

The Knowledge Graph as the Default Data Model for Machine Learning

Xander Wilcke^{a,b,*} Peter Bloem^a and Victor de Boer^a

^a Faculty of Sciences, Vrije Universiteit Amsterdam

^b Faculty of Spatial Economics, Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Abstract. In modern machine learning, *raw data* is the preferred input for our models. Where a decade ago data scientists were still engineering features, manually picking out the details they thought salient, they now prefer the data in their raw form. As long as we can assume that all relevant and irrelevant information is present in the input data, we can design deep models that build up intermediate representations to sift out relevant features. However, these models are often domain specific and tailored to the task at hand, and therefore unsuited for learning on *heterogeneous knowledge*: information of different types and from different domains. If we can develop methods that operate on this form of knowledge, we can dispense with a great deal of ad-hoc feature engineering and train deep models end-to-end in many more domains. To accomplish this, we first need a data model capable of expressing *heterogeneous knowledge* naturally in various domains, in as usable a form as possible, and satisfying as many use cases as possible. In this position paper, we argue that the *knowledge graph* is a suitable candidate for this data model. This paper describes current research and discusses some of the promises and challenges of this approach.

Keywords: Knowledge Graphs, Semantic Web, Machine Learning, End-to-End Learning, Position paper

1. Introduction

In the last decade, the methodology of Data Science has changed radically. Where machine learning practitioners and statisticians would generally spend most of their time on extracting meaningful features from their data, often creating a derivative of the original data in the process, they now prefer to feed their models the data in their raw form. Specifically, data which still

contains all relevant and irrelevant information rather than having been reduced to features selected or engineered by data scientists. This shift can largely be attributed to the emergence of *deep learning*, which showed that we can build layered models of intermediate representations to sift out relevant features, and which allows us to dispense with manual *feature engineering*.

For example, in the domain of image analysis, popular feature extractors like SIFT [Low99] have given way to Convolutional Neural Networks [LDH⁺90, KSH12], which naturally consume raw images. These are used for instance, in facial recognition models building up layers of intermediate representations: from low level features built on the raw pixels like local edge detectors, to higher level features like specialized detectors for eyes, noses, up to the face of a specific person [Le13]. Similarly, in audio analysis, it is common to use models that consume audio data directly [GMH13] and in Natural Language Processing it is possible to achieve state-of-the-art performance without explicit preprocessing steps such as POS-tagging and parsing [NG15].

This is one of the strongest benefits of deep learning: we can directly feed the model the dataset as a whole, containing all relevant and irrelevant information, and trust the model to unpack it, to sift through it, and to construct whatever low-level and high-level features are relevant for the task at hand. Not only do we not need to choose what features might be relevant to the learning task—making ad-hoc decisions and removing, adding, and reshaping information in the process—we can let the model surprise us: it may find features in our data that we would never have thought of ourselves. With feature engineering now being part of the model itself, it becomes possible to learn

*Corresponding Author, E-mail: w.x.wilcke@vu.nl.

directly from the data. This is called end-to-end learning (we further explain this in the text box below).

However, all present end-to-end learning methods are domain-specific: they are tailored to images, to sound, or to language. When faced with *heterogeneous knowledge*—information of different types and from different domains—we often find ourselves resorting back to manual feature engineering. To avoid this, we require a machine learning model capable of directly consuming heterogeneous knowledge, and a data model suitable of expressing such knowledge naturally and with minimal loss of information. In this paper, we argue that the knowledge graph is a suitable data model for this purpose and that, in order to achieve end-to-end learning on heterogeneous knowledge, we should a) adopt the knowledge graph as the default data model for this kind of knowledge and b) develop end-to-end models that can directly consume these knowledge graphs.

Concretely, we will use the term *heterogeneous knowledge* to refer to: entities (things), their relations, and their attributes. For instance, in a company database, we may find entities such as employees, departments, resources and clients. Relations express which employees work together, which department each employee works for and so on. Attributes can be simple strings, such as names and social security numbers, but also richer media like short biographies, photographs, and promotional videos.

We will first explain the principle behind the knowledge graph model with the help of several practical examples, followed by a discussion on the potential of knowledge graphs for end-to-end learning and on the challenges of this approach. We will finish with a concise overview of promising current research in this area.

End-to-End Learning

Why is end-to-end learning now so important to data scientists? Is this just a modern affectation? Paul Mineiro provides a good reason to consider this a more fundamental practice [Min]. In most areas of software engineering, solving a complex problem begins with breaking the problem up into subproblems: divide and conquer. Each subproblem is then solved in one module, and the modules are chained together to produce the required end result. If, however, these modules use machine learning, we have to take into account that their answers are necessarily *inexact*.

“Unfortunately, in machine learning we never exactly solve a problem. At best, we approximately solve a problem. This is where the technique needs modification: in software engineering the subproblem solutions are exact, but in machine learning errors compound and the aggregate result can be complete rubbish. In addition apparently paradoxical situations can arise where a component is “improved” in isolation yet aggregate system performance degrades when this “improvement” is deployed (e.g., due to the pattern of errors now being unexpected by downstream components, even if they are less frequent).

Does this mean we are doomed to think holistically (which doesn’t sound scalable to large problems)? No, but it means you have to be defensive about subproblem decomposition. The best strategy, when feasible, is to train the system end-to-end, i.e., optimize all components (and the composition strategy) together rather than in isolation.”

- Paul Mineiro, 15-02-2017 [Min]

Even if we are forced to pre-train each component in isolation, it is crucial to follow that pre-training up with a complete end-to-end training step when all the modules are composed [Bot]. This puts a very strong constraint on the kind of modules that we can use: an error signal needs to be able to propagate through all layers of the architecture, from the output back to the original data that inspired it.

Any pre-processing done on the data, any manual feature extraction, harmonization and or scaling can be seen as a module in the pipeline that cannot be tweaked, and does not allow a final optimization end-to-end. Any error introduced by such modules can never be retrieved. Since these are often modules at the start of our pipeline, even the smallest mistake or suboptimal choice can be blown up exponentially as we add layers to the model.

1.1. Use cases

Throughout the paper, we will use three different use cases as running examples:

Spam detection is one of the first classification problems to be solved well enough to be widely implemented in commercial products. Early approaches tackled this task by converting email text to term vectors, and using these term vectors in a naive Bayes classifier.

Movie recommendation is a standard use case for recommender systems. Here, we have a set of users, and a set of movies. Some users have given ratings to some movies. In one early successful model, ratings are written as a matrix, which is then decomposed into factors that are multiplied back again to produce new ratings from which recommendations are derived.

Market basket analysis is one of earliest success stories which helped retailers understand customer purchasing behaviour, and which allowed them to adjust their strategy accordingly. The breakthrough that allowed this came with association rule mining, which converts all transactions into vectors and then computes their inner and outer correlations.

2. The knowledge graph

The aforementioned use cases share several common aspects: in each case we have a set of instances, and we have a collection of diverse and heterogeneous facts representing our knowledge about these instances. Some facts link instances together (John is a friend of Mary, John likes Jurassic Park) and some describe attributes of instances (Jurassic Park was released on June 9, 1993).

The question of how to represent such knowledge is not a new one. It has been studied by AI researchers since the invention of the field, and before [DSS93]. The most recent large-scale endeavour in this area is undoubtedly the *Semantic Web*, where knowledge is encoded in knowledge graphs.

The knowledge graph data model used in the Semantic Web is based on three basic principles:

1. Encode knowledge using statements.
2. Express background knowledge in ontologies.
3. Reuse knowledge between datasets.

We will briefly discuss each of these next.

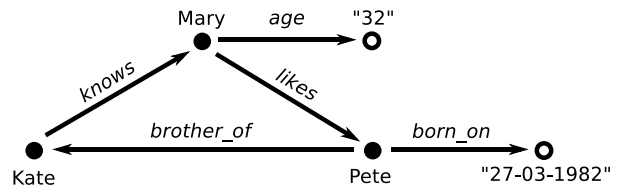


Figure 1. Graphical representation of the example given in §2. Edges represent binary relations. Vertices' shapes reflect their roles: solid circles represent entities, empty circles represent their attributes.

2.1. Encode knowledge using statements

The most fundamental idea behind the Semantic Web is that knowledge should be expressed using *statements*. Consider the following example:

```

Kate knows Mary.
Mary likes Pete.
Mary age "32".
Pete brother_of Kate.
Pete born_on "27-03-1982".
  
```

All of the above are statements that comply with the Resource Description Framework (RDF), a data model which forms the basic building block of the Semantic Web.¹ This model specifies that each statement should consist of a single binary *property* (the verb) which relates two *resources* (the subject and object) in a left-to-right order. Together, these three are referred to as an RDF *triple*. We can also represent this example as a directed graph as shown in Figure 1.

Resources can be either entities (things) or literals which hold values such as text, numbers or dates. Triples can either express relations between entities when the resources on both sides are things, or they can express attributes when the resource on the right-hand side is a literal. For instance, the last line of our example set expresses an attribute of Pete (date of birth) with value "27-03-1982".

Apart from the few rules already listed, the RDF data model itself does not impose any further restrictions on how knowledge engineers should model their knowledge: we could have modelled our example differently, for instance by representing dates as resources. In general, such modelling choices depend on the domain and on the intended purposes of the dataset.

2.2. Express background knowledge in ontologies

Where the RDF data model gives free rein over modelling choices, ontologies offer a way to express how

¹<https://www.w3.org/RDF/>

knowledge is structured in a given domain and by a given community. For this purpose, ontologies contain classes (entity types) and properties that describe the domain, as well as constraints and inferences on these classes and properties. For instance, an ontology might define `Person` as the class containing all individual persons. It might likewise define `type` as the property that assigns an entity to a class. As an example, let us use these to extend our example set with the following statements:

```
Kate type Person.
Mary type Person.
Pete type Person.
```

Kate, Mary, and Pete are now all said to be instances of the class `Person`. This class may hold various properties, such as that it is equivalent to the class `Human`, disjoint with the class `Animal`, and that it is a subclass of class `Agent`. This last property is an example of a recursive property, and can be expressed using the RDFS—RDF Schema—ontology² which extends the bare RDF model with several practical classes and properties. The other two relations are more complex, and require a more expressive ontology to be stated. OWL, the Web Ontology Language, is generally the preferred choice for this purpose.³

Ontologies can be used to derive *implicit knowledge*. For instance, knowing that Kate is of the type `Person`, and that `Person` is itself a subclass of `Agent`, allows a reasoning engine to derive that Kate is an `Agent` as well. We will return to this topic in Section 4.2.

2.3. Reuse knowledge between datasets

Re-using knowledge can be done by referring to resources not by name, but by a unique identifier. On the Semantic Web, these identifiers are called *Internationalized Resource Identifiers*, or IRIs, and generally take the form of a web address. For instance, we can use the IRIs <http://vu.nl/staff/KateBishop> and <http://vu.nl/staff/MaryWatson> to refer to Kate and Mary, respectively. More often, we would write these IRIs as `vu:KateBishop` and `vu:MaryWatson`, with `vu:` as shorthand for the <http://vu.nl/staff/> namespace. We can now rewrite the first statement of our example set as

```
vu:KateBishop knows vu:MaryWatson .
```

²<https://www.w3.org/TR/rdf-schema/>

³<https://www.w3.org/OWL/>

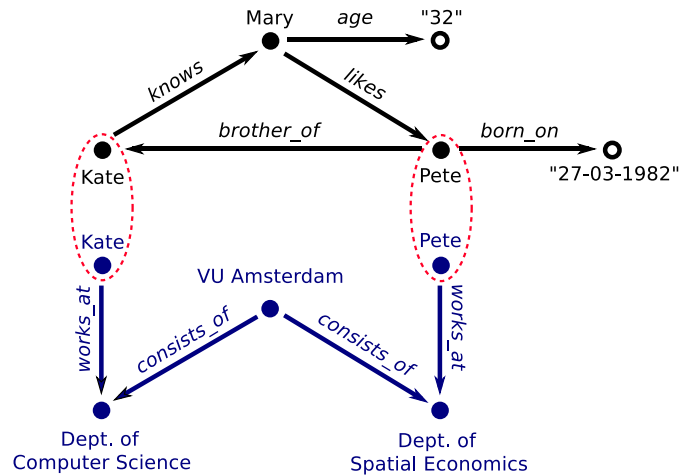


Figure 2. Extension of the original example (Fig. 1) with a dataset on VU employees. Resources `Kate` and `Pete` occur in both graphs and can therefore be used to link the datasets together.

This statement implies the same as before, but now we can safely add other people also named Kate or Mary without having to worry about clashes. Of course, we can do the same for our properties. To spice things up, let us assume that we used an already existing ontology, say the widely used *FOAF* (Friend Of A Friend) ontology.⁴ This lets us write the statement as

```
vu:KateBishop foaf:knows vu:MaryWatson .
```

We now have a triple that is fully compliant with the RDF data model, and which uses knowledge from a shared and common ontology.

The principle of reusing knowledge is a simple idea with several consequences, most particular with respect to integrating, dereferencing, and disambiguating knowledge:

Integrated knowledge

Integrating datasets is as simple as linking two knowledge graphs at equivalent resources. If such a resource holds the same IRI in both datasets an implicit coupling already exists and no further action is required. In practice, this boils down to simply concatenating one set of statements to another. For instance, we can extend our example set with another dataset on VU employees as long as that dataset contains any of the three resources: Kate, Mary, or Pete (Fig. 2). Of course, integration on the data level does not mean that

⁴<https://xmlns.com/foaf/spec/>

the knowledge itself is neatly integrated as well: different knowledge graphs can be the result of different modelling decisions. These will persist after integration. We will return to this topic in more detail in Section 4.4.

Dereferenceable knowledge

An IRI is more than just an identifier: it can also be a web address pointing to the location where a resource or property is described. For these data points, we can retrieve the description using standard HTTP. This is called dereferencing, and allows for an intuitive way to access external knowledge. In practice, not all IRIs are dereferenceable, but many are.

Disambiguated knowledge

Dereferencing IRIs allows us to directly and unambiguously retrieve relevant information about entities in a knowledge graph, amongst which are classes and properties in embedded ontologies. Commonly included information encompasses type specifications, descriptions, and various constraints. For instance, dereferencing `foaf:knows` tells us it is a property used to specify that a certain person knows another person, and that we can infer that resources that are linked through this property are of type `Person`.

We have recently seen uptake of these principles on a grand scale, with the Linked Open Data (LOD) cloud as prime example. With more than 38 billion statements from over 1100 datasets (Fig. 3), the LOD cloud constitutes a vast distributed knowledge graph which encompasses almost any domain imaginable. As such, it comes close to the original dream of a Semantic Web.

With this wealth of data available, we now face the challenge of designing machine learning models capable of learning in a world of knowledge graphs.

3. Learning in a world of knowledge graphs

We now revisit the three use cases described in the introduction and discuss how they can benefit from the use of knowledge graphs as default data model, and how this leads to a suitable climate for end-to-end learning by removing the need for manual feature engineering.

3.1. Spam detection

Before, we discussed how early spam detection methods classified e-mails based solely on the content of

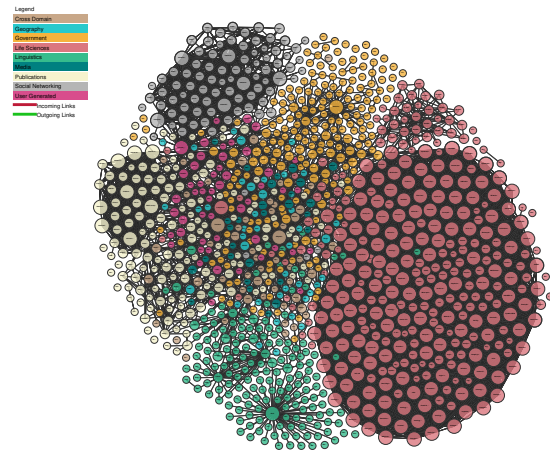


Figure 3. A depiction of the LOD cloud, holding over 38 billion facts from more than 1100 linked datasets. Each vertex represents a separate dataset in the form of a knowledge graph. An edge between two datasets indicates that they share at least one IRI. Figure from [AMB⁺].

the message. We often have much more information at hand. We can distinguish between the body text, the subject heading, and the quoted text from previous emails. But we also have other attributes: the sender, the receiver, and everybody listed in the CC. We know the IP address of the SMTP server used to send the email, which can be easily linked to a region. In a corporate setting, many users correspond to employees of our companies, for whom we know dates-of-birth, departments of the company, perhaps even portrait images or a short biography. All these aspects provide a wealth of information that can be used in the learning task.

In the traditional setting, the data scientist must decide how to translate all this knowledge into feature vectors, so that machine learning models can learn from it. This translation has to be done by hand and the data scientist in question will have to make a judgement in each case whether the added feature is worth the effort. Instead, it would be far more convenient and effective if we can train a suitable end-to-end model directly on the dataset as a whole, and let *it* learn the most important features itself. We can achieve this by expressing this dataset in a knowledge graph.

An example of how such a knowledge graph might look is depicted in Figure 4. Here, information about who sent the e-mails, who received them, which e-mails are replies, and which SMTP servers were used are combined in a single graph. The task is now to la-

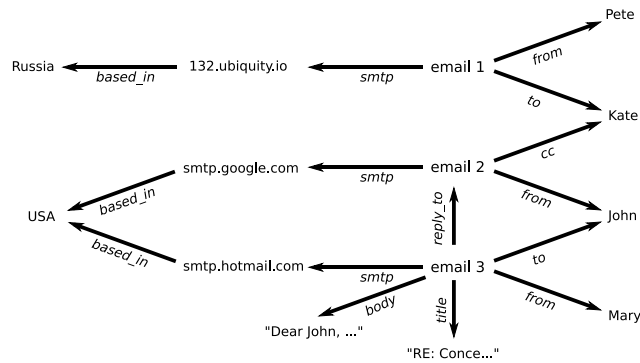


Figure 4. An example dataset on email conversations used in the use case on spam detection of §3.1.

bel the vertices that represent emails as spam or not spam—a straightforward entity classification task.

3.2. Movie Recommendation

In traditional recommender systems, movie recommendations are generated by constructing a matrix of movies, people and received ratings. This approach assumes that people are likely to enjoy the same movies as people with a similar taste, and therefore needs existing ratings for effective recommendation [KBV09]. Unfortunately, we do not always have actual ratings yet and are thus unable to start these computations. This is a common issue in the traditional setting, called the *cold-start problem*.

We can circumvent this problem by relying on additional information to make our initial predictions. For instance, we can include the principal actors, the director, the genre, the country of origin, the year it was made, whether it was adapted from a book, et cetera. Including this knowledge solves the cold start problem because we can link movies and users for which no ratings are yet available to similar entities through this background data.

An example of a knowledge graph about movies is depicted in Figure 5. The dataset featured there consists of two integrated knowledge graphs: one about movies in general, and another containing movie ratings provided by users. Both graphs refer to movies by the same IRIs, and can thus be linked together via those resources. We can now recast the recommendation task as *link prediction*, specifically the prediction of the property `likes` that binds users to movies. Background knowledge and existing ratings can both be used, as their availability allows. For instance, while the movie `Indiana_Jones` has no ratings, we do know that it is of the same genre and

from the same director as `Jurassic_Park`. Any user who likes `Jurassic_Park` might therefore also like `Indiana_Jones`.

3.3. Market Basket Analysis

Before, we mentioned how retailers originally used transactional information to map customer purchase behaviour. Of course, we can include much more information than only anonymous transactions. For instance, we can take into account the current discount on items, whether they are healthy, and where they are placed in the store. Consumers are already providing retailers with large amounts of personal information as well: age, address, and even indirectly information about their marital and financial status. All these attributes can contribute to a precise profile of our customers.

Limiting the data purely to items imposes an upper bar on the complexity of the patterns our methods can discover. However, by integrating additional knowledge on products, ingredients, and ecological reports, our algorithms can discover more complex patterns. They might, for example, find that Certified Humane⁵ products are often bought together, that people who buy these products also buy those which are eco-friendly, or that products with a low nutritional value are more often bought on sunny days.

An example of how a knowledge graph on transactions might look is shown in Figure 6. Each transaction is linked to the items that were bought at that moment. For instance, all three transactions involve buying drumsticks. This product consist of chicken, which we know due to the coupling of the knowledge graph on transactions with that of product information. We

⁵<http://certifiedhumane.org/>

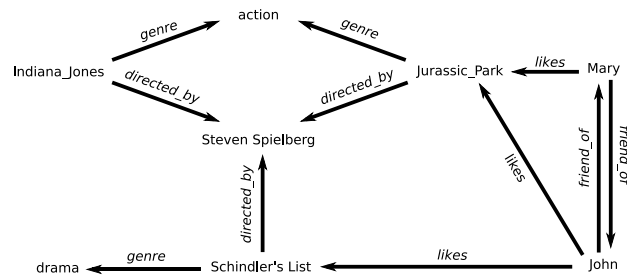


Figure 5. An example dataset on movies and ratings used in the use case on movie recommendations of §3.2.

further extended this by integrating external datasets about suppliers and ecological reports.

3.4. The default data model?

All three use cases benefited from the use of knowledge graphs to model heterogeneous knowledge, as opposed to the current *de facto* default: the table. There are however, more data models capable of expressing heterogeneous knowledge natively. This raises the question whether the same can also be accomplished by modelling our knowledge in some other data model.

Let us consider two popular alternatives: the relational model (for database management) and XML.

The tree structure of XML is a limiting factor compared to knowledge graphs. Any graph structure we want to store in XML data must be encoded using links that are not native to the data model. If, for instance, we want to store a social network in an XML format, say with a single node for each person, the relations between these people must be encoded by links between these nodes that are not native to the data model. A learning model designed to consume XML would exploit the tree structure, but not the ad-hoc graph structure between these nodes.

The differences between the relational model and the knowledge graph are more subtle. Indeed, there are often very seamless translations between the two. Nevertheless, there are some differences, mostly based on the way these models are currently used (rather than their intrinsic properties), that make knowledge graphs a more practical candidate for end-to-end learning on heterogeneous knowledge.

One important difference is how both data models allow data integration: where it is a simple task to integrate two knowledge graphs at the data level—we only need one IRI shared by both—this is a considerable problem with relational databases and typically requires various complex table operations [HS95]. While data integration by matching IRIs is certainly no silver

bullet (as discussed further in Section 4.4), it does allow a very simple first step to integration. This is important in the context of end-to-end learning, because it makes it possible, in principle, to let the model *learn* the rest of the data integration. While this is an ambitious goal, knowledge graphs in principle allow us to link two datasets in the simplest possible manner, and let the model learn the harmonization between the two, insofar as it is relevant to the learning task.⁶

Another difference is simply the availability of data: relational databases are typically designed for a specific purpose and often operate as solitary units in an enclosed environment. Data hosted as such is usually in some proprietary format and difficult to retrieve as a single file, in a standardized open format. Knowledge graphs widely published, and a mature stack of open standards is available to encode them.

Of course, there are domains in which the knowledge graph is a poor choice of data model. It would be highly impractical to convert every pixel of every image in an image database to a node in a knowledge graph. However, it *is* feasible to represent images as *literals* in a knowledge graph. For a given set of images, this allows us to present the raw pixel data, together with its metadata in a unified format. This creates a spectrum of datasets, where at one end, the relevant information is primarily encoded in the literals and at the other, the relevant data is primarily encoded in the graph structure itself.

4. The challenges ahead

In the previous section, we argued that expressing heterogeneous knowledge in knowledge graphs holds

⁶A similar effect *could* be achieved for relational databases if IRIs (or some other universal naming scheme) were adopted to create keys between databases, but we are not aware of any practical efforts to that effect.

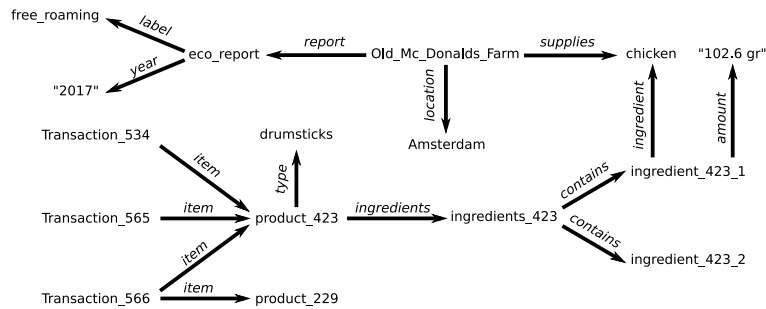


Figure 6. An example dataset on transactions, their items, and additional information used in the use case on market basket analysis of §3.3.

great promise. We assumed in each case that effective end-to-end learning models are available. However, to develop such models some key challenges need to be addressed, specifically on how to deal with incomplete knowledge, implicit knowledge, heterogeneous knowledge, and differently-modelled knowledge. We will briefly discuss each of these problems next.

4.1. Incomplete knowledge

Knowledge graphs are inherently forgiving towards missing values: rather than to force knowledge engineers to fill in the blanks with artificial replacements—NONE, NULL, -1, 99999, et cetera—missing knowledge is simply omitted altogether.

When dealing with real-world knowledge, we are often faced with large amounts of missing values: for many properties in such a dataset, there may be more entities for which the value is missing, than for which it is known.

While the occasional missing value can be dealt with accurately enough using current imputation methods, guessing values attributes for the majority of instances from a small sample of provides values is problematic. Ideally, models for knowledge graphs will instead simply use the information that is present, and ignore the information that is not, dealing with the uneven distribution of information among entities natively.

4.2. Implicit knowledge

Knowledge graphs contain a wealth of implicit knowledge, implied through the interplay of assertion knowledge and background knowledge. Consider class inheritance: for any instance of class C_1 holds that, if C_1 is a subclass of C_2 , then it is also an instance of class C_2 . Here, additional knowledge is derived by exploiting the property’s transitivity.

In the case of end-to-end learning, the ability to exploit implicit knowledge should ideally be part of the model itself. Already, studies have shown that machine learning models are capable of approximating deductive reasoning with background knowledge [PS16]. If we can incorporate such methods into end-to-end models, it becomes possible to let these models learn the most appropriate level of inference themselves.

4.3. Heterogeneous knowledge

Recall that literals allow us to state explicit values—texts, numbers, dates, IDs, et cetera—as direct attributes of resources. This means that literals contain their own values, which contrasts with non-literal resources for which their local neighbourhood—their context—is the ‘value’. Simply treating literals the same as non-literal resources will therefore be ineffective. Concretely, this would imply that literals and non-literals can be compared using the same distance metric. However, any comparison between explicit values and contexts is unlikely to yield sensible results. Instead, we must treat literals and non-literals as separate cases. Moreover, we must also deal with each different data type separately and accordingly: texts as strings, numbers and dates as ordinal values, IDs as nominal values, et cetera.

For instance, in our spam detection example, both the e-mails’ title and body were modelled as string literals. The simplest solution would be to simply ignore these attributes and to focus solely on non-literal resources, but doing so comes at the cost of losing potentially useful knowledge. Instead, we can also design our models with the ability to compare strings using some string similarity metric, or represent them using a learned embedding. That way, rather than perceiving the title “Just saying hello” as totally different from “RE: Just saying hello”, our models would discover that these two titles are actually very similar.

4.4. Differently-modelled knowledge

Different knowledge engineers represent their knowledge in different ways. The choices they make are reflected in the topology of the knowledge graphs they produce: some knowledge graphs have a relatively simple structure while others are fairly complex, some require one step to link properties while others use three, some strictly define their constraints while others are lenient.

Recall how easy it is to integrate two knowledge graphs: as long as they share at least one IRI, an implicit integration already exists. Of course, this integration only affects the data layer: the combined knowledge expressed by these data remains unchanged. This means that differences in modelling decisions remain present in the resulting knowledge graph after integration. This can lead to an internally heterogeneous knowledge graph.

As a concrete example, consider once more the use case of movie recommendation (Fig 5). To model the ratings given by users, we linked users to movies using a single property: *X likes Y*. We can also model the same relation using an intermediate vertex—a movie rating—and let *it* link both to the movie which was rated and to the literal which holds the actual rating itself:

```
Mary has_rating Mary_Rating_7.
Mary_Rating_7 rates Jurassic_Park.
Mary_Rating_7 has_value "1.0".
Mary_Rating_7 timestamp "080517T124559".
```

Dealing with knowledge modelled in different ways remains a challenge for effective machine learning. Successful end-to-end models need to take this topological variance into account so they can recognize that similar information is expressed in different ways.

5. Current approaches

Recent years witnessed a growing interest in the knowledge graph by the machine learning community. Initial explorations focused primarily on how entire knowledge graphs can be ‘flattened’ into plain tables—a process known as *propositionalization*—for use with traditional learning methods, whereas more recent studies are looking for more natural ways to process knowledge graphs. This has led to various methods which can be split into two different approaches:

1) those which extract feature vectors from the graph for use as input to traditional models, and 2) those which create an internal representation of the knowledge graph itself.

5.1. Extracting feature vectors

Rather than trying to learn directly over knowledge graphs, we could also first translate them into a more-manageable form for which we already have many methods available. Specifically, we can try to find feature vectors for each vertex in the graph that represents an instance—an *instance vertex*—in our training data. We will briefly discuss two prominent examples that use this approach: substructure counting and RDF2Vec. Clearly, these methods fall short of the ideal of end-to-end learning, but they do provide a source of inspiration for how to manage the challenges posed in the previous chapter.

5.1.1. Substructure counting

Substructure counting graph kernels [dVdR15], are a family of algorithms that generate feature vectors for instance vertices by counting various kinds of substructure that occur in the direct neighborhood of the instance vertex. While these methods are often referred to as *kernels*, they can be used equally well to generate explicit feature vectors, so we will view them as feature extraction methods here.

The simplest form of substructure counting method takes the *k*-neighborhood around an instance vertex, and simply counts each *label*: that is, each edge label, and each vertex label. Each label encountered in the neighborhood of an instance vertex then becomes a feature, with its frequency as the value. For instance, for each e-mail in our example dataset (Fig. 4), the feature space consists of at least one sender (e.g., `from_Mary: 1`), one main recipient (e.g., `to_John: 1`), and zero or more other recipients (e.g., `cc_Pete: 0` and `bcc_Kate: 0`).

More complex kernels define the neighborhood around the instance vertex differently (as a tree, for instance) and vary the structures that are counted to form the features (for instance, paths or trees). The Weisfeiler-Lehman (WL) graph kernel [SSL⁺11] is a specific case, and the WL algorithm is the key to efficiently computing feature vectors for many substructure-counting graph methods.

5.1.2. RDF2Vec

The drawback of substructure-counting methods is that the size of the feature vector grows with the size of the

data. *RDF2Vec* [RP16] is a method which generates feature vectors of a *given* size, and does so efficiently, even for large graphs. This means that, in principle, even when faced with a machine learning problem on the scale of the web, we can reduce the problem to a set of feature vectors of, say, 500 dimensions, after which we can solve the problem locally on commodity hardware.

RDF2Vec is a relational version of the idea behind *DeepWalk* [PARS14], an algorithm that finds embeddings for the vertices of unlabeled graphs. The principle is simple: extract short random walks starting at the instance vertices, and feed these as sentences to the *Word2Vec* [MCCD13] algorithm. This means that a vertex is modeled by its context and a vertex’s context is defined by the vertices up to d steps away. For instance, in our example dataset on customer transactions (Fig. 6), a context of depth 3 allows *RDF2Vec* to represent each transaction via chains such as

```
transaction_X → ingredients_X → ingredient_Y
transaction_X → ingredients_X → ingredient_Z
```

For large graphs, reasonable classification performance can be achieved with samples of a few as 500 random walks. Other methods for finding embeddings on the vertices of a knowledge graph are *TransE* [BUGD⁺13] and *ProjE* [SW16].

5.2. Internal graph representation

Both the WL-kernel and *RDF2Vec* are very effective ways to perform machine learning on relational knowledge, but they fall short of our goal of true end-to-end learning. While these methods consume heterogeneous knowledge in the form of RDF, they operate in a pipeline of discrete steps. If, for instance, they are used to perform classification, both methods first produce feature vectors for the instance vertices, and then proceed to use these feature vectors with a traditional classifier. Once the feature vectors are extracted, the error signal from the task can no longer be used to fine-tune the feature extraction. Any information lost in transforming the data to feature vectors is lost forever.

In a true end-to-end model, every step can be fine-tuned based on the learning task. To accomplish this, we need models capable of directly consuming knowledge graphs and which can hold internal representations of them. We next briefly discuss two prominent models that employ this approach: tensors and graph convolutional networks.

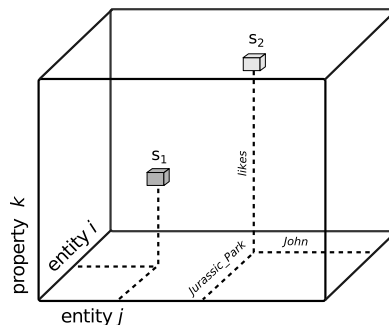


Figure 7. Representing statements as points in a third-order tensor. Two statements are illustrated: s_1 and s_2 , with $s_2 = \text{John likes Jurassic_Park}$

5.2.1. Tensor representation

A tensor is the generalization of a matrix into more than two dimensions, called orders. Given that knowledge graph statements consist of three elements, we can use a third-order tensor to map them: two orders for entities, and another order for properties. The intersection of all three orders, a point, will then represent a single statement. This principle is depicted in Figure 7. As an example, let i, j, k be the indices of a tensor \mathbf{T} used to represent our dataset on movie recommendations (Fig 5). If now $\mathbf{T}[i] = \text{John}$, $\mathbf{T}[j] = \text{Jurassic_Park}$, and $\mathbf{T}[k] = \text{likes}$, then intersection $\mathbf{T}[i, j, k]$ will constitute the statement *John likes Jurassic_Park*.

A tensor representation allows for all possible combinations of entities and properties, even those which are false. To reflect this, the value at each point holds the truth value of that statement: 1.0 if it holds, and 0.0 otherwise. In that sense, it is similar to an adjacency matrix.

To predict which unknown statements might also be true, we can apply tensor decomposition. Similar to matrix decomposition, this approach decomposes the tensor into multiple second-order tensors by which latent features emerge. These tensors are again multiplied to create an estimate of the original tensor. However, where before some of the points had 0.0 as value, they now have a value somewhere between 0.0 and 1.0.

This application of tensor decomposition was first introduced as a semantically-aware alternative [FSSS09] to authority ranking algorithms such as PageRank and HITS, but gained widespread popularity after being reintroduced as a distinct model for collective learning on knowledge graphs [NTK11]. Others have later integrated this tensor model as a layer in a regular [SCMN13] or recursive neural network [SPW⁺13].

5.2.2. Graph Convolutional Neural Networks

Graph Convolutional Networks (GCN) strike a balance between modeling the full structure of the graph dynamically, as the tensor model does, and modeling the local neighborhood structure through extracted features (as substructure counting methods and RDF2Vec do). The RGCN used the graph structure of the data to inform the topology of a neural network, allowing information to propagate along the edges of the graph with each layer of the neural network.

The Relational Graph Convolutional Network (RGCN) introduced in [SKB⁺17], and the related *column networks* [PTPV17] are relatively straightforward translation of the graph-convolutional network [KW16, BZSL13] to the domain of knowledge graphs. We will briefly explain the basic principle behind GCNs, to give the reader a basic intuition of the principle.

Assume that we have an undirected graph with N vertices, with a small feature vector x for each vertex. We can either use the natural features of the vertex in the data, or if the data does not label the vertices in any way, we can assign each vertex i a one-hot vector⁷ of length N . For this example, we will assume that each vertex is assigned a random and unique color, represented by a vector of length 3 (a point in the RGB color space).

Let x^0 be the color of vertex i . We define x^k as the mixture of the colors of all vertices in the graph, weighted by the probability that a length- k random walk from vertex i ends up in each vertex. If \mathbf{X}^0 is the N by 3 matrix containing all original vertex features we can define this principle mathematically as $\mathbf{X}^{k+1} = \mathbf{A}\mathbf{X}^k$, where \mathbf{A} is the normalized adjacency matrix of graph G . If we start with one-hot vectors instead of colors, x^k becomes a probability vector with x_j^k the probability that a random walk of k steps from vertex i ends up in vertex j .

For most graphs, x^k converges with k to a single vector independent of the starting vertex. This gives us a specific-to-generic sequence of representations for vertex i : x^0 is too specific, and x^k is too generic. Somewhere in-between, we have a good representation, expressing both similarities and differences.

The GCN model (Fig. 8) uses these ideas to create a differentiable map from one vector representation into another. We start with a matrix of one-hot vectors \mathbf{X} . These are multiplied by \mathbf{A} , and then translated down

⁷A vector u representing element i out of a set of N elements: u is 0 for all indices except for u_i , which is 1.

to a lower dimensional feature space by a matrix \mathbf{W} . \mathbf{W} represents the “weights” of the model; the elements that we will modify to fit the model to the data. The result is then transformed by a nonlinearity σ (commonly a linear rectifier) to give us our intermediate representations \mathbf{H} :

$$\mathbf{H} = f_{\mathbf{W}}^{\sigma}(\mathbf{X}) = \sigma(\mathbf{A}\mathbf{X}\mathbf{W}).$$

Row i of matrix \mathbf{H} now contains a feature vector of length 16, describing vertex i .

To create a classifier with M classes, we normally compose two such “layers”, giving the second a softmax⁸ restriction on the output vectors. This gives us a length- M probability vector y for each vertex, representing the classification.

Thus, the complete model becomes

$$\mathbf{Y} = f_{\mathbf{V}}^{\text{softmax}}(\mathbf{A}f_{\mathbf{W}}^{\sigma}(\mathbf{A}\mathbf{X}\mathbf{W})\mathbf{V}),$$

where \mathbf{X} is the identity matrix (i.e. a stack of one-hot vectors for each vertex), and \mathbf{Y} is an $N \times M$ matrix with \mathbf{Y}_{ij} the probability that vertex i has class j . We then learn the weights \mathbf{V} and \mathbf{W} by minimizing the cross-entropy between the training examples and the corresponding rows of \mathbf{Y}_{ij} through gradient descent.

For the RGCN model, we have one adjacency matrix per relation in the graph, one for its inverse of each relation, and one for self-connections. For a more complete description, see [SKB⁺17].

Like RDF2Vec, the RGCNs learn fixed-size intermediate representations of the vertices of the graph. Unlike RDF2Vec, the transformation to this representation can use the error signal from the next layer to tune its parameters. The price we pay is that these models are currently much less scalable than alternatives like RDF2Vec.

6. Conclusion

When faced with heterogeneous knowledge in a traditional machine learning context, data scientists craft feature vectors which can be used as input for learning algorithms. These transformations are performed by removing, and reshaping data, and can result in the loss of information and accuracy. To solve this problem, we require end-to-end models which can directly

⁸This ensures that the output values for a given node always sum to one.

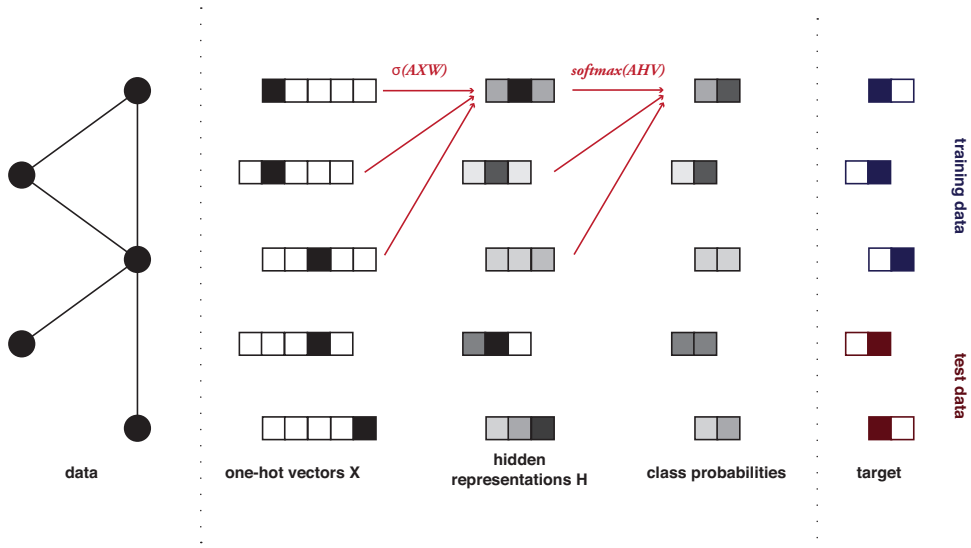


Figure 8. The Graph Convolutional Neural Network. Vertices are represented as one-hot vectors, which are translated to a lower-dimensional space from which class probabilities are obtained with a softmax layer.

consume heterogeneous knowledge, and a data model suited to represent this knowledge naturally. We argue that the knowledge graph is a suitable data model for this purpose.

In this paper we have argued—using three running examples—for the potential of using knowledge graphs: a) they allow for true end-to-end-learning by removing the need for feature engineering, b) they simplify the integration and harmonization of heterogeneous knowledge, and c) they provide a natural way to integrate different forms of background knowledge.

The idea of end-to-end learning on knowledge graphs suggests many research challenges. These include coping with incomplete knowledge, (how to fill the gaps), implicit knowledge (how to exploit implied information), heterogeneous knowledge, (how to process different data types), and differently-modelled knowledge (how to deal with topological diversity). We have shown how several promising approaches already address some of these challenges.

The question may rise whether we are moving the goalposts. Where data scientists were previously faced with the task of creating feature vectors from heterogeneous sources of data, we are now asking them to create a knowledge graph instead. Why should they

bother? We do not claim that the effort will be reduced. We do, however, offer several other reasons to opt for knowledge graphs. Our main claim, as argued throughout this paper: the translation from the original knowledge to knowledge graphs does not remove any information. We are presenting our learning models with the whole of our heterogeneous knowledge or as close a representation as we can make. Relatedly, the knowledge graph is *task-independent*: once created, the same knowledge graph can be used for many tasks, even those beyond machine learning. Finally, because of this re-usability, a great deal of data is already available in knowledge graph form.

End-to-end learning models that can be applied to these graphs off-the-shelf will provide further incentives to knowledge engineers and data owners to produce even more data that is open, well-modeled, and interlinked. We hope that in this way, the Semantic Web and Data Science communities can complement and strengthen one another in a positive feedback loop.

References

- [AMB⁺] A Abele, JP McCrae, P Buitelaar, A Jentzsch, and R Cyganiak. Linking open data cloud diagram.

- <http://lod-cloud.net>. Accessed: 2017-03-01.
- [Bot] Leon Bottou. Two big challenges in machine learning. <http://icml.cc/2015/invited/LeonBottouICML2015.pdf>. Accessed: 2017-03-01.
- [BUGD⁺13] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*, pages 2787–2795, 2013.
- [BZSL13] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.
- [DSS93] Randall Davis, Howard Shrobe, and Peter Szolovits. What is a knowledge representation? *AI magazine*, 14(1):17, 1993.
- [dVdR15] Gerben Klaas Dirk de Vries and Steven de Rooij. Substructure counting graph kernels for machine learning from rdf data. *Web Semantics: Science, Services and Agents on the World Wide Web*, 35:71–84, 2015.
- [FSS09] Thomas Franz, Antje Schultz, Sergej Sizov, and Steffen Staab. Triplerank: Ranking semantic web data by tensor decomposition. *The Semantic Web-ISWC 2009*, pages 213–228, 2009.
- [GMH13] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, speech and signal processing (icassp), 2013 IEEE international conference on*, pages 6645–6649. IEEE, 2013.
- [HS95] Mauricio A Hernández and Salvatore J Stolfo. The merge/purge problem for large databases. In *ACM Sigmod Record*, volume 24, pages 127–138. ACM, 1995.
- [KBV09] Yehuda Koren, Robert M. Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.
- [KW16] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.
- [LDH⁺90] B Boser LeCun, JS Denker, D Henderson, RE Howard, W Hubbard, and LD Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*. Citeseer, 1990.
- [Le13] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.
- [Low99] David G Lowe. Object recognition from local scale-invariant features. In *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*, volume 2, pages 1150–1157. Ieee, 1999.
- [MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.
- [Min] Paul Mineiro. Software engineering vs machine learning concepts. <http://www.machinedlearnings.com/2017/02/software-engineering-vs-machine.html>. Accessed: 2017-03-01.
- [NG15] Thien Huu Nguyen and Ralph Grishman. Relation extraction: Perspective from convolutional neural networks. In *Proceedings of NAACL-HLT*, pages 39–48, 2015.
- [NTK11] Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A three-way model for collective learning on multi-relational data. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 809–816, 2011.
- [PARS14] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.
- [PS16] Heiko Paulheim and Heiner Stuckenschmidt. Fast approximate a-box consistency checking using machine learning. In *International Semantic Web Conference*, pages 135–150. Springer, 2016.
- [PTPV17] Trang Pham, Truyen Tran, Dinh Q. Phung, and Svetha Venkatesh. Column networks for collective classification. In Satinder P. Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, pages 2485–2491. AAAI Press, 2017.
- [RP16] Petar Ristoski and Heiko Paulheim. Rdf2vec: Rdf graph embeddings for data mining. In *International Semantic Web Conference*, pages 498–514. Springer, 2016.
- [SCMN13] Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pages 926–934, 2013.
- [SKB⁺17] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. *arXiv preprint arXiv:1703.06103*, 2017.
- [SPW⁺13] Richard Socher, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, Christopher Potts, et al. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, volume 1631, page 1642. Citeseer, 2013.
- [SSL⁺11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(Sep):2539–2561, 2011.
- [SW16] Baoxu Shi and Tim Weninger. Proje: Embedding projection for knowledge graph completion. *arXiv preprint arXiv:1611.05425*, 2016.