may give rise to very different shapes.
- Unlike our expressions, which are discrete, we can continuously transform one feature vector into another. This enables many powerful machine learning algorithms to be used, such as those based on *gradient descent* or, in our case, arithmetic means.

- Feature vectors can be chosen to represent the relevant information in a more compressed form than the raw data; for example, we might replace verbose, descriptive identifiers with sequential numbers. This reduces the input size of the machine learning problem, improving efficiency.

Another benefit of feature extraction is to *normalise* the input data to a fixed-size representation. Many ML algorithms only work with fixed-size inputs; for example, the popular *backpropagation* [13] algorithm works on models with *fixed* topology (e.g. *artificial neural networks* with fixed connections between nodes). This requires some form of pre-processing in domains where the size of each input is not known, may vary or may even be unbounded.

For example, in the case of *online* learning we must make predictions/decisions before seeing all of the inputs. Unbounded input appears in domains such as programming and theorem proving, where individual term may be trees of unbounded depth. In these situations we need a mapping from arbitrary inputs to a fixed representation which is amenable to learning.

As an example, say we want to learn relationships between the following program fragments:

```
data Maybe a = Nothing | Just a
```

```
data Either a b = Left a | Right b
```

We might hope our algorithm discovers relationships like:

- Both are valid Haskell code.
- Both describe datatypes.
- Both datatypes have two constructors.
- `Either` is a generalisation of `Maybe` (we can define `Maybe a = Either () a` and `Nothing = Left ()`).
- There is a symmetry in `Either`: `Either a b` is equivalent to `Either b a` if we swap occurrences of `Left` and `Right`.
- It is trivial to satisfy `Maybe` (using `Nothing`).
- It is not trivial to satisfy `Either`; we require an `a` or a `b`.

However, this is too optimistic. Without our domain-knowledge of Haskell, an ML algorithm cannot impose any structure on these fragments, and will treat them as strings of bits. Our high-level hopes are obscured by low-level details: the desirable patterns of Haskell types are mixed with undesirable patterns of ASCII bytes, of letter frequency in English words, and so on.

In theory we could throw more computing resources and data at a problem, but available hardware and corpora are always limited. Instead, feature extraction lets us narrow the ML problem to what we, with our domain knowledge, consider important.

There is no *fundamental* difference between raw representations and features: the identity function is a valid feature extractor. Likewise, there is no crisp distinction between feature extraction and machine learning: a sufficiently-powerful learner doesn't require feature extraction, and a sufficiently-powerful feature extractor doesn't require any learning!¹

Rather, the terms are distinguished for purely *practical* reasons: by separating feature extraction from learning, we can distinguish straightforward, fast data transformation (feature extraction) from complex, slow statistical analysis (learning). This allows for modularity, separation of concerns, and in particular allows “off-the-shelf” ML to be re-used across a variety of different domains.

Even if we have no domain knowledge, we can still use a feature extraction phase to improve efficiency: first we learn a compact representation for our data, for example using *autoencoding*; then we use that encoder as a feature extractor for our main learning task. This stacking of one learning algorithm on top of another, especially with greedy learning of each layer, has led to the recent trend of *deep learning*.

Truncation and Padding

The simplest way to limit the size of our inputs is to truncate anything larger than a particular size (and pad anything smaller). This is the approach taken by ML4PG [6], which limits itself to trees with at most 10 levels and 10 elements per level; each tree is converted to a 30×10 matrix (3 values per tree node) and learning takes place on these normalised representations.

Truncation is unsatisfactory in the way it balances *data* efficiency with *time* efficiency. Specifically, truncation works best when the input data contains no redundancy and is arranged with the most significant data first (in a sense, it is “big-endian”). The less these assumptions hold, the less we can truncate. Since many ML algorithms scale poorly with input size, we would prefer to eliminate the redundancy using a more aggressive algorithm, to keep the resulting feature size as low as possible.

Dimension Reduction

A more sophisticated approach to the problem of reducing input size is to view it as a *dimension reduction* technique: our inputs can be modelled as points in high-dimensional spaces, which we want to project into a lower-dimensional space.

Truncation is a trivial dimension reduction technique: take the first N coordinates. More sophisticated projection functions consider the *distribution* of the points, and project with the hyperplane which preserves as much of the variance as possible (or, equivalently, reduces the *mutual information* between the points).

¹Consider a classification problem, to assign a label $l \in L$ to each input. If we only extract a single feature $f \in L$, we have solved the classification problem without using a separate learning step.

There are many techniques to find these hyperplanes, such as *principle component analysis* (PCA) and *autoencoding*. If the data are labelled, these classes can also be taken into account during dimension reduction [10]. However, since these techniques are effectively ML algorithms in their own right, they suffer some of the same constraints we’re trying to avoid:

- They operate *offline*, requiring all input points up-front
- All input points must have the same dimensionality

In particular, the second constraint is precisely what we’re trying to avoid. Sophisticated dimension reduction is still useful for *compressing* large, redundant features into smaller, information-dense representations, and as such provides a good complement to truncation.

The requirement for offline “batch” processing is more difficult to overcome, since any learning we perform for feature extraction will interfere with the core learning algorithm that’s consuming these features (this is why deep learning is often done greedily).

Sequences

To handle input of *variable* size, research attention has been given to the handling of *sequences*. This is a lossless approach, which splits the input into fixed-size *chunks* (e.g. splitting text into a sequence of characters), which are fed into an appropriate ML algorithm one at a time. The sequence is terminated by a sentinel; an “end-of-sequence” marker which, by construction, is distinguishable from the data chunks. This technique allows us to trade *space* (the size of our input) for *time* (the number of chunks in a sequence).

Not all ML algorithms can be adapted to accept sequences. One notable approach is to use *recurrent ANNs* (RANNs), which allow arbitrary connections between nodes, including cycles. Compared to *feed-forward* ANNs (FFANNs), which are acyclic, the *future output* of a RANN may depend arbitrarily on its *past inputs* (in fact, RANNs are universal computers).

The main problem with RANNs, compared to the more widely-used FFANNs, is the difficulty of training them. If we extend the standard backpropagation algorithm to handle cycles, we get the *backpropagation through time* algorithm [14]. However, this suffers a problem known as the *vanishing gradient*: error values decay exponentially as they propagate back through the cycles, which prevents effective learning of delayed dependencies, undermining the main advantage of RANNs. The vanishing gradient problem is the subject of current research, with countermeasures including *neuroevolution* (using evolutionary computation techniques to train an ANN) and *long short-term memory* (LSTM; introducing a few special, untrainable nodes to persist values for long time periods [7]).

An alternative to introducing recurrent connections, is the use of *attention*. In a (typically feed-forward) ANN with attention, the whole input is provided to

the network, which produces some output as well as some “attention” information; the network is run again, but with the input adjusted using this attention information (increasing and decreasing some of the input values); more output is produced, along with new attention information, and so on. Such networks do not need to “remember” long-distance relationships, whether by recurrent connections or gated nodes; they instead use the same input over and over again, but use their “attention” to emphasise those parts which should be relevant to future iterations.

Recursive Structure

Recursive structures, like trees and lists, have *fractal* dimension: adding layers to a recursive structure gives us more *fine-grained* features, rather than *orthogonal* features. For data mining context-free languages (e.g. those of programming and theorem-proving systems), we will mainly be concerned with tree and graph structures of variable size.

Using sequences to represent recursive structures is also problematic: if we want our learning algorithm to exploit structure (such as the depth of a token), it will have to discover how to parse the sequences for itself, which seems wasteful. The *back-propagation through structure* approach [4] is a more direct solution to this problem, using a feed-forward NN to learn recursive distributed representations [12] which correspond to the recursive structure of the inputs. Such distributed representations can also be used for sequences, which we can use to encode sub-trees when the branching factor of nodes is not uniform [9]. More recent work has investigated storing recursive structures inside LSTM cells [16].

A simpler alternative for generating recursive distributed representations is to use circular convolution [11]. Although promising results are shown for its use in *distributed tree kernels* [15], our preliminary experiments in applying circular convolution to functional programming expressions found most of the information to be lost in the process; presumably as the expressions are too small.

Kernel methods have also been applied to structured information, for example in [3] the input data (including sequences, trees and graphs) are represented using *generative models*, such as hidden Markov models, of a fixed size suitable for learning. Many more applications of kernel methods to structured domains are given in [1], which could be used to learn more subtle relations between expressions than recurrent clustering alone.

Bibliography

- [1] Gökhan Bakir. *Predicting structured data*. MIT press, 2007.
- [2] Scott Garrabrant, Tsvi Benson-Tilsen, Andrew Critch, Nate Soares, and Jessica Taylor. Logical induction. *arXiv preprint arXiv:1609.03543*, 2016.
- [3] Thomas Gärtner. A survey of kernels for structured data. *ACM SIGKDD Explorations Newsletter*, 5(1):49–58, 2003.
- [4] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.
- [5] CARL G. HEMPEL. I.—STUDIES IN THE LOGIC OF CONFIRMATION (II.). *Mind*, LIV(214):97–121, 04 1945.
- [6] Jónathan Heras and Ekaterina Komendantskaya. ML4PG: proof-mining in Coq. *CoRR*, abs/1302.6421, 2013.
- [7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
- [8] Marcus Hutter, John W. Lloyd, Kee Siong Ng, and William T. B. Uther. Probabilities on Sentences in an Expressive Logic. *Journal of Applied Logic*, 11(4):386–420, December 2013.
- [9] Stan C Kwasny and Barry L Kalman. Tail-recursive distributed representations and simple recurrent networks. *Connection Science*, 7(1):61–80, 1995.
- [10] F. Oveisi, S. Oveisi, A. Erfanian, and I. Patras. Tree-Structured Feature Extraction Using Mutual Information. *IEEE Transactions on Neural Networks and Learning Systems*, 23(1):127–137, January 2012.
- [11] Tony Plate. Holographic Reduced Representations: Convolution Algebra for Compositional Distributed Representations. In John Mylopoulos and Raymond Reiter, editors, *IJCAI*, pages 30–35. Morgan Kaufmann, 1991.

- [12] Jordan B Pollack. Recursive distributed representations. *Artificial Intelligence*, 46(1):77–105, 1990.
- [13] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.
- [14] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
- [15] Fabio Massimo Zanzotto and Lorenzo Dell’Arciprete. Distributed tree kernels. *arXiv preprint arXiv:1206.4607*, 2012.
- [16] Xiaodan Zhu, Parinaz Sobihani, and Hongyu Guo. Long short-term memory over recursive structures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1604–1612, 2015.