Journal Title 0 (2017) 1–0
IOS Press

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 as raw as possible. 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. In some areas, however, we struggle to find this raw form of data. One such area involves *heterogeneous* knowledge: entities, their attributes and internal relations. The Semantic Web community has invested decades of work on just this problem: how to represent *knowledge*, in various domains, in as raw and as usable a form as possible, satisfying many use cases. This work has led to the Linked Open Data Cloud, a vast and distributed knowledge graph. If we can develop methods that operate on this raw form of data–the knowledge graph–we can dispense with a great deal of ad-hoc feature engineering and train deep models end-to-end in many more domains. In this position paper, we describe current research in this area and discuss 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 often spend most of their time on extracting useful features from their data, they now prefer to feed their models the data in its most raw form. 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 *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 consume raw image data. For facial recognition, deep learning models build up layers of intermediate representations: from low level features 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 raw 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 feed the model with the data in its most raw form, 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, we can let the model surprise us: it may find features in our data that we would never have thought of ourselves.

However, these methods are all domain-specific: they are tailored to images, sound or language. When faced with heterogeneous knowledge—information in different media and from different domains—we often find ourselves resorting back to manual preprocessing. To avoid this, we require a data model suited to such knowledge and a machine learning method capable of consuming data in this format. In this paper we argue that *knowledge graphs* are a more

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

suitable expression of raw heterogeneous knowledge than the feature vectors commonly used in data science.

Concretely, in this paper we will use the phrase *heterogeneous knowledge* to refer to: entities, 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.

To perform, say, classification on the employees in this database, all this information may prove relevant, from an employee's coworkers, to their facial features, to the unstructured information in their biographies. If we resort to manual pre-processing in order to force the data into the mold of feature vectors that traditional methods expect as input, we depart from the ideals of deep learning: we have made ad-hoc decisions, removed information, and fed our model a derivative of the raw data.

The alternative is to shape our knowledge not into feature vectors, but into a data model that can express it naturally and with minimal loss of information. In this paper, we will argue that the knowledge graph is that model. This presents the challenge of designing machine learning methods that can consume knowledge graphs without pre-processing.

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 the knowledge graph for end-to-end learning. 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 though all layers of the architecture, from the output back to the raw 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 machine learning problems to be solved well enough to be 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 to 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 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 (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*. In the Semantic Web, decades of research on knowledge representation can be condensed and simplified into three basic principles:

1. encode knowledge using statements,
2. express background knowledge in ontologies, and
3. reuse knowledge between datasets.

We will briefly discuss each of these next.

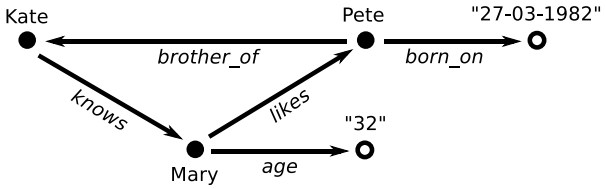

Figure 1. Graphical representation of the example given in Section 2. Edges represent binary relations. Vertices' shapes reflect their roles: solid circles represent entities, empty circles represent their attributes.

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 RDF – *Resource Description Framework* – data model, which is 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* in a left-to-right order. Together, these three are referred to as RDF *triples*. We can also represent this example as a directed graph as shown in Fig. 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. In Fig. 1, the last line represents an attribute of Pete with value "`27-03-1982`".

2.2. Express background knowledge in ontologies

Ontologies are a way to express how knowledge is modeled in a given domain. They contain classes (entity types) and properties that describe the domain as well as constraints and inferences on these classes and properties. For instance, we can define `Person` as the class containing all individual persons. We can likewise define `type` as the property that assigns an entity to a class. As an example, let us use our newly-defined

[1] https://www.w3.org/RDF/

class and property 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 itself, such as that it is equivalent to the class `Human`, disjoint with the class `Animals`, or 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 – ontology[2] 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].
Once expressed, we can use background knowledge to derive implied 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 of the type `Agent` as well.

2.3. Reuse knowledge between datasets

Re-using knowledge can be done by referring to resources not by name, but by unique identifier. On the Semantic Web, these identifiers are called *Internationalized Resource Identifier*, 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 .

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.

Dereferenceable knowledge
: An IRI is more than just an identifier: it is also the location at which a resource or property resides. Because IRIs are often web addresses, we can thus retrieve any data point of which we know the IRI using standard HTTP. This is called dereferencing, and allows for an intuitive way to access external knowledge.

Disambiguated knowledge
: Dereferencing IRIs allows us to directly and unambiguously retrieve relevant information about entities. This includes information about classes and properties in the ontology.

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. 2), 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 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 show how the use of knowledge graphs can remove the need for feature engineering. Feature engineering is a cumbersome task which has to be performed by hand. The data scientist in question

[2] https://www.w3.org/TR/rdf-schema/
[3] https://www.w3.org/OWL/
[4] http://xmlns.com/foaf/spec/

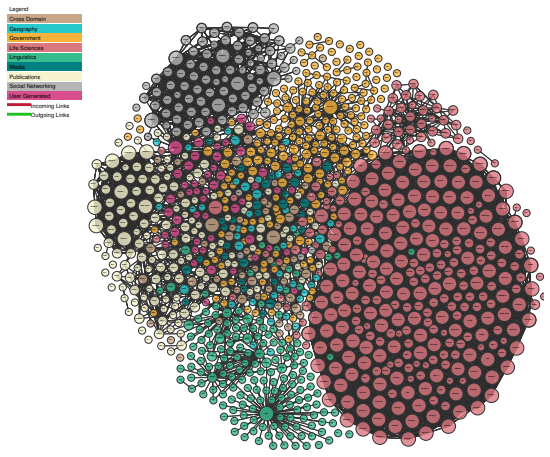

Figure 2. A depiction of the LOD cloud, holding over 38 billion facts from more than 1100 linked datasets. Each vertex represents a separate dataset in the form of a knowledge graph. An edge between two datasets indicates that they share at least one IRI. Figure from [AMB+].

will have to make a judgement in each case whether the added feature is worth the effort. If, instead, we describe data as a knowledge graph, we can define end-to-end models that incorporate the feature engineering as a learnable module.

3.1. Spam detection

Before, we discussed how early spam detection methods classified e-mails based solely on the content of 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 can be used as features in the learning task.

However, for all of this knowledge, we must decide how to translate it to a feature vector, so that our models can learn from it. This 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.

If we have a model that consumes knowledge graphs rather than feature vectors, we can express our knowledge as shown in Fig 3. The task is now to label the vertices representing emails as spam or not spam. We do not need to decide which features contribute to the spam-detection task, we can let the model discover this from the data.

3.2. Movie Recommendation

In most 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. Unfortunately, we do not always have actual ratings yet and are thus unable to start these computations. This *cold-start problem* is common in the traditional setting.

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 such attributes solves the cold start problem because we can link movies and users for which no ratings are available to similar entities through this background data. For instance, if `Indiana_Jones` has no ratings, but we know that the film is directed by Steven Spielberg, which `John` likes, we can recommend `Indiana_Jones` to `John`.

Here again, we run into the problem of manual feature engineering. For each element of background knowledge added, we need to work out how to translate it to a feature vector and how to integrate it into the algorithm we use to produce the rating-based recommendations.

However, consider the knowledge graph fragment shown in Fig 4, which consists of two integrated knowledge graphs: one about movies in general, and another containing movie ratings provided by users. Both refer to movies by the same IRIs, and are thus linked via those resources. On this dataset the recommendation task can be recast as *link prediction*, specifically of the property `liked` from users to movies. Background knowledge and existing ratings can both be used, as their availability allows.

3.3. Market Basket Analysis

To map customer purchase behaviour, we can include more information than simply anonymous transactions. For instance, we can take into account the

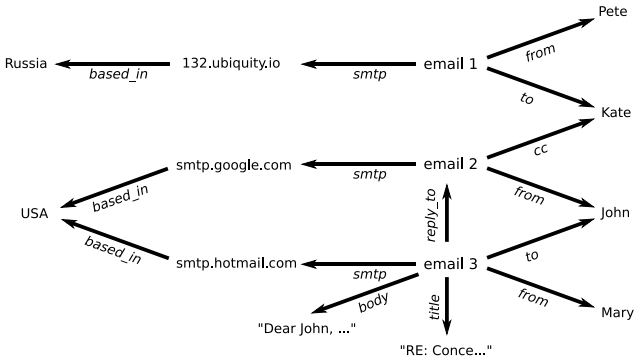

Figure 3. An example dataset on email conversations used in the use case on spam detection of section 3.1.

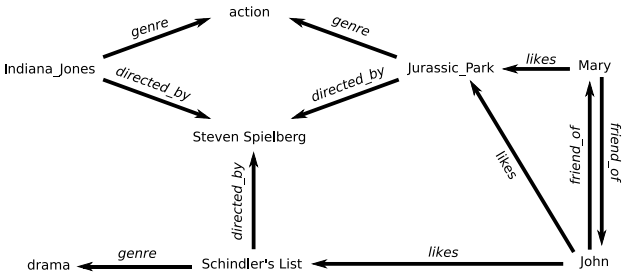

Figure 4. An example dataset on movies and ratings used in the use case on movie recommendations of section 3.2.

current discount on items, whether they are supposedly 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 our marital and financial status. All these attributes can contribute to a precise pattern of our behaviour as customers.

Consider the example shown in Fig 5, which involves a dataset on transactions from a retailer. Each transaction is linked to the items that were bought at that moment. For instance, all three transactions involve the buying of drumsticks. This product can be seen to consist of chicken, which we know due to the coupling of the knowledge graph on transactions with that of product information. We further extended this by linking to external datasets on suppliers and ecological reports.

Where our use of basket analysis has thus far been limited to the discovery of correlations between the items themselves, our algorithms are now able to mine more complex patterns. We might, for example, find that Certified Humane products are often bought together, that people who buy these products also buy those which are eco-friendly, or that products with a high nutritional value are more often bought on sunny days. Of course, we can include the additional attributes ourselves and done some manual feature engineering. But once again, it is far more convenient and effective to train a suitable end-to-end model directly on a knowledge graph, and let *it* learn the most important features itself.

4. The promises and challenges of knowledge graphs

In the previous section, we argued that expressing heterogeneous knowledge in knowledge graphs holds 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. We now discuss several aspects of this problem.

4.1. Incomplete knowledge

In the traditional setting, inflexible data models force knowledge engineers to squeeze their data into an unnatural shape, which often leads to the creation of artificial replacements – `NONE`, `NULL`, `-1`, `99999`, et cetera – for values which are missing or which are simply unknown. To deal with these, data scientists can

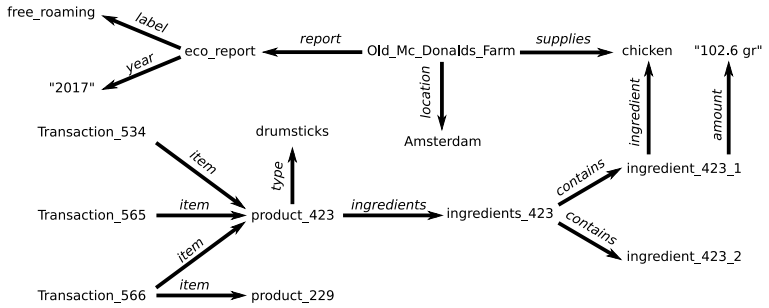

Figure 5. An example dataset on transactions, their items, and additional information used in the use case on market basket analysis of section 3.3.

use one of many available imputation methods. These work well enough for the occasional missing or unknown value, but can become problematic when having to deal with a large number of them. In the case of heterogenous knowledge, we are often faced with large amounts of unavailable data. For many properties in a dataset, there may be more entities for which the data is missing, than for which it is known, often by a factor of ten.

As an example, let us consider a normal social network. For a small amount of users, their profile data is nearly complete: a detailed life history, their birthdate, hobbies, favourite movies and their future plans. For a larger group, only basic information is available: a name, a brief biography, some contact details. For a large majority only sparse knowledge is available: their first name and where they work. In short, for every instance for which we know attribute X, there will be many instances for which it is missing.

Knowledge graphs follow the OWA – Open World Assumption – which implies that only knowledge which we know is true is available: missing or unknown values are now simply omitted altogether. This creates an inherent robustness to incomplete knowledge, and makes the knowledge graph very suitable to model real-world data with. At the same time, this means that we no longer have to concern ourselves with the question of how incomplete knowledge should be dealt with: any method capable of processing knowledge graphs will need to be designed with this characteristic in mind. Concretely, having only a partial view of the world available to us means that we must find a way to translate differently-sized sets of attributes to feature vectors of uniform size.

4.2. Implied knowledge

Knowledge graphs allow knowledge engineers to represent explicit statements as well as background knowledge that can be used to derive implicit statements. The consequence is that in knowledge graphs, many implied statements are not explicitly found in the graph. This avoids redundant and overly large graphs. An example of such implicit knowledge involves class inheritance: for any instance of class C_1 holds that, if $C_1$is a subclass of C_2, it is also an instance of class C_2. The same holds for properties, which can use the `subPropertyOf` property to specify inheritance. This transitivityoffers a wealth of implicit knowledge which can be exploited in learning tasks.

For example, movie recommendation can be improved if learning methods are able to take the genre hierarchy into account. Without this, sub-genres like *action adventure* and *action fantasy* would be seen as completely distinct classes. If the algorithm can consider the subclass hierarchy, it can exploit the fact that both are subgenres of *action*. It can also use the distance in the graph between genres as a similarity function.

4.3. Topology of knowledge

Knowledge graph topologies can vary greatly: some graphs have a relatively shallow hierarchy while others are fairly deep, some require one step to link properties while others use three, some strictly define their semantics while others are lenient, et cetera. Such variations are the result of how different communities choose to represent their knowledge. This freedom releases knowledge engineers from the burden of having to model their knowledge into an unnatural and rigid format.

Typical knowledge graphs are often the combination of several smaller knowledge graphs made by multiple knowledge engineering efforts and can therefore be internally heterogeneous.Creating features from such knowledge graphs can be a challenging task. Without first thoroughly familiarizing ourselves with the

knowledge, we have no way knowing of how the path from a resource to a value is modelled, and whether other parts of the knowledge graph model this differently.

As an example, consider once more use use case of movie recommendation (Fig 4). To model the ratings given by users, we linked users to movies using a single property. 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_26.
Mary_Rating_26 rates Jurassic_Park.
Mary_Rating_26 has_value "1.0".
```

Dealing with knowledge modelled in different ways remains one of the challenges 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.

4.4. Literal 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. This in contrast to non-literal resources, here their local neighbourhood is the 'value'. Simply treating literals the same as these other resources will therefore be ineffective. Concretely, treating both the same implies that all literals are unique and evenly separated. Instead, we must deal with each different data type separately and accordingly: texts as strings, numbers and dates as ordinal values, IDs as nominal values, et cetera.

Considering our spam detection example, we have modelled an e-mail's title and body using literals. If we would treat these as all other resources, our learning method would perceive the title *"Just saying hello!"* as totally different from its reply *"RE: Just saying hello!"*. Rather, we can alternatively design our method to compare title values using a string similarity measure, and discover that these two titles are actually very similar.

Exploiting literal knowledge when learning from knowledge graphs is still in its infancy, and typically reserved for very specific use cases. For example, all methods discussed in the next section either ignore statements containing literals or treat literals as unique vertices. Nevertheless, we can already effectively deal with such knowledge in traditional machine learning scenarios. It is therefore a challenge to integrate this experience into methods that can learn from knowledge graphs.

4.5. Which dataset? the *dataset*

Knowledge graphs present a new perspective on datasets. Where we traditionally have a multitude of datasets, about countless of domains and tailored to numerous different tasks, we now have only one: a single vast distributed knowledge graph. Because everything is stored in the same data model (RDF), learning on this entire knowledge graph will be no different from learning on any of its countless of subgraphs. We can therefore simply take any subgraph, which itself is a valid knowledge graph, and use it as input to our end-to-end model. If we want to change domain, scope, or granularity of our dataset, we select a different subgraph and rerun our experiment. Similarly, if our learning task changes, we only need to pick a different model while the dataset can remain the same. In either case, we greatly reduce the effort of preparing our setup by largely skipping the pre-processing step.

For the LOD Cloud, there are multiple strategies to recover relevant subgraphs. This can be done by dereferencing individual IRIs – follow-your-nose –, by downloading datasets directly or through query endpoints. Furthermore, LOD search engines allow for identifying and retrieving datasets. Already, this approach is being used to construct benchmark datasets for machine learning [RdVP16].

5. Current approaches

Only recently has the machine learning community began to show interest in the knowledge graph. Initial explorations focused primarily on how knowledge graphs can be 'flattened' into plain tables for use with traditional learning methods, whereas more recent studies are looking for more natural ways to process knowledge graphs. This has lead to various data models. An early class of methods represent knowledge using logical rules and use variants of Inductive Logic Programming (such as, the recent Amie system[GTHS13]) or Probabilistic Soft Logic [BBHG15] to infer new knowledge. More recently, tensor representations, embeddings and graph convolutional networks have become prominent meth-

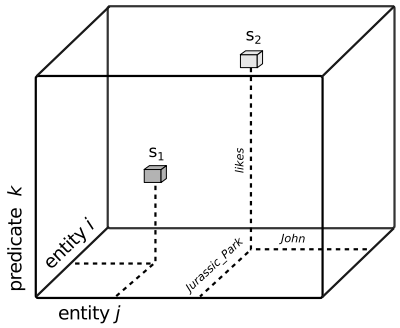

Figure 6. Representing statements as points in a third-order tensor. Two statements are illustrated: s_1 and s_2, with s_2 = *John likes Jurassic_Park*

ods which present opportunities for end-to-end modeling.

5.1. Tensor representation

Tensors are 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 Fig 6. As an example, let i, j, k be the indices of a tensor $\mathbf{T}$ used to represent our dataset on movie recommendations (Fig 4). If now $\mathbf{T}[i] = \textit{John}$, $\mathbf{T}[j] = \textit{Jurassic\_Park}$, and $\mathbf{T}[k] = \textit{likes}$, 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 intersection 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 intersections 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 layer in a regular [SCMN13] or recursive neural network [SPW+13].

5.2. Embeddings

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 we represents an instance in our training data (the *instance vertices*). For instance, in our email example, we could simply transform each vertex representing an email into a feature vector, and then apply any off-the-shelf classifier to perform classification. The translation of knowledge graphs to feature vectors is also known as *propositionalization* or *embedding*.

5.2.1. 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 equally well be used to generate explicit feature vectors, so we will not view them as kernels here.

The simplest form of substructure counting method takes the k-neighborhood around the 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. 3), 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 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.

5.2.2. 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 features 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 dimension, 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. 5), a context of depth 3 allows RDF2Vec to represent each transaction via chains such as `transaction_X` → `ingredients_X` → `ingredient_Y` and `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].

5.3. Graph Convolutional Neural Networks

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

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 vector[5] 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 X^0 is the N by 3 matrix containing all original vertex features we can define this principle mathematically as $X^{k+1} = AX^k$, where A is the normalized adjacency matrix of G. If we start with one-hot vectors instead of colors, x^k becomes a probability vector with x^k_j 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 (Figure **??**) uses these ideas to create a differentiable map from one vector representation into another. We start with a matrix of one-hot vectors X. These are multiplied by A, and then translated down to a lower dimensional feature space by a matrix W. 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 H

$$H = f^{\sigma}_{W}(X) = \sigma(AXW).$$

Row i of matrix 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", given the second a softmax restriction on the output vectors. This gives us a length-M probability vector y for each vertex, representing the classification.

[5] A 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.

Thus, the complete model becomes

$$Y = f_V^{\text{softmax}}\left(A f_W^{\sigma}(AXW)V\right),$$

where X is the identity matrix (i.e. a stack of one-hot vectors for each vertex), and Y is an $N \times M$ matrix with Y_{ij} the probability that vertex i has class j. We then learn the weights V and W by minimizing the cross-entropy between the training examples and the corresponding rows of 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, they 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. We do however, pay a price: we must store the whole graph in memory. In the case of RDF2Vec, we only need to perform random walks on the graph, which suggests the method should scale to almost any size.

6. Conclusion

Traditionally, when faced with heterogeneous knowledge in a machine learning context, data scientists preprocess the data and engineer feature vectors so they can be used as input for learning algorithms (e.g., for classification). These transformations can result in loss of information and introduce bias. To solve this problem, we require machine learning methods to consume knowledge in a data model more suited to represent this heterogeneous knowledge. We argue that knowledge graphs are that data model.

In this paper, we have shown—using three examples—the benefits of using knowledge graphs: a) they allow for true end-to-end-learning, b) they simplify the integration of heterogeneous data sources and data harmonization, and c) they provide a natural way to seamlessly integrate different forms of background knowledge.

The idea of end-to-end learning on knowledge graphs suggests many research challenges. These include coping with missing data, integrating heterogeneous structures in the data (such as graph structure, natural language and numeric values) and including implied knowledge through ontologies. We show how several promising approaches address these challenges.

There is currently an abundance of well-constructed knowledge graphs in the form of the LOD cloud. End-to-end learning models that can be applied to these graphs to derive useful insights will provide further incentives to produce even more data that is open, well-modeled, and interlinked. In this way, we are confident that 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.

[BBHG15] Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. Hinge-loss markov random fields and probabilistic soft logic. *CoRR*, abs/1505.04406, 2015.

[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.

[FSSS09] 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.

[GTHS13] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. Amie: association rule mining under incomplete evidence in ontological knowledge bases. In *Proceedings of the 22nd international conference on World Wide Web*, pages 413–422. ACM, 2013.

[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.

[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.

[RdVP16] Petar Ristoski, Gerben Klaas Dirk de Vries, and Heiko Paulheim. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In *International Semantic Web Conference*, pages 186–194. Springer, 2016.

[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.