Knowledge Representation and Rule Mining in Entity-Centric Knowledge Bases

Fabian M. Suchanek[1], Jonathan Lajus[1], Armand Boschin[1], and Gerhard Weikum[2]

[1] Telecom Paris, Institut Polytechnique de Paris
[2] Max Planck Institute for Informatics

Abstract. Entity-centric knowledge bases are large collections of facts about entities of public interest, such as countries, politicians, or movies. They find applications in search engines, chatbots, and semantic data mining systems. In this paper, we first discuss the knowledge representation that has emerged as a pragmatic consensus in the research community of entity-centric knowledge bases. Then, we describe how these knowledge bases can be mined for logical rules. Finally, we discuss how entities can be represented alternatively as vectors in a vector space, by help of neural networks.

1 Introduction

1.1 Knowledge Bases

When we send a query to Google or Bing, we obtain a set of Web pages. However, in some cases, we also get more information. For example, when we ask "When was Steve Jobs born?", the search engine replies directly with "February 24, 1955". When we ask just for "Steve Jobs", we obtain a short biography, his birth date, quotes, and spouse. All of this is possible because the search engine has a huge repository of knowledge about people of common interest. This knowledge takes the form of a knowledge base (KB).

The KBs used in such search engines are entity-centric: they know individual entities (such as *Steve Jobs*, the *United States*, the *Kilimanjaro*, or the *Max Planck Society*), their semantic classes (such as *SteveJobs is-a computer-Pioneer, SteveJobs is-a entrepreneur*), relationships between entities (e.g., *SteveJobs founded AppleInc, SteveJobs hasInvented iPhone, SteveJobs hasWonPrize NationalMedalOfTechnology*, etc.) as well as their validity times (e.g., *SteveJobs wasCEOof Pixar [1986,2006]*).

The idea of such KBs is not new. It goes back to seminal work in Artificial Intelligence on universal knowledge bases in the 1980s and 1990s, most notably, the Cyc project [41] at MCC in Austin and the WordNet project [19] at Princeton University. These knowledge collections were hand-crafted and manually curated. In the last ten years, in contrast, KBs are often built automatically by extracting information from the Web or from text documents. Salient projects with publicly available resources include KnowItAll (UW Seattle, [17]), ConceptNet (MIT,

[44]), DBpedia (FU Berlin, U Mannheim, & U Leipzig, [40]), NELL (CMU, [9]), BabelNet (La Sapienza, [58]), Wikidata (Wikimedia Foundation, [77]), and YAGO (Telecom Paris & Max Planck Institute, [70]). Commercial interest in KBs has been strongly growing, with projects such as the Google Knowledge Graph [15] (including Freebase [6]), Microsoft's Satori, Amazon's Evi, LinkedIn's Knowledge Graph, and the IBM Watson KB [20]. These KBs contain many millions of entities, organized in hundreds to hundred thousands of semantic classes, and hundred millions of relational facts between entities. Many public KBs are interlinked, forming the Web of Linked Open Data [5].

1.2 Applications

KBs are used in several applications, including the following:

Semantic Search and Question Answering. Both the Google search engine [15] and Microsoft Bing[3] use KBs to give intelligent answers to queries, as we have seen above. They can answer simple factual questions, provide movie showtimes, or show a list of "best things to do" at a travel destination. Wolfram Alpha[4] is another prominent example of a question answering system. The IBM Watson system [20] used knowledge from a KB to win against human champions in the TV quiz show Jeopardy.

Intelligent Assistants. Chatbots such as Apple's Siri, Amazon's Alexa, Google's Allo, or Microsoft's Cortana aim to help a user achieve daily tasks. The bots can, e.g., suggest restaurants nearby, answer simple factual questions, or manage calendar events. The background knowledge that the bots need for this work usually comes from a KB. With embodiments such as Amazon's Echo system or Google Home, such assistants will share more and more people's homes in the future. Other companies, too, are experimenting with chat bots that treat customer requests or provide help to users.

Semantic Data Mining. Daily news, social media, scholarly publications, and other Web contents are the raw inputs for analytics to obtain insights on business, politics, health, and more. KBs can help to discover and track entities and relationships in order to generate opinion maps, informative recommendations, and other kinds of intelligence towards decision making. For example, we can mine the gender bias from newspapers, because the KB knows the gender of people (see [71] for a survey). There is an entire domain of research dedicated to "predictive analytics", i.e., the prediction of events based on past events.

1.3 Knowledge Representation and Rule Mining

In this article, we first discuss how the knowledge is usually represented in entity-centric KBs. The field of knowledge representation has a long history,

[3] `http://blogs.bing.com/search/2013/03/21/understand-your-world-with-bing/`
[4] `https://wolframalpha.com`

and goes back to the early days of Artificial Intelligence. It has developed numerous knowledge representation models, from frames and KL-ONE to recent variants of description logics. The reader is referred to survey works for comprehensive overviews of historical and classical models [62,67]. In this article, we discuss the knowledge representation that has emerged as a pragmatic consensus in the research community of entity-centric knowledge bases.

In the second part of this article, we discuss logical rules on knowledge bases. A logical rule can tell us, e.g., that if two people are married, then they (usually) live in the same city. Such rules can be mined automatically from the knowledge base, and they can serve to correct the data or fill in missing information. We discuss first classical Inductive Logic Programming approaches, and then show how these can be applied to the case of knowledge bases.

In the third part of this article, we discuss an alternative way to represent entities: as vectors in a vector space. Such so-called embeddings can be learned by neural networks from a knowledge base. The embeddings can then help deduce new facts – much like logical rules.

2 Knowledge Representation

2.1 Entities

2.1.1 Entities of Interest

The most basic element of a KB is an *entity*. An entity is any abstract or concrete object of fiction or reality, or, as Bertrand Russell puts it in his *Principles of Mathematics* [81]:

Definition 1 (Entity): *An entity is whatever may be an object of thought.*

This definition is completely all-embracing. Steve Jobs, the Declaration of Independence of the United States, the Theory of Relativity, and a molecule of water are all entities. Events (such as the French Revolution), are entities, too. An entity does not even have to exist: Harry Potter, e.g., is a fictional entity. Phlogiston was presumed to be the substance that makes up heat. It turned out to not exist – but it is still an entity.

KBs model a part of reality. This means that they choose certain entities of interest, give them names, and put them into a structure. Thus, a KB is a structured view on a selected part of the world. KBs typically model only distinct entities. This cuts out a large portion of the world that consists of variations, flows and transitions between entities. Drops of rain, for instance, fall down, join in a puddle and may be splattered by a passing car to form new drops [66]. KBs will typically not model these phenomena. This choice to model only discrete entities is a projection of reality; it is a grid through which we see only distinct things. Many entities consist of several different entities. A car, for example, consists of wheels, a bodywork, an engine, and many other pieces. The engine consists of the pistons, the valves, and the spark plug. The valves consist again of several parts, and so on, until we ultimately arrive at the level of atoms or

below. Each of these components is an entity. However, KBs will typically not be concerned with the lower levels of granularity. A KB might model a car, possibly its engine and its wheels, but most likely not its atoms. In all of the following, we will only be concerned with the entities that a KB models.

Entities in the real world can change gradually. For example, the Greek philosopher Eubilides asks: If one takes away one molecule of an object, will there still be the same object? If it is still the same object, this invites one to take away more molecules until the object disappears. If it is another object, this forces one to accept that two distinct objects occupy the same spatio-temporal location: The whole and the whole without the molecule. A related problem is the question of identity. The ancient philosopher Theseus uses the example of a ship: Its old planks are constantly being substituted. One day, the whole ship has been replaced and Theseus asks, "Is it still the same ship?". To cope with these problems, KBs typically model only atomic entities. In a KB, entities can only be created and destroyed as wholes.

2.1.2 Identifiers and Labels

In computer systems (as well as in writing of any form), we refer to entities by *identifiers*.

Definition 2 (Identifier): *An identifier for an entity is a string of characters that represents the entity in a computer system.*

Typically, these identifiers take a human-readable form, such as *ElvisPresley* for the singer Elvis Presley. However, some KBs use abstract identifiers. Wikidata, e.g., refers to Elvis Presley by the identifier *Q303*, and Freebase by */m/02jq1*. This choice was made so as to be language-independent, and so as to provide an identifier that is stable in time. If, e.g., Elvis Presley reincarnates in the future, then *Q303* will always refer to the original Elvis Presley. It is typically assumed that there exists exactly one identifier per entity in a KB. For what follows, we will not distinguish identifiers from entities, and just talk of entities instead.

Entities have names. For example, the city of New York can be called "city of New York", "Big Apple", or "Nueva York". As we see, one entity can have several names. Vice versa, the same name can refer to several entities. "Paris", e.g., can refer to the city in France, to a city of that name in Texas, or to a hero of Greek mythology. Hence, we need to carefully distinguish *names* – single words or entire phrases – from their *senses* – the entities that they denote. This is done by using labels.

Definition 3 (Label): *A label for an entity is a human-readable string that names the entity.*

If an entity has several labels, the labels are called synonymous. If the same label refers to several entities, the label is polysemous. Not all entities have labels. For example, your kitchen chair is clearly an entity, but it probably does not have any particular label. An entity that has a label is called a *named entity*.

KBs typically model mainly named entities. There is one other type of entities that appears in KBs: literals.

Definition 4 (Literal): *A literal is a fixed value that takes the form of a string of characters.*

Literals can be pieces of text, but also numbers, quantities, or timestamps. For example, the label "Big Apple" for the city of New York is a literal, as is the number of its inhabitants (8,175,133).

2.2 Classes

2.2.1 Classes and Instances

KBs model entities of the world. They usually group entities together to form a *class*:

Definition 5 (Class): *A class (also: concept, type) is a named set of entities that share a common trait. An element of that set is called an instance of the class.*

Under this definition, the following are classes: The class of singers (i.e., the set of all people who sing professionally), the class of historical events in Latin America, and the class of cities in Germany. Some instances of these classes are, respectively, Elvis Presley, the independence of Argentina, and Berlin. Since everything is an entity, a class is also an entity. It has (by definition) an identifier and a label.

Theoretically, KBs can form classes based on arbitrary traits. We can, e.g., construct the class of singers whose concerts were the first to be broadcast by satellite. This class has only one instance (Elvis Presley). We can also construct the class of left-handed guitar players of Scottish origin, or of pieces of music that the Queen of England likes. There are several theories as to whether humans actually build and use classes, too [46]. Points of discussion are whether humans form crisp concepts, and whether all elements of a concept have the same degree of membership. For the purpose of KBs, however, classes are just sets of entities.

It is not always easy to decide whether something should be modeled as an instance or as a class. We could construct, e.g., for every instance a singleton class that contains just this instance (e.g., the class of all Elvis Presleys). Some things of the world can be modeled both as instances and as classes. A typical example is *iPhone*. If we want to designate the type of smartphone, we can model it as an instance of the class of smartphone brands. However, if we are interested in the iPhones owned by different people and want to capture them individually, then *iPhone* should be modeled as a class. A similar observation holds for abstract entities such as *love*. Love can be modeled as an instance of the class *emotion*, where it resides together with the emotions of *anger*, *fear*, and *joy*. However, when we want to model individual feelings of love, then *love* would be a class. Its instances are the different feelings of love that different people have. It is our choice how we wish to model reality.

A pragmatic test of whether something should be modeled as a class is as follows: If we are interested in the plural form of a word or phrase, then we should model it as a class. If we talk, e.g., about "iPhones", then we model several instances of iPhones, and hence *iPhone* should be a class. If we only talk about "iPhone" along with other brand names (such as "HTC One"), then *iPhone* may well be considered an instance. Analogously, if we talk of "love" only in singular, then we may model it as an instance, along with other emotions. If we talk of "loves" (as in "Elvis had many loves during his time as a star"), then *love* is the set of all love affairs – and thus a class. The reason for this test is that only countable nouns can be classes, and only countable nouns can be put into plural. Another method to distinguish classes from instances is to say "An X", or "Every X". If that is possible, then X is best modeled as a class, because it can have instances. For example, it is possible to say "a CEO", but not "a Steve Jobs". Hence, *ceo* should be a class, and *SteveJobs* should not. If we can say "This is X", then X is an instance – as in "This is Steve Jobs". If we can say "X is a Y", then X is an instance of Y – as in "Steve Jobs is a CEO".

A particular case are mass nouns like "milk". The word "milk" (in the sense of the liquid) does not have a plural form. Therefore, we could model it as an instance (e.g., as an instance of the class of liquids). However, if we are interested in individual servings of milk, such as bottles of milk, then we can model it as a class, *servingOfMilk*.

Some KBs do not make the distinction between classes and instances (e.g., the SKOS vocabulary, [84]). In these KBs, everything is an entity. There is, however, usually a "is more general than" link between a more special entity and a more general entity. Such a KB may contain, e.g., the knowledge that *iPhone* is more special than *smartphone*, without worrying whether one of them is a class. The distinction between classes and instances adds a layer of granularity. This granularity is used, e.g., to define the domains and ranges of relations, as we shall see in Section 2.3.

2.2.2 Taxonomies

Definition 6 (Subsumption): *Class A is a subclass of class B if A is a subset of B.*

For example, the class of singers is a subclass of the class of persons, because every singer is a person. We also say that the class of singers is a *specialization* of the class of persons, or that singer *is subsumed by* or *included in* person. Vice versa, we say that person is a *superclass* or a *generalization* of the class of singers. Technically speaking, two equivalent classes are subclasses of each other. This is the way the RDFS standard models subclasses [83]. We say that a class is a *proper subclass* of another class, if the second contains more entities than the first. We use the notion of subclass here to refer to *proper* subclasses only.

It is important not to confuse class inclusion with the relationship between parts and wholes. For example, an arm is a part of the human body. That does not mean, however, that every arm is a human body. Hence, *arm* is not a subclass

of *body*. In a similar manner, New York is a part of the US. That does not mean that New York would be a subclass of the US. Neither New York nor the US are classes, so they cannot be subclasses of each other.

Class inclusion is transitive: If A is a subclass of B, and B is a subclass of C, then A is a subclass of C. For example, *viper* is a subclass of *snake*, and *snake* is a subclass of *reptile*. Hence, by transitivity, *viper* is also a subclass of *reptile*. We say that a class is a *direct subclass* of another class, if there is no class in the KB that is a superclass of the former and a subclass of the latter. When we talk about subclasses, we usually mean only direct subclasses. The other subclasses are *transitive subclasses*. Since classes can be included in other classes, they can form an inclusion hierarchy – a taxonomy.

Definition 7 (Taxonomy): *A taxonomy is a directed graph, where the nodes are classes and there is an edge from class X to class Y if X is a proper direct subclass of Y.*

The notion of taxonomy is known from biology. Zoological or botanic species form a taxonomy: *tiger* is a subclass of *cat*. *cat* is a subclass of *mammal*, and so on. This principle carries over to all other types of classes. We say, e.g., that *internetCompany* is a subclass of *company*, and that *company* is a subclass of *organization*, etc. Since a taxonomy models proper inclusion, it follows that the taxonomic graph is acyclic: If a class is the subclass of another class, then the latter cannot be a subclass of the former. Thus, a taxonomy is a directed acyclic graph. A taxonomy does not show the transitive subclass edges. If the graph contains transitive edges, we can always remove them. Given a finite directed acyclic graph with transitive edges, the set of direct edges is unique [2].

Transitivity is often essential in applications. For example, consider a question-answering system where a user asks for artists that are married to actors. If the KB only knew about Elvis Presley and Priscilla Presley being in the classes *rockSinger* and *americanActress*, the question could not be answered. However, by reasoning that *rockSingers* are also *singers*, who in turn are *artists* and *americanActresses* being *actresses*, it becomes possible to give this correct answer.

Usually (but not necessarily), taxonomies are connected graphs: Every node in the graph is, directly or indirectly, linked to every other node. Usually, the taxonomies have a single root, i.e., a single node that has no outgoing edges. This node identifies the most general class, of which every other class is a subclass. In zoological KBs, this may be class *animal*. In a person database, it may be the class *person*. In a general-purpose KB, this class has to be the most general possible class. In YAGO and Wordnet, the class is *entity*. In the RDF standard, it is called *resource* [82]. In the OWL standard [85], the highest class that does not include literals is called *thing*.

Some taxonomies have at most one outgoing edge per node. Then, the taxonomy forms a tree. The biological taxonomy, e.g., forms a tree, as does the Java class hierarchy. However, there can be taxonomies where a class has two distinct direct superclasses. For example, if we have the class *singer* and the classes of *woman* and *man*, then the class *femaleSinger* has two superclasses: *singer* and

woman. Note that it would be wrong to make *singer* a subclass of *man* and *woman* (as if to say that singers can be men or women). This would actually mean that all singers are at the same time men and women.

When a taxonomy includes a "combination class" such as *FrenchFemaleSingers*, then this class can have several superclasses. *FrenchFemaleSingers*, e.g., can have as direct superclasses *FrenchPeople*, *Women*, and *Singers*. In a similar manner, one entity can be an instance of several classes. Albert Einstein, e.g., is an instance of the classes *physicist*, *vegetarian*, and *violinPlayer*.

When we populate a KB with new instances, we usually try to assign them to the most specific suitable class. For example, when we want to place *Bob Dylan* in our taxonomy, we would put him in the class *americanBluesSinger*, if we have such a class, instead of in the class *person*. However, if we lack more specific information about the instance, then we might be forced to put it into a general class. Some named entity recognizers, e.g., distinguish only between organizations, locations, and people, which means that it is hard to populate more specific classes. It may also happen that our taxonomy is not specific enough at the leaf level. For example, we may encounter a musician who plays the Arabic oud, but our taxonomy does not have any class like *oudPlayer*. Therefore, a class may contain more instances than the union of its subclasses. That is, for a class C with subclasses $C_1, \ldots, C_k$, the invariant is $\cup_{i=1..k} C_k \subseteq C$, but $\cup_{i=1..k} C_k = C$ is often false.

2.2.3 Special Cases

Some KBs assign literals to classes, too. For example, the literal "Hello" can be modeled as an instance of the class *string*. Such literal classes can also form taxonomies. For example, the class *nonNegativeIntegers* is a subclass of the class of *integers*, which is again a subclass of the more general class *numbers*.

We already observed that classes are entities. Thus, we can construct classes that contain other classes as instances. For example, we can construct the class of all classes *class* ={*car, person, scientist, ...*}. This leads to awkward questions about self-containment, reminiscent of Bertrand Russel's famous set of sets that do not include themselves. The way this is usually solved [82] is to distinguish the class (as an abstract concept) from the extension of the class (the set of its instances). For example, the class of singers is the abstract concept of people who sing. Its extension is the set {*Elvis*, *Madonna*, ...}. In this way, a class is not a set, but just an abstract entity. Therefore, the extension of a class can contain another class. This is, however, a rather theoretical problem, and in what follows, we will not distinguish classes from their extensions.

To distinguish classes from other entities, we call an entity that is neither a class nor a literal an *instance* or a *common entity*.

2.3 Relations

2.3.1 Relations and Statements

KBs model also *relationships* between entities:

Definition 8 (Relation): *A relationship (also: relation) over the classes* $C_1, ..., C_n$ *is a named subset of the Cartesian product* $C_1 \times ... \times C_n$.

For example, if we have the classes *person*, *city*, and *year*, we may construct the *birth* relationship as a subset of the cartesian product *person*×*city*×*year*. It will contain tuples of a person, their city of birth, and their year of birth. For example, $\langle ElvisPresley, Tupelo, 1935\rangle \in birth$. In a similar manner, we can construct *tradeAgreement* as a subset of *country*×*country*×*commodity*. This relation can contain tuples of countries that made a trade agreement concerning a commodity. Such relationships correspond to classical relations in algebra or databases.

As always in matters of knowledge representation (or, indeed, informatics in general), the identifier of a relationship is completely arbitrary. We could, e.g., call the *birth* relationship *k42*, or, for that matter, *death*. Nothing hinders us to populate the *birth* relationship with tuples of a person, and the time and place where that person ate an ice cream. However, most KBs aim to model reality, and thus use identifiers and tuples that correspond to real-world relationships.

If $\langle x_1, ..., x_n\rangle \in R$ for a relationship R, we also write $R(x_1, ..., x_n)$. In the example, we write $birth(ElvisPresley, Tupelo, 1935)$. The classes of R are called the *domains* of R. The number of classes n is called the *arity* of R. $\langle x_1, ..., x_n\rangle$ is a *tuple of* R. $R(x_1, ..., x_n)$ is called a *statement*, *fact*, or *record*. The elements $x_1, ..., x_n$ are called the *arguments* of the facts. Finally, a *knowledge base*, in its simplest form, is a set of statements. For example, a KB can contain the relations *birth*, *death* and *marriage*, and thus model some of the aspects of people's lives.

2.3.2 Binary Relations

Definition 9 (Binary Relation): *A binary relation is a relation of arity 2.*

Examples of binary relations are *birthPlace, friendOf*, or *marriedTo*. The first argument of a binary fact is called the *subject*, and the second argument is called the *object* of the fact. The relationships are sometimes called *properties*. Relationships that have literals as objects, and that have at most one object per subject are sometimes called *attributes*. Examples are *hasBirthDate* or *hasISBN*. The *domain* of a binary relation $R \subset A \times B$ is A, i.e., the class from which the subjects are taken. B is called the *range* of R. For example, the domain of *birthPlace* is *person*, and its range is *city*. The *inverse* of a binary relation R is a relation R^{-1}, such that $R^{-1}(x, y)$ iff $R(x, y)$. For example, the inverse relation of *hasNationality* (between a person and a country) is *hasNationality*$^-$ (between a country and a person) – which we could also call *hasCitizen*.

Any n-ary relation R with $n > 2$ can be split into n binary relations. This works as follows. Assume that there is one argument position i that is a *key*,

i.e., every fact $R(x_1, ..., x_n)$ has a different value for x_i. In the previously introduced 3-ary *birth* relationship, which contains the person, the birth place, and the birth date, the person is the key: every person is born only once at one place. Without loss of generality, let the key be at position $i = 1$. We introduce binary relationships $R_2, ..., R_n$. In the example, we introduce *birthPlace* for the relation between the person and the birth place, and *birthDate* for the relation between the person and the birth year. Every fact $R(x_1, ..., x_n)$ gets rewritten as $R_2(x_1, x_2), R_3(x_1, x_3), R_4(x_1, x_4), ..., R_n(x_1, x_n)$. In the example, the fact *birth(Elvis,Tupelo,1935)* gets rewritten as *birthPlace(Elvis,Tupelo)* and *birthDate(Elvis,1935)*. Now assume that a relation R has no key. As an example, consider again the *tradeAgreement* relationship. Obviously, there is no key in this relationship, because any country can make any number of trade-agreements on any commodity. We introduce binary relationships $R_1, ...R_n$ for every argument position of R. For *tradeAgreement*, these could be *country1*, *country2* and *tradeCommodity*. For each fact of R, we introduce a new entity, an *event entity*. For example, if the US and Brazil make a trade-agreement on coffee, *tradeAgreement(Brazil,US,Coffee)*, then we create *coffeeAgrBrUs*. This entity represents the fact that these two countries made this agreement. In general, every fact $R(x_1, ..., x_n)$ gives rise to an event entity $e_{x1,...,xn}$. Then, every fact $R(x_1, ..., x_n)$ is rewritten as $R_1(e_{x1,...,xn}, x_1), R_2(e_{x1,...,xn}, x_2), ..., R_n(e_{x1,...,xn}, x_n)$. In the example, *country1(coffeeAgrBrUs, Brazil), country2(coffeeAgrBrUs, US), tradeCommodity(coffeeAgrBrUs, Coffee)*. This way, any n-ary relationship with $n > 2$ can be represented as binary relationships. For $n = 1$, we can always invent a binary relation *hasProperty*, and use the relation as an additional argument. For example, instead of *male(Elvis)*, we can say *hasProperty(Elvis, male)*.

The advantage of binary relationships is that they can express facts even if one of the arguments is missing. If, e.g., we know only the birth year of Steve Jobs, but not his birth place, then we cannot make a fact with the 3-ary relation *birth* $\subset$ *person*$\times$*city*$\times$*year*. We have to fill the missing arguments, e.g., with *null* values. If the relationship has a large arity, many of its arguments may have to be null values. In the case of binary relationships, in contrast, we can easily state *birthDate(SteveJobs, 1955)*, and omit the *birthPlace* fact. Another disadvantage of n-ary relationships is that they do not allow adding new pieces of information a posteriori. If, e.g., we forgot to declare the astrological ascendant as an argument to the 3-ary relation *birth*, then we cannot add the ascendant for Steve Job's birth without modifying the relationship. In the binary world, in contrast, we can always add a new relationship *birthAscendant*. Thus, binary relationships offer more flexibility. This flexibility can be a disadvantage, because it allows adding incomplete information (e.g., a birth place without a birth date). However, since knowledge bases are often inherently incomplete, binary relationships are usually the method of choice.

2.3.3 Functions

Definition 10 (Function): *A function is a binary relation that has for each subject at most one object.*

Typical examples for functions are *birthPlace* and *hasLength*: Every person has at most one birth place and every river has at most one length. The relation *ownsCar*, in contrast, is not a function, because a (rich) person can own multiple cars. In our terminology, we call a relation a function also if it has no objects for certain subjects, i.e., we include partial functions (such as *deathDate*).

Some relations are *functions in time*. This means that the relation can have several objects, but at each point of time, only one object is valid. A typical example is *isMarriedTo*. A person can go through several marriages, but can only have one spouse at a time (in most systems). Another example is *hasNumberOfInhabitants* for cities. A city can grow over time, but at any point of time, it has only a single number of inhabitants. Every function is a function in time.

A binary relation is an *inverse function*, if its inverse is a function. Typical examples are *hasCitizen* (if we do not allow double nationality) or *hasEmailAddress* (if we talk only about personal email addresses that belong to a single person). Some relations are both functions and inverse functions. These are identifiers for objects, such as the social security number. A person has exactly one social security number, and a social security number belongs to exactly one person. Functions and inverse functions play a crucial role in entity matching: If two KBs talk about the same entity with different names, then one indication for this is that both entities share the same object of an inverse function. For example, if two people share an email address in a KB about customers, then the two entities must be identical.

Some relations are "nearly functions", in the sense that very few subjects have more than one object. For example, most people have only one *nationality*, but some may have several. This idea is formalized by the notion of *functionality* [69]. The functionality of a relation r in a KB is the number of subjects, divided by the number of facts with that relation:

$$fun(r) := \frac{|\{x : \exists y : r(x,y)\}|}{|\{x,y : r(x,y)\}|}$$

The functionality is always a value between 0 and 1, and it is 1 if r is a function. It is undefined for an empty relation.

We usually have the choice between using a relation and its inverse relation. For example, we can either have a relationship *isCitizenOf* (between a person and their country) or a relationship *hasCitizen* (between a country and its citizens). Both are valid choices. In general, KBs tend to choose the relation with the higher functionality, i.e., where the subject has fewer objects. In the example, the choice would probably be *isCitizenOf*, because people have fewer citizenships than countries have citizens. The intuition is that the facts should be "facts about the subject". For example, the fact that two authors of this

paper are citizens of Germany is clearly an important property of the authors (it appears on the Wikipedia page of the last author). Vice versa, the fact that Germany is fortunate enough to count these authors among its citizens is a much less important property of Germany (it does not appear on the Wikipedia page of Germany).

2.3.4 Relations with Classes

In Section 2.2.3, we have introduced the class *class*, which contains all classes. This allows us to introduce the relationship between an instance and its class: *type*$\subset$*entity*$\times$*class*. We can now say *type(Elvis, singer)*.[5] We also introduce *subclassOf*$\subset$*class*$\times$*class*, which is the relationship between a class and its superclasses. For example, *subclassOf(singer, person)*. In the same way as we have introduced the class of all classes, we can introduce the class of all relations. We call this class *property*. With this, we can define the relationship between a binary relation and its domain: *domain*$\subset$*property*$\times$*class*. We can now say *domain(birthPlace, person)*. Analogously, we introduce *range*$\subset$*property*$\times$*class*, so that we can say *range(birthPlace, city)*. This way, an entire KB, with its relations and schema information, can be written as binary relationships. There is no distinction between data and meta-data – the KB describes itself.

In some cases, we have the choice whether to model something as a relationship or as a class. For example, to say that Berlin is located in Germany, we can either say *locatedIn(Berlin, Germany)* or *type(Berlin, germanCity)*, or both. There is no definite agreement as to which method is the right way to go, but there are advantages and disadvantages for each of them. If the entities in question can have certain properties that other entities cannot have, then it is useful to group them into a class. Practically speaking, this means that as soon as there is a relationship that has these entities as domain or range, the entities should become a class. For example, if we model Landkreise (the German equivalent of regions), then we can have *inLandkreis*$\subset$*germanCity*$\times$*Landkreis*. No city other than German cities can be in a Landkreis. Thus, it is useful to have the class *germanCity*. If, however, German cities behave just like all other cities in our KB, then a class for them is less useful. In this spirit, it makes sense to have a class for scientists (who have a graduation university), or digital cameras (which have a resolution), but less so for male scientists or Sony cameras.

However, if we want to express that an entity stands in a relationship with another entity, and if that other entity has itself many relationships, then it is useful to use a relational fact. This allows more precise querying. For example, German cities stand in a relationship with Germany. Germany is located in Europe, and it is one of the German speaking countries. Thus, by saying *locatedIn(Berlin, Germany)*, we can query for cities located in European countries and for German-speaking cities, without introducing a class for each of them. In this spirit, it makes sense to use the relational modeling for German cities or American actors, but much less so for, say, zoological categories such as mammals

[5] We can even say *type(class, class)*, i.e., *class* is an instance of *class*.

or reptiles. Sometimes neither choice may have strong arguments in favor, and sometimes both forms of modeling together may be useful.

2.4 Knowledge Bases

2.4.1 Completeness and Correctness

Knowledge bases model only a part of the world. In order to make this explicit, one imagines a complete knowledge base $\mathcal{K}^*$ that contains all entities and facts of the real world in the domain of interest. A given KB $\mathcal{K}$ is *correct*, if $\mathcal{K} \subseteq \mathcal{K}^*$. Usually, KBs aim to be correct. In real life, however, large KBs tend to contain also erroneous statements. YAGO, e.g., has an accuracy of 95%, meaning that 95% of its statements are in $\mathcal{K}^*$ (or, rather, in Wikipedia, which is used as an approximation of $\mathcal{K}^*$). This means that YAGO still contains hundreds of thousands of wrong statements. For most other KBs, the degree of correctness is not even known.

A knowledge base is *complete*, if $\mathcal{K}^* \subseteq \mathcal{K}$ (always staying within the domain of interest). The *closed world assumption* (CWA) is the assumption that the KB at hand is complete. Thus, the CWA says that any statement that is not in the KB is not in $\mathcal{K}^*$ either. In reality, however, KBs are hardly ever complete. Therefore, KBs typically operate under the *open world assumption* (OWA), which says that if a statement is not in the KB, then this statement can be either true or false in the real world.

KBs usually do not model negative information. They may say that Caltrain serves the city of San Francisco, but they will not say that this train does not serve the city of Moscow. While incompleteness tells us that some facts may be missing, the lack of negative information prevents us from specifying which facts are missing because they are false. This poses considerable problems, because the absence of a statement does not allow any conclusion about the real world [60].

2.5 The Semantic Web

The common exchange format for knowledge bases is RDF/RDFS [82]. It specifies a syntax for writing down statements with binary relations. Most notably, it prescribes URIs as identifiers, which means that entities can be identified in a globally unique way. To query such RDF knowledge bases, one can use the query language SPARQL [86]. SPARQL borrows its syntax from SQL, and allows the user to specify graph patterns, i.e., triples where some components are replaced by variables. For example, we can ask for the birth date of Elvis by saying "SELECT ?birthdate WHERE { ⟨Elvis⟩ ⟨bornOnDate⟩ ?birthdate }".

To define semantic constraints on the data, RDF is extended by OWL [85]. This language allows specifying constraints such as functions or disjointness of classes, as well as more complex axioms. The formal semantics of these axioms is given by Description Logics [3]. These logics distinguish facts about instances

from facts about classes and axioms. The facts about instances are called the *A-Box* ("Assertions"), and the class facts and axioms are called the *T-Box* ("Theory"). Sometimes, the term *ontology* is used to mean roughly the same as *T-Box*. Description Logics allow for automated reasoning on the data.

Many KBs are publicly available online. They form what is known as the *Semantic Web*. Some of these KBs talk about the same entities – with different identifiers. The Linked Open Data project [5] aims to establish links between equivalent identifiers, thus weaving all public KBs together into one giant knowledge graph.

2.6 Challenges in Knowledge Representation

Knowledge representation is a large field of research, which has received ample attention in the past, and which still harbors many open questions. Some of these open issues in the context of knowledge bases are the following.

Negative Information. For some applications (such as question answering or knowledge curation), it is important to know whether a statement is *not* true. As we have seen, KBs usually do not store negative information, and thus the mining of negative information is an active field of research. In some cases, axioms can help deducing negative information. For example, if some relation is a function, and if one object is present, then it follows that all other objects cannot be in the relation. In other cases, a variant of the closed world assumption can help [55].

Completeness. Today's KBs do not store the fact that they are complete in some domains. For example, if the KB knows all children of Barack Obama, then it would be helpful to store that the KB is complete on the children of Obama. Different techniques for storing completeness information have been devised (see [60] for a survey), and completeness can also be determined automatically to some degree [23,38,65], but these techniques are still in their infancy.

Correctness. Some KBs (e.g., NELL or YAGO) store a probability value with each statement, indicating the likelihood that the statement is correct. There is an ample corpus of scientific work on dealing with such probabilistic knowledge bases, but attaching probabilities to statements is currently not a universally adopted practice.

Provenance. Some KBs (e.g., Wikidata, NELL and YAGO) attach provenance information to their statements, i.e., the source where the statement was found, and the technique that was used to extract it. This information can be used to debug the KB, to justify the statements, or to optimize the construction process. Again, there is ample literature on dealing with provenance (see [4] for a survey of works in artificial intelligence, databases, and the Semantic Web) – although few KBs actually attach provenance information.

Time and Space. Some KBs (e.g., Wikidata and YAGO) store time and space information with their facts. Thus, they know where and when a fact happened. This is often achieved by giving each fact a fact identifier, and by making statements about that fact identifier. Other approaches abound [21,62,80,33,28]. They

include, e.g., the use of 5-ary facts, the introduction of a sub-property for each temporal statement, or the attachment of time labels.

Facts about Facts. We sometimes wish to store not just the time of a statement, but more facts about that statement. For example, we may want to store the correctness or provenance of a fact, but also the authority who vouches for the fact, access rights to the fact, or beliefs or hypotheses (as in "Fabian believes that Elvis is alive"). RDF provides a mechanism called *reification* for this purpose, but it is clumsy to use. Named Graphs [10] and annotations [76] have been proposed as alternatives. Different other alternatives are surveyed in [4]. Newer approaches attach attributes to statements [37,47].

Textual Extension. The textual source of the facts often contains additional subtleties that cannot be captured in triples. It can therefore be useful to add the textual information into the KB, as it is done, e.g., in [87].

NoRDF. For some information (such as complex events, narratives, or larger contexts), the representation as triples is no longer sufficient. We call this the realm of NoRDF knowledge (in analogy to NoSQL databases). For example, it is clumsy, if not impossible, to represent with binary relations the fact that Leonardo diCaprio was baptized "Leonardo" by his mother, because she visited a museum in Italy while she was still pregnant, and felt that the baby kicked while she saw a work of Leonardo DaVinci.

Commonsense Knowledge. Properties of everyday objects (e.g. that spiders have eight legs) and general concepts are of importance for text understanding, sentiment analysis, and object recognition in images and videos. This line of knowledge representation is well covered in classical works [62,41], and is lately also enjoying attention in the KB community [72,73].

Intensional Knowledge. Commonsense knowledge can also take the form of rules. For example, if a doctoral student is advised by a professor, then the university of graduation will be the employer of the professor. Again, this type of knowledge representation is well covered in classical works [62,41], and recent approaches have turned to using it for KBs [25,26,11]. This type of intensional knowledge is what we will now discuss in the next section.

3 Rule Mining

3.1 Rules

Once we have a knowledge base, it is interesting to look out for *patterns* in the data. For example, we could notice that if some person A is married to some person B, then usually B is also married to A (symmetry of marriage). Or we could notice that, if, in addition, A is the parent of some child, then B is usually also a parent of that child (although not always).

We usually write such rules using the syntax of first-order logic. For example, we would write the previous rules as:

$$marriedTo(x, y) \Rightarrow marriedTo(y, x)$$

$$marriedTo(x, y) \wedge hasChild(x, z) \Rightarrow hasChild(y, z)$$

Such rules have several applications: First, they can help us complete the KB. If, e.g., we know that Elvis Presley is married to Priscilla Presley, then we can deduce that Priscilla is also married to Elvis – if the fact was missing. Second, the rules can help us disambiguate entities and correct errors. For example, if Elvis has a child Lisa, and Priscilla has a different child Lisa, then our rule could help find out that the two Lisa's are actually a single entity. Finally, those frequent rules give us insight about our data, biases in the data, or biases in the real world. For example, we may find that European presidents are usually male or that Ancient Romans are usually dead. These two rules are examples of rules that have not just variables, but also entities:

$$type(x, AncientRoman) \Rightarrow dead(x)$$

We are now interested in discovering such rules automatically in the data. This process is called Rule Mining. Let us start with some definitions. The components of a rule are called atoms:

Definition 11 (Atom): *An atom is of the form $r(t_1, \ldots, t_n)$, where r is a relation of arity n (for KBs, usually $n = 2$) and $t_1, \ldots t_n$ are either variables or entities.*

In our example, $marriedTo(x, y)$ is an atom, as is $marriedTo(Elvis, y)$. We say that an atom is *instantiated*, if it contains at least one entity. We say that it is *grounded*, if it contains only entities and no variables. A *conjunction* is a set of atoms, which we write as $\boldsymbol{A} = A_1 \wedge ... \wedge A_n$. We are now ready to combine atoms to rules:

Definition 12 (Rule): *A Horn rule (rule, for short) is a formula of the form $\boldsymbol{B} \Rightarrow h$, where $\boldsymbol{B}$ is a conjunction of atoms, and h is an atom. $\boldsymbol{B}$ is called the body of the rule, and h its head.*

For example, $marriedTo(x, y) \Rightarrow marriedTo(y, x)$ is a rule. Such a rule is usually read as "If x is married to y, then y is married to x". In order to apply such a rule to specific entities, we need the notion of a *substitution*:

Definition 13 (Substitution): *A substitution is a function that maps variables to entities or to other variables.*

For example, a substitution σ can map $\sigma(x) = Elvis$ and $\sigma(y) = z$ – but not $\sigma(Elvis) = z$. A substitution can be generalized straightforwardly to atoms, sets of atoms, and rules: if $\sigma(x) = Elvis$, then $\sigma(marriedTo(Priscilla, x)) = marriedTo(Priscilla, Elvis)$. With this, an *instantiation of a rule* is a variant of the rule where all variables have been substituted by entities (so that all atoms are grounded). If we substitute $x = Elvis$ and $y = Priscilla$ in our example rule, we obtain the following instantiation:

$$marriedTo(Elvis, Priscilla) \Rightarrow marriedTo(Priscilla, Elvis)$$

Thus, an instantiation of a rule is an application of the rule to one concrete case. Let us now see what rules can predict:

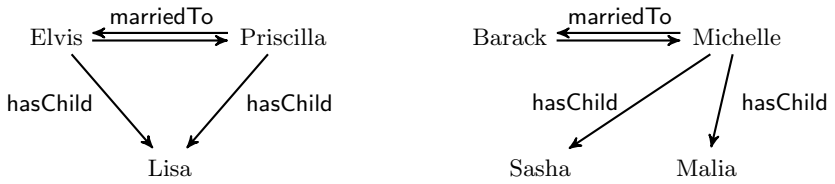

Fig. 1. Example KB

Definition 14 (Prediction of a rule): *The predictions P of a rule $\boldsymbol{B} \Rightarrow h$ in a KB $\mathcal{K}$ are the head atoms of all instantiations of the rule where the body atoms appear in $\mathcal{K}$. We write $\mathcal{K} \wedge (\boldsymbol{B} \Rightarrow h) \models P$. The predictions of a set of rules are the union of the predictions of each rule.*

For example, consider the KB in Figure 1. The predictions of the rule $marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$ are *hasChild(Priscilla, Lisa), hasChild(Elvis, Lisa), hasChild(Barack, Sasha), hasChild(Barack, Malia), hasChild(Michelle, Sasha), hasChild(Michelle, Malia)*. This is useful, because two of these facts are not yet in the KB.

Logic. From a logical perspective, all variables in a rule are implicitly universally quantified (over every entity defined in the KB). Thus, our example rule is more explicitly written as

$$\forall x, y, z : marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$$

It can be easily verified that such a rule is equivalent to the following disjunction:

$$\forall x, y, z : \neg marriedTo(x, y) \vee \neg hasChild(y, z) \vee hasChild(x, z)$$

While every Horn rule corresponds to a disjunction with universally quantified variables, not every such disjunction corresponds to a Horn rule. Only those disjunctions with exactly one positive atom correspond to Horn rules. In principle, we could mine arbitrary disjunctions, and not just those that correspond to Horn rules. We could even mine arbitrary first-order expressions, such as $\forall x : person(x) \Rightarrow \neg(underage(x) \wedge adult(x))$. For simplicity, we stay with Horn rules in what follows, and point out when an approach can be generalized to disjunctions or arbitrary formulae.

3.2 Rule Mining

3.2.1 Inductive Logic Programming

We now turn to mining rules automatically from a KB. This endeavor is based on *Inductive Reasoning*. To reason by induction is to expect that events that always appeared together in the past will always appear together in the future. For example, inductive reasoning could tell us: "All life forms we have

seen so far need water. Therefore, all life forms in general need water.". This is the fundamental principle of empirical science: the generalization of past experiences to a scientific theory. Of course, inductive reasoning can never deliver the logical certitude of deductive reasoning. This is illustrated by Bertrand Russel's analogy of the turkey [61]: The turkey is fed every day by its owner, and so it comes to believe that the owner will always feed the turkey – which is true only until Christmas day. The validity and limitations of modeling the reality using inductive reasoning are a debated topic in philosophy of science. For more perspectives on the philosophical discussions, we refer the reader to [29] and [31]. In the setting of KBs, inductive reasoning is formalized as *Inductive Logic Programming* [57,63,51]:

Definition 15 (Inductive Logic Programming): *Given a background knowledge $\mathcal{B}$ (in general, any first order logic expression; in our case: a KB), a set of positive example facts E^+, and a set of negative example facts E^-, Inductive Logic Programming (ILP) is the task of finding an hypothesis $\mathfrak{h}$ (in general, a set of first order logic expressions; in our case: a set of rules) such that $\forall e^+ \in E^+ : \mathcal{B} \wedge \mathfrak{h} \models e^+$ and $\forall e^- \in E^- : \mathcal{B} \wedge \mathfrak{h} \not\models e^-$.*

This means that the rules we seek have to predict all positive examples (they have to be *complete*), and they may not predict a negative example (they have to be *correct*). For example, consider again the KB from Figure 1 as background knowledge, and let the sets of examples be:

$$E^+ = \{\ \textit{isMarriedTo(Elvis, Priscilla), isMarriedTo(Priscilla, Elvis),}\\ \textit{isMarriedTo(Barack, Michelle), isMarriedTo(Michelle, Barack)}\}$$

$$E^- = \{\ \textit{isMarriedTo(Elvis, Michelle), isMarriedTo(Lisa, Barack),}\\ \textit{isMarriedTo(Sasha, Malia)}\}$$

Now consider the following hypothesis:

$$\mathfrak{h} = \{\textit{isMarriedTo}(x, y) \Rightarrow \textit{isMarriedTo}(y, x)\}$$

This hypothesis is complete, as every positive example is a prediction of the rule, and it is correct, as no negative example is predicted.

The attentive reader will notice that the difficulty is now to correctly determine the sets of positive and negative examples. In the ideal case the positive examples should contain any fact that is true in the real world and the negative examples contain any other fact. Thus, in a correct KB, every fact is a positive example.

Definition 16 (Rule Mining): *Given a KB, Rule Mining is the ILP task with the KB as background knowledge, and every single atom of the KB as a positive example.*

This means that the rule mining will find several rules, in order to explain all facts of the KB. Three problems remain: First, we have to define the set of negative examples (Section 3.2.2). Second, we have to define what types of rules we are interested in (Section 3.2.3). Finally, we have to adapt our mining to cases where the rule does not always hold (Section 3.2.4).

3.2.2 The Set of Negative Examples

Rule mining needs negative examples (also called *counter-examples*). The problem is that KBs usually do not contain negative information (Section 2.6). We can think of different ways to generate negative examples.

Closed World Assumption. The Closed World Assumption (CWA) says that any statement that is not in the KB is wrong (Section 2.4.1). Thus, under the Closed-World Assumption, any fact that is not in the KB can serve as a negative example. The problem is that these may be exactly the facts that we want to predict. In our example KB from Figure 1, we may want to learn the rule $marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$. For this rule, the fact *hasChild(Barack, Malia)* is a counter-example. However, this fact is exactly what we want to predict, and so it would be a counter-productive counter-example.

Open World Assumption. Under the Open-World Assumption (OWA), any fact that is not in the KB can be considered either a negative or a positive example (see again Section 2.4.1). Thus the OWA does not help in establishing counter-examples. Without counter-examples, we can learn any rule. For example, in our KB, the rule *type(x, person)* $\Rightarrow$ *marriedTo(x, Barack)* has a single positive example (for $x =$ *Michelle*), and no counter-examples under the Open World Assumption. Therefore, we could deduce that everyone is married to Barack.

Partial Completeness Assumption. Another strategy to generate negative examples is to assume that entities are complete for the relations they already have. For example, if we know that Michelle has the children Sasha and Malia, then we assume (much like Barack) that Michelle has no other children. If, in contrast, Barack does not have any children in the KB, then we do not conclude anything. This idea is called the Partial-Completeness Assumption (PCA) or the Local Closed World Assumption [25]. It holds trivially for functions (such as *hasBirthDate*), and usually [26] for relations with a high functionality (such as *hasNationality*). The rationale is that if the KB curators took the care to enter some objects for the relation, then they will most likely have entered all of them, if there are few of them. In contrast, the assumption does usually not hold for relations with low functionality (such as *starsInMovie*). Fortunately, relations usually have a higher functionality than their inverses (see Section 2.3.3). If that is not the case, we can apply the PCA to the object of the relation instead.

Random Examples. Another strategy to find counter-examples is to generate random statements [50]. Such random statements are unlikely to be correct, and can thus serve as counter-examples. This is one of the methods used by DL-Learner [30]. As we shall see in Section 4.3.1, it is not easy to generate helpful random counter-examples. If, e.g., we generate the random negative example *marriedTo(Barack,USA)*, then it is unlikely that a rule will try to predict this example. Thus, the example does not actually help in filtering out any rule. The challenge is hence to choose counter-examples that are false, but still reasonable. The authors of [55] describe a method to sample negative statements about

semantically connected entities by help of the PCA. We will also revisit the problem in the context of representation learning (Section 4.3.1).

3.2.3 The Language Bias

After solving the problem of negative examples, the next question is what kind of rules we should consider. This choice is called the *language bias*, because it restricts the "language" of the hypothesis. We have already limited ourselves to Horn Rules, and in practice we even restrict ourselves to connected and closed rules.

Definition 17 (Connected rules): *Two atoms are connected if they share a variable, and a rule is connected if every non-ground atom is transitively connected to one another.*

For example, the rule *presidentOf(x, America)* $\Rightarrow$ *hasChild(Elvis, y)* is not connected. It is an uninteresting and most likely wrong rule, because it makes a prediction about arbitrary y.

Definition 18 (Closed rules): *A rule is closed if every variable appears in at least two atoms.*

For example the rule *marriedTo(x, y)* $\wedge$ *worksAt(x, z)* $\Rightarrow$ *marriedTo(y, x)* is not closed. It has a "dangling edge" that imposes that x works somewhere. While such rules are perfectly valid, they are usually less interesting than the more general rule without the dangling edge.

Finally, one usually imposes a limit on the number of atoms in the rule. Rules with too many atoms tend to be very convoluted [26]. That said, mining rules without such restrictions is an interesting field of research, and we will come back to it in Section 3.5.

3.2.4 Support and Confidence

One problem with classical ILP approaches is that they will find rules that apply to very few entities, such as *marriedTo(x, Elvis)* $\Rightarrow$ *hasChild(x, Lisa)*. To avoid this type of rules, we define the *support* of a rule:

Definition 19 (Support): *The support of a rule in a KB is the number of positive examples predicted by the rule.*

Usually, we are interested only in rules that have a support higher than a given threshold (say, 100). Alternatively, we can define a relative version of support, the *head coverage* [25], which is the number of positive examples predicted by the rule divided by the number of all positive examples with the same relation. Another problem with classical ILP approaches is that they will not find rules if there is a single counter-example. To mitigate this problem, we define the *confidence*:

Definition 20 (Confidence): *The confidence of a rule is the number of positive examples predicted by the rule (i.e., the support of the rule), divided by the number of examples predicted by the rule.*

This notion depends on how we choose our negative examples. For instance, under the CWA, the rule $marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$ has a confidence of 4/6 in Figure 1. We call this value the *standard confidence*. Under the PCA, in contrast, the confidence for the example rule is 4/4. We call this value the *PCA confidence*. While the standard confidence tends to "punish" rules that predict many unknown statements, the PCA confidence will permit more such rules. We present in Appendix A the exact mathematical formula of these measures.

In general, the support of a rule quantifies its completeness, and the confidence quantifies its correctness. A rule with low support and high confidence indicates a conservative hypothesis and may be overfitting, i.e. it will not generalize to new positive examples. A rule with high support and low confidence, in contrast, indicates a more general hypothesis and may be overgeneralizing, i.e., it does not generalize to new negative examples. In order to avoid these effects we are looking for a trade-off between support and confidence.

Definition 21 (Frequent Rule Mining): *Given a KB $\mathcal{K}$, a set of positive examples (usually $\mathcal{K}$), a set of negative examples (usually according to an assumption above) and a language of rules, Frequent rule mining is the task of finding all rules in the language with a support and a level of confidence superior to given thresholds.*

3.3 Rule Mining Approaches

Using substitutions (see Definition 13), we can define a syntactical order on rules:

Definition 22 (Rule order): *A rule $R \equiv (\boldsymbol{B} \Rightarrow h)$ subsumes a rule $R' \equiv (\boldsymbol{B}' \Rightarrow h')$, or R is "more general than" R', or R' "is more specific than" R, if there is a substitution σ such that $\sigma(\boldsymbol{B}) \subseteq \boldsymbol{B}'$ and $\sigma(h) = h'$. If both rules subsume each other, the rules are called equivalent.*

For example, consider the following rules:

$$\begin{cases} hasChild(x, y) \Rightarrow hasChild(z, y) & (R_0) \\ hasChild(Elvis, y) \Rightarrow hasChild(Priscilla, y) & (R_1) \\ hasChild(x, y) \Rightarrow hasChild(z, Lisa) & (R_2) \\ hasChild(x, y) \wedge marriedTo(x, z) \Rightarrow hasChild(z, y) & (R_3) \\ marriedTo(v_1, v_2) \wedge hasChild(v_1, v_3) \Rightarrow hasChild(v_2, v_3) & (R_4) \\ hasChild(x, y) \wedge marriedTo(z, x) \Rightarrow hasChild(z, y) & (R_5) \end{cases}$$

The rule R_0 is more general than the rule R_1, because we can rewrite the variables x and z to *Elvis* and *Priscilla* respectively. However R_0 and R_2 are incomparable as we cannot choose to bind only one y and not the other in R_0. The rules R_3, R_4 and R_5 are more specific than R_0. Finally R_3 is equivalent to R_4 but not to R_5.

Proposition 23 (Prediction inclusion): *If a rule R is more general than a rule R', then the predictions of R' on a KB are a subset of the predictions of R. As a corollary, R' cannot have a higher support than R.*

This observation gives us two families of rule mining algorithms: top-down rule mining starts from very general rules and specializes them until they become too specific (i.e., no longer meet the support threshold). Bottom-up rule mining, in contrast, starts from multiple ground rules and generalizes them until the rules become too general (i.e., too many negative examples are predicted).

3.3.1 Top-Down Rule Mining

The concept of specializing a general rule to more specific rules can be traced back to [63] in the context of an exact ILP task (under the CWA). Such approaches usually employ a *refinement operator*, i.e. a function that takes a rule (or a set of rules) as input and returns a set of more specific rules. For example, a refinement operator could take the rule $hasChild(y, z) \Rightarrow hasChild(x, z)$ and produce the more specific rule $marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$. This process is iterated, and creates a set of rules that we call the *search space* of the rule mining algorithm. On the one hand, the search space should contain every rule of a given rule mining task, so as to be complete. On the other hand, the smaller the search space is, the more efficient the algorithm is.

Usually, the search space is *pruned*, i.e., less promising areas of the search space are cut away. For example, if a rule does not have enough support, then any refinement of it will have even lower support (Proposition 23). Hence, there is no use refining this rule.

AMIE. AMIE [25] is a top-down rule mining algorithm that aims to mine any connected rule composed of binary atoms for a given support and minimum level of confidence in a KB. AMIE starts with rules composed of only a head atom for all possible head atoms (e.g., $\Rightarrow$ *marriedTo(x, y)*). It uses three refinement operators, each of which adds a new atom to the body of the rule.

The first refinement operator, `addDanglingAtom`, adds an atom composed of a variable already present in the input rule and a new variable.

$$\text{Some refinements of: } \Rightarrow hasChild(z, y)\ (R_h)$$
$$\text{are: } \begin{cases} hasChild(x, y) \Rightarrow hasChild(z, y)\ (R_0) \\ marriedTo(x, z) \Rightarrow hasChild(z, y)\ (R_a) \\ marriedTo(z, x) \Rightarrow hasChild(z, y)\ (R_b) \end{cases}$$

The second operator, `addInstantiatedAtom`, adds an atom composed of a variable already present in the input rule and an entity of the KB.

$$\text{Some refinements of: } \Rightarrow hasChild(Priscilla, y)\ (R'_h)$$
$$\text{are: } \begin{cases} hasChild(Elvis, y) \Rightarrow hasChild(Priscilla, y)\ (R_1) \\ hasChild(Priscilla, y) \Rightarrow hasChild(Priscilla, y)\ (R_\top) \\ marriedTo(Barack, y) \Rightarrow hasChild(Priscilla, y)\ (R_\bot) \end{cases}$$

The final refinement operator, `addClosingAtom`, adds an atom composed of two variables already present in the input rule.

$$\text{Some refinements of:} \quad marriedTo(x,z) \Rightarrow hasChild(z,y)\ (R_a)$$
$$\text{are:} \quad \begin{cases} hasChild(x,y) \wedge marriedTo(x,z) \Rightarrow hasChild(z,y)\ (R_3) \\ marriedTo(z,y) \wedge marriedTo(x,z) \Rightarrow hasChild(z,y)\ (R_\alpha) \\ marriedTo(x,z) \wedge marriedTo(x,z) \Rightarrow hasChild(z,y)\ (R_a^2) \end{cases}$$

As every new atom added by an operator contains at least a variable present in the input rule, the generated rules are connected. The last operator is used to close the rules (for example R_3), although it may have to be applied several times to actually produce a closed rule (cf. Rules R_α or R_a^2).

The AMIE algorithm works on a queue of rules. Initially, the queue contains one rule of a single head atom for each relation in the KB. At each step, AMIE dequeues the first rule, and applies all three refinement operators. The resulting rules are then pruned: First, any rule with low support (such as $R_\perp$) is discarded. Second, different refinements may generate equivalent rules (using the closing operator on R_0 or R_a, e.g., generates among others two equivalent "versions" of R_3). AMIE prunes out these equivalent versions. AMIE+ [26] also detects equivalent atoms as in $R_\top$ or R_a^2 and rewrites or removes those rules. There are a number of other, more sophisticated pruning strategies that estimate bounds on the support or confidence. The rules that survive this pruning process are added to the queue. If one of the rules is a closed rule with a high confidence, it is also output as a result. In this way, AMIE enumerates the entire search space.

The top-down rule mining method is generic, but its result depends on the initial rules and on the refinement operators. The operators directly impact the language of rules we can mine (see Section 3.2.3) and the performance of the method. We can change the refinement operators to mine a completely different language of rules. For example, if we don't use the `addInstantiatedAtom` operator, we restrict our search to any rule without instantiated atoms, which also drastically reduce the size of the search space[6].

Apriori Algorithm. There is an analogy between top-down rule mining and the Apriori algorithm [1]. The Apriori algorithm considers a set of transactions (sales, products bought in a supermarket), each of which is a set of items (items bought together, in the supermarket analogy). The goal of the Apriori algorithm is to find a set of items that are frequently bought together.

These are frequent patterns of the form $\boldsymbol{P} \equiv I_1(x) \wedge \cdots \wedge I_n(x)$, where $I(t)$ is in our transaction database if the item I has been bought in the transaction t. Written as the set (called an "itemset") $\boldsymbol{P} \equiv \{I_1, \ldots, I_n\}$, any subset of $\boldsymbol{P}$ forms a "more general" itemset than $\boldsymbol{P}$, which is at least as frequent as P. The Apriori algorithm uses the dual view of the support pruning strategy: Necessarily, all patterns more general than $\boldsymbol{P}$ must be frequent for $\boldsymbol{P}$ to be frequent[7]. The

[6] Let $|\mathcal{K}|$ be the number of facts and $|r(\mathcal{K})|$ the number of relations in a KB $\mathcal{K}$. Let d be the maximal length of a rule. The size of the search space is reduced from $O(|\mathcal{K}|^d)$ to $O(|r(\mathcal{K})|^d)$ when we remove the `addInstantiatedAtom` operator.

[7] instead of: if a rule is not frequent, none of its refinements can be frequent

refinement operator of the Apriori algorithm takes as input all frequent itemsets of size n and generate all itemsets of size $n+1$ such that any subset of size n is a frequent itemset. Thus, Apriori can be seen as a top-down rule mining algorithm over a very specific language where all atoms are unary predicates.

The WARMR algorithm [13], an ancestor of AMIE, was the first to adapt the Apriori algorithm to rule mining over multiple (multidimensional) relations.

3.3.2 Bottom-Up Rule Mining

As the opposite of a refinement operator, one can define a generalization operator that considers several specific rules, and outputs a rule that is more general than the input rules. For this purpose, we will make use of the observation from Section 3.1 that a rule $b_1 \wedge ... \wedge b_n \Rightarrow h$ is equivalent to the disjunction $\neg b_1 \vee \cdots \vee \neg b_n \vee h$. The disjunction, in turn, can be written as a set $\{\neg b_1, \ldots, \neg b_n, h\}$ – which we call a *clause*. For example, the rule $marriedTo(x, y) \wedge hasChild(y, z) \Rightarrow hasChild(x, z)$ can be written as the clause $\{\neg marriedTo(x, y), \neg hasChild(y, z), hasChild(x, z)\}$. Bottom-up rule mining approaches work on clauses. Thus, they work on universally quantified disjunctions – which are more general than Horn rules. Two clauses can be combined to a more general clause using the "least general generalization" operator [57]:

Definition 24 (Least general generalization): *The least general generalization (lgg) of two clauses is computed in the following recursive manner:*

- *The lgg of two terms (i.e., either entities or variables) t and t' is t if $t = t'$ and a new variable $x_{t/t'}$ otherwise.*
- *The lgg of two negated atoms is the negation of their lgg.*
- *The lgg of $r(t_1, \ldots, t_n)$ and $r(t'_1, \ldots, t'_n)$ is $r(lgg(t_1, t'_1), \ldots, lgg(t_n, t'_n))$.*
- *The lgg of a negated atom with a positive atom is undefined.*
- *Likewise, the lgg of two atoms with different relations is undefined.*
- *The lgg of two clauses R and R' is the set of defined pair-wise generalizations:*

$$lgg(R, R') = \{lgg(l_i, l'_j) \ : \ l_i \in R, \ l'_j \in R', \ \text{and } lgg(l_i, l'_j) \text{ is defined}\}$$

For example, let us consider the following two rules:

$$\begin{aligned} &hasChild(Michelle, Sasha) \wedge marriedTo(Michelle, Barack) \\ &\qquad \Rightarrow hasChild(Barack, Sasha) \quad (R) \\ &hasChild(Michelle, Malia) \wedge marriedTo(Michelle, x) \\ &\qquad \Rightarrow hasChild(x, Malia) \qquad (R') \end{aligned}$$

In the form of clauses, these are

$$\begin{aligned} &\{\neg hasChild(Michelle, Sasha), \neg marriedTo(Michelle, Barack), \\ &\qquad hasChild(Barack, Sasha)\} \quad (R) \\ &\{\neg hasChild(Michelle, Malia), \neg marriedTo(Michelle, x), \\ &\qquad hasChild(x, Malia)\} \qquad (R') \end{aligned}$$

Now, we have to compute the lgg of every atom of the first clause with every atom of the second clause. As it turns out, there are only 3 pairs where the lgg is defined:

$$\begin{aligned}
&lgg(\neg hasChild(Michelle, Sasha), \neg hasChild(Michelle, Malia)) \\
&\quad = \neg lgg(hasChild(Michelle, Sasha), hasChild(Michelle, Malia)) \\
&\quad = \neg hasChild(lgg(Michelle, Michelle), lgg(Sasha, Malia)) \\
&\quad = \neg hasChild(Michelle, x_{Sasha/Malia})
\end{aligned}$$

$$\begin{aligned}
&lgg(\neg marriedTo(Michelle, Barack), \neg marriedTo(Michelle, x)) \\
&\quad = \neg marriedTo(Michelle, x_{Barack/x})
\end{aligned}$$

$$\begin{aligned}
&lgg(hasChild(Barack, Sasha), hasChild(x, Malia)) \\
&\quad = hasChild(x_{Barack/x}, x_{Sasha/Malia})
\end{aligned}$$

This yields the clause

$$\{\neg hasChild(Michelle, x_{Sasha/Malia}),\ \neg marriedTo(Michelle, x_{Barack/x}),\ hasChild(x_{Barack/x}, x_{Sasha/Malia})\}$$

This clause is equivalent to the rule

$$hasChild(Michelle, x) \wedge\ marriedTo(Michelle, y) \Rightarrow hasChild(x, y)$$

Note that the generalization of two different terms in an atom should result in the same variable as the generalization of these terms in another atom. In our example, we obtain only two new variables $x_{Sasha/Malia}$ and $x_{Barack/x}$. In this way, we have generalized the two initial rules to a more general rule. This can be done systematically with an algorithm called *GOLEM*.

GOLEM. The GOLEM/RLGG algorithm [51] creates, for each positive example $e \in E^+$, the rule $\mathcal{B} \Rightarrow e$, where $\mathcal{B}$ is the background knowledge. In our case, $\mathcal{B}$ is the entire KB, and so a very long conjunction of facts. The algorithm will then generalize these rules to shorter rules. More precisely, the *relative lgg* (rlgg) of a tuple of ground atoms $(e_1, \ldots, e_n)$ is the rule obtained by computing the lgg of the rules $\mathcal{B} \Rightarrow e_1$, ..., $\mathcal{B} \Rightarrow e_n$. We will call a rlgg *valid* if it is defined and does not predict any negative example.

The algorithm starts with a randomly sampled pair of positive examples (e_1, e_2) and selects the pair for which the rlgg is valid and predicts ("covers") the most positive examples. It will then greedily add positive examples, chosen among a sample of "not yet covered positive examples", to the tuple – as long as the corresponding rlgg is valid and covers more positive examples. The resulting rule will still contain ground atoms from B. These are removed, and the rule is output. Then the process starts over to find other rules for uncovered positive examples.

Progol and others. More recent ILP algorithms such as Progol [49], HAIL [59], Imparo [36] and others [88,34] use inverse entailment to compute the hypothesis

more efficiently. This idea is based on the observation that a hypothesis $\mathfrak{h}$ that satisfies $\mathcal{B} \wedge \mathfrak{h} \models E^+$ should equivalently satisfy $\mathcal{B} \wedge \neg E^+ \models \neg \mathfrak{h}$ (by logical contraposition). The algorithms work in two steps: they will first construct an intermediate theory F such that $\mathcal{B} \wedge \neg E^+ \models F$ and then generalize its negation $\neg F$ to the hypothesis $\mathfrak{h}$ using inverse entailment.

3.4 Related Approaches

This article cannot give a full review of the field of rule mining. However, it is interesting to point out some other approaches in other domains that deal with similar problems:

OWL. OWL is a Description logic language designed to define rules and constraints on the KB. For example, an OWL rule can say that every person must have a single birth date. Such constraints are usually defined upfront by domain experts and KB architects when they design the KB. They are then used for automatic reasoning and consistency checks. Thus, constraints *prescribe* the shape of the data, while the rules we mine *describe* the shape of the data. In other words, constraints are used deductively – instead of being found inductively. As such, they should suffer no exception. However, rule mining can provide candidate constraints to experts when they want to augment their theory [30].

Probabilistic ILP. As an extension of the classic ILP problem, Probabilistic ILP [12] aims to find the logical hypothesis $\mathfrak{h}$ that, given probabilistic background knowledge, maximizes the probability to observe a positive example, and minimizes the probability to observe a negative example. In our case, it would require a probabilistic model of the real world. Such models have been proposed for some specific use cases [38,90], but they remain an ongoing subject of research.

Graph Mining and Subgraph Discovery. Subgraph discovery is a well studied problem in the graph database community (see [27] Part 8 for a quick overview). Given a set of graphs, the task is to mine a subgraph that appears in most of them. Rule mining, in contrast, is looking for patterns that are frequent in the same graph. This difference may look marginal, but the state-of-the-art algorithms are very different and further work would be needed to determine how to translate one problem to the other.

Link Prediction. Rules can be used for link prediction, i.e., to predict whether a relation links two entities. This task can also be seen as a classification problem ([27] Part 7): given two entities, predict whether there is a relation between them. A notable work that unites both views [39] uses every conjunction of atoms (a possible body for a rule, which they call a "path") as a feature dimension for this classification problem. We will extensively present a more recent approach to this problem in Section 4.

3.5 Challenges in Rule Mining

Today, Horn rules can be mined efficiently on large KBs [68]. However, many challenges remain.

Negation. KBs usually do not contain negative information. Therefore, it is difficult to mine rules that have a negated atom in the body or in the head, such as $marriedTo(x, y) \wedge y \neq z \Rightarrow \neg marriedTo(x, z)$. Newer approaches use a variant of the PCA [55], class information [22], or new types of confidence measures [16].

External Information. Since KBs are both incomplete and lacking negative information, it is tempting to add in data from other sources to guide the rule mining. One can e.g., add in information about cardinalities [56], or embeddings computed on text [32].

Numerical Rules. We can imagine rules that detect numerical correlations (say, between the population of a city and the size of its area), bounds on numerical values (say, on the death year of Ancient Romans), or even complex numerical formulae (say, that the ratio of inhabitants of the capital is larger in city states) [24,48,18].

Scaling. While today's algorithms work well on large KBs, they do less well once we consider rules that do not just contain variables, but also entities. Furthermore, KBs grow larger and larger. Thus, scalability remains a permanent problem. It can be addressed, e.g., by smarter pruning strategies, parallelization, or by precomputing cliques in the graph of the KB.

4 Representation Learning

After having discussed symbolic representations of entities and rules, we now turn to subsymbolic representations. In this setting, entities are represented not as identifiers with relations, but as numerical vectors. Facts are predicted not by logical rules, but by computing a score for fact candidates.

4.1 Embedding

The simplest way to represent an entity as a vector is by a *one-hot encoding*:

Definition 25 (One-hot encoding): *Given an ordered set of objects $S = \{o_1, ...o_n\}$, the one-hot encoding of the object o_i is the vector $h(o_i) \in \mathbb{R}^n$ that contains only zeros, and a single one at position i.*

For example, in our KB in Figure 1, we have 7 entities. We can easily order them, say alphabetically. Then, *Barack* is the first entity, and hence his one-hot encoding is $\begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}^T$ (where the T just means that we wrote the vector horizontally instead of vertically). Such representations are not particularly useful, because they do not reflect any semantic similarity: The vector of Barack has the same distance to the vector of Michelle as to the vector of Lisa.

Definition 26 (Embedding): *An n-dimensional embedding for a group of objects (e.g. words, entities) is an injective function that maps each object to a vector in $\mathbb{R}^n$, so that the intrinsic relations between the objects are maintained.*

For example, we want to embed the entity *Barack* in such a way that his vector is close to the vector of Michelle, or maybe to the vectors of other politicians. Embeddings can also be used for words of natural language. In that case, the goal is to find an embedding where the vectors of related words are close. For example, the vector of the word "queen" and the vector of "king" should be close to each other. An ideal word embedding would even permit arithmetic relations such as *v(king) - v(man) + v(woman) = v(queen)* (where $v(\cdot)$ is the embedding function). This means that removing the vector for "man" from "king", and adding "woman" should yield the vector for "queen". Vectors are usually denoted with bold letters.

Embeddings are interesting mainly for two reasons: first they lower the dimensions of object representations. For example, there may be millions of entities in a KB, but they can be embedded in vectors of a few hundred dimensions. It is typically easier for down-stream tasks to deal with vectors than with sets of this size. Second, the structure of the embedding space makes it possible to compare objects that were incomparable in their original forms (e.g. it is now easy to define a distance between entities or between words by measuring the euclidean distance between their embeddings).

There are many ways to compute embeddings. A very common one is to use neural networks, as we shall discuss next.

4.2 Neural networks

4.2.1 Architecture

We start our introduction to neural networks with the notion of an *activation function*:

Definition 27 (Activation Function): *An activation function is a non-linear real function.*

Typical examples of activation functions are the hyperbolic tangent, the sigmoid function $\sigma : x \mapsto (1 + e^{-x})^{-1}$ and the rectified linear unit function ReLU: $x \mapsto \max(0, x)$. Although these functions are defined on a single real value, they are usually applied point-wise on a vector of real values. For example, we write $\sigma(\langle x_1, ..., x_n \rangle)$ to mean $\langle \sigma(x_1), ..., \sigma(x_n) \rangle$. Neural networks consist of several layers with such activation functions:

Definition 28 (Layer): *In the context of neural networks, a layer is a function* $\ell \colon \mathbb{R}^i \to \mathbb{R}^j$ *that is either linear or the composition of a linear function and an activation function (i and j are non-zero naturals).*

Thus, a layer is a function that takes as input a vector $v \in \mathbb{R}^i$, and does two things with it. First, it applies a linear function to v, i.e., it multiplies v with a matrix $W \in \mathbb{R}^i \times \mathbb{R}^j$ (the *weight matrix*). This yields $W \cdot v \in \mathbb{R}^j$. Then, it applies the activation function to this vector, which yields again a vector of size j. We can now compose the layers to neural networks:

Definition 29 (Neural Network): *In its simplest form (that of a Multilayer perceptron), a neural network is a function* $g\colon \mathbb{R}^n \to \mathbb{R}^p$*, such that* g *is a composition of layers. The parameters* p*,* n *and the intermediate dimensions of the layers are non-zero naturals.*

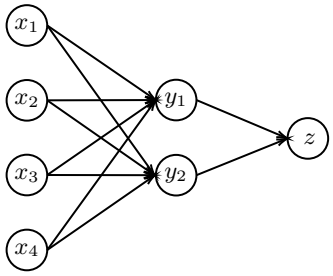

Fig. 2. Example of a one-hidden-layer network.

Figure 2 shows a one-hidden-layer network that takes as input vectors $x \in \mathbb{R}^4$ and outputs real values $z \in \mathbb{R}^1$. The function g of the network can be decomposed as $g = \ell_2 \circ \ell_1$ where $\ell_1\colon \mathbb{R}^4 \to \mathbb{R}^2$ is the hidden layer and $\ell_2\colon \mathbb{R}^2 \to \mathbb{R}^1$ is the output layer. Let us now see how such a network computes its output. Let us assume that the weight matrix of the first layer is A, that the weight matrix of the second layer is B and that the input is x:

$$x = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, A = \begin{pmatrix} 0.4 & 0.5 & -0.3 & 0.1 \\ 0.8 & -0.6 & 0.4 & 0.2 \end{pmatrix}, B = \begin{pmatrix} 0.5 & -0.6 \end{pmatrix} \tag{1}$$

If both layers use the sigmoid activation function σ, we can compute the result of the first layer as $y = \ell_1(x) = \sigma(A \cdot x)$, and the result of the second layer (and thus of the entire network) as $z = \ell_2(y) = \sigma(B \cdot y)$:

$$y = \sigma\left(\begin{pmatrix} 0.2 & 0.5 & -0.3 & 0.1 \\ 0.8 & -0.6 & 0.4 & 0.2 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}\right) = \sigma\left(\begin{pmatrix} 0.5 \\ -0.6 \end{pmatrix}\right) = \begin{pmatrix} \sigma(0.5) \\ \sigma(-0.6) \end{pmatrix} \approx \begin{pmatrix} 0.62 \\ 0.35 \end{pmatrix} \tag{2}$$

$$z = \sigma\left(\begin{pmatrix} 0.5 & -0.7 \end{pmatrix} \cdot \begin{pmatrix} 0.62 \\ 0.35 \end{pmatrix}\right) = \sigma\left(\begin{pmatrix} 0.063 \end{pmatrix}\right) = \begin{pmatrix} \sigma(0.063) \end{pmatrix} \approx \begin{pmatrix} 0.51 \end{pmatrix} \tag{3}$$

Figure 3 shows this computation graphically. The function computed by the network is $g = \ell_2 \circ \ell_1 = \sigma \circ b \circ \sigma \circ a$, where a and b are linear functions defined by the matrices A and B.

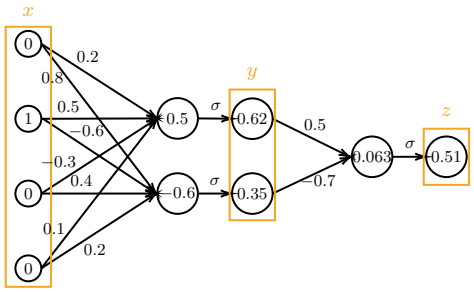

Fig. 3. Example of a one-hidden-layer network with computations.

4.2.2 Training

We want the neural network to compute an embedding of KB entities and relations. For this, we first have to understand how neural networks can perform supervised learning tasks. In supervised learning, we are interested in approximating some function $f : \mathbb{R}^p \mapsto \mathbb{R}^k$. We do not know f. We only know some datapoints of f: $\{(x_i, \alpha_i) \in \mathbb{R}^p \times \mathbb{R}^k | i = 0, \ldots, n\}$, with $f(x_i) = \alpha_i$ for any i. The goal is to find the best estimation of f. With our notations, we would like to find the neural network whose function $\hat{f}$ approximates f.

We first decide on an architecture of the neural network. We already know how many input nodes it has (namely p), and how many output nodes it has (namely k). We just have to decide how many hidden layers it has (typically a single one for simple tasks), and what the activation functions are (the sigmoid function is a popular choice). Then, we initialize the weight matrices of the layers randomly. Training the network model now means adapting the weight matrices for each layer so that $\hat{f}(x_i) = \alpha_i$ for all datapoints (or at least for as many as possible). This is achieved using gradient descent: for each sample x_i of the dataset, a loss is computed (comparing the output $\hat{f}(x_i)$ to the true value α_i) and the weights are updated in the opposite direction of the gradient of the sum of the losses with respect to the weights of the network.

Interestingly, neural networks can be trained for many functions between vectors. It has been proven that the range of functions neural networks can approximate is very large and grows very rapidly with the depth (number of layers) of the network [74]. This is a big strength of these models.

4.2.3 Embeddings

Let us now see how we can use neural networks to compute an m-dimensional embedding function for a set of objects $S = \{o_1, ..., o_n\}$. The input to the network will be the one-hot encoding of the object, i.e., we need n input nodes. In the ideal case, the neural network would directly output a vector of size m (the embedding). Then, however, we would not know how to train the network,

because we have no given embeddings to compare to. Therefore, we use a trick: We do not let the network compute the embedding directly, but a function whose output we know. For example, suppose the objects are people, and suppose we know the gender of the people (1 for female, 0 for male, or anything in between). We build a network with a single hidden layer of size m and an output layer of size 1 (because we want to predict a single value, the gender). Figure 2 shows such a network for $n = 4$ and $m = 2$.

Then we train the network to predict the gender of each person (i.e., we find the weights so that $\hat{f}(o_i) = 1$ if o_i is female, etc.). In our example from Figure 3, we have trained the network so that the object o_2 has a gender value of 0.51. Interestingly, after training, the first layer of the network is often a very good embedding function. In our example, the embedding of $x = \begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}^T$ would be $y = \begin{pmatrix}0.62 & 0.35\end{pmatrix}^T$.

Why is that a good choice? We first observe that the embedding function has the right dimensions: it maps a one-hot encoded vector of dimension n to a vector of dimension m, as desired ($v : \mathbb{R}^n \mapsto \mathbb{R}^m$). Then, we observe that the second layer (which computes the gender) bases its computation purely on the outputs of the first layer (the sigmoid of the embedding). Therefore, the output of the hidden layer provided enough information to reconstitute the gender, i.e., our embedding maintains the crucial information. Selecting a hidden layer as embedding comes down to dividing the network in two parts: the first layer computes features (the components of y) that should capture the information relevant for the application it is trained on; the second layer computes the value $\hat{f}(x)$ using only those extracted features contained in y. This division makes it intuitive that if the training task is well-chosen, the computed features should capture interesting aspects of the data and constitute a good embedding candidate. Note that even if we are interested only in y it is still necessary to train the entire network as we can only evaluate the performance of the embedding by comparing $\hat{f}$ to f.

The method that creates an embedding (in our case: a neural network) is often called a *model*. We will now see how to create models for facts in KBs.

4.3 Knowledge Base Embeddings

If we want to embed a KB, we can either embed entities, relations, or facts. Most models in the literature embed entities and relations together. These models take as input a fact of a subject s, a relation r and an object o as one-hot encoded vectors, which are concatenated together to one long vector with three 1s. Let's take as example the knowledge base from Figure 1. This KB has 7 entities (*Barack, Michelle, Sasha, Malia, Elvis, Priscilla Lisa*) and 2 relations (*marriedTo, hasChild*). To feed the fact *marriedTo(Barack, Michelle)* into the model, we create the one-hot encoded vectors and concatenate them to one long vector: $(\begin{pmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0\end{pmatrix}, \begin{pmatrix}1 & 0\end{pmatrix}, \begin{pmatrix}0 & 1 & 0 & 0 & 0 & 0 & 0\end{pmatrix})^T$. The output of the model will be a scoring function:

Definition 30 (Scoring Function): *In the context of knowledge base embeddings, a scoring function maps a fact $r(s, o)$ to a real-valued score.*

The score of a fact is an estimate of the true theoretical and unknown function deciding whether the fact is true. Obviously, the score should be high for the facts in a correct KB (Section 2.4.1). In certain probabilistic contexts, the score can be interpreted as the likelihood of the fact to be true. We denote the scoring function of the fact $r(s, o)$ by $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o})$.

As for the embeddings we already saw, models are divided in two parts: the first one which links the one-hot encoded vectors to the embeddings $\boldsymbol{r}, \boldsymbol{s}$ and $\boldsymbol{o}$ and the second part which computes $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o})$. We now have to train the model to predict whether an input fact is true (has a high score) or not (has a low score). Once the model is trained, we will be able to read off the embeddings from the weight matrix of one of the hidden layers.

Let us now see where we can find training data. For the true facts, the KB obviously provides lots of datapoints. However, if we just train the model on positive facts, it will just learn to always predict a high score. This is the same problem we already saw in Section 3.2.2. Therefore, we also need to provide negative facts:

Definition 31 (Negative fact): *Given a fact $r(s, o)$ from a KB, a negative fact is a statement $r(s', o')$ that is not in the KB.*

The process of generating negative fact is called *negative sampling* and is detailed in Section 4.3.1. For now, let us just assume that we have such negative facts at our disposal. To train the network, we use a loss function:

Definition 32 (Loss function): *A loss function ℓ is a function from $\mathbb{R}^2$ to $\mathbb{R}$.*

We will apply the loss function to the score $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o})$ that the network computed for a true fact $r(s, o)$ and the score $f_{\boldsymbol{r}}(\boldsymbol{s'}, \boldsymbol{o'})$ that the network computed for a negative fact $r(s', o')$. Naturally, the two scores should be very different: the first score should be high, and the second one should be low. If the two scores are close, the network is not trained well. Therefore, the loss function $\ell(x_1, x_2)$ should be larger the closer x_1 and x_2 are. The logistic loss or the margin loss are usual examples. They are defined respectively in Equation 4 and Equation 5, where γ is a parameter, and $\eta_z = 1$ if z is the score for a positive example and $\eta_z = -1$ if z is the score for a negative example:

$$(x, y) \mapsto \log(1 + \exp(-\eta_x \times x)) + \log(1 + \exp(-\eta_y \times y)) \tag{4}$$

$$(x, y) \mapsto \max(0, \gamma + \eta_x \times x + \eta_y \times y) \tag{5}$$

Definition 33 (Training): *Training a knowledge base embedding model is finding the best parameters of the model (and then the best embeddings) so that the scoring function $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o})$ is maximized for true facts and minimized for negative ones.*

Training is done by minimizing the sum of loss functions by gradient descent over a training set of facts. The sum is usually computed as follows, where $r(s, o)$

is a fact, $r(s', o')$ is a negative fact generated from $r(s, o)$ (c.f. Section 4.3.1), and ℓ a loss function:

$$\mathcal{L} = \sum_{(s,r,o)\in\mathcal{K}} \ell\left(f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o}), f_{\boldsymbol{r}}(\boldsymbol{s'}, \boldsymbol{o'})\right) \tag{6}$$

4.3.1 Negative sampling

Let us now see how we can generate the negative facts for our model. Feeding negative samples to the model is vital during training. If the model was only trained on true samples, then it could minimize any loss by trivially returning a large score for any fact it is fed with. This is the same problem that we already saw in Section 3.2.2, and in principle the same considerations and methods apply here as well. In the context of knowledge base embeddings, the generation of negative facts is usually done by *negative sampling*. Negative sampling is the process of corrupting a true fact's subject or object in order to create a wrong statement. This is very related to the Partial Completeness Assumption that we already saw in Section 3.2.2: If we have a fact *hasChild(Michelle, Sasha)*, then any variant of this fact with a different object is assumed to be a wrong statement – unless it is already in the KB. For example, we would generate the negative facts *hasChild(Michelle, Elvis)* and *hasChild(Michelle, Barack)*.

It has been observed that the quality of the resulting embedding highly depends on the quality of the negative sampling. Thus, we have to choose wisely which facts to generate. Intuitively, negative samples introduce repulsive forces in the embedding space so that entities that are not interchangeable in a fact should have embeddings far away from each other. It is of course easy to generate negative facts, simply by violating type constraints. For example, we can generate *hasChild(Michelle, USA)*, which is certain to be a false statement (due to the domain and range constraints, see Section 2.3.2). Then, however, we run into the *zero loss problem* [78]: The model learns to compute a low score for statements that are so unrealistic that they are not of interest anyway. It will not learn to compute a low score for statements such as *hasChild(Michelle, Lisa)*, which is the type of statements that we are interested in.

We thus have to choose negative facts that are as realistic as possible. As training goes on, we will provide the model with negative samples that are closer and closer to true facts, in order to adjust in a finer way the embeddings. To this end, various methods have been presented. One of them uses rules on the types of entities in order to avoid impossible negative facts such as *hasChild(Michelle, USA)* [42]. Another more complex method is adversarial negative sampling [78].

4.3.2 Shallow Models

Shallow models rely on the intuition that for a given fact $r(s, o)$, we would like the vectors $\boldsymbol{s}+\boldsymbol{r}$ and $\boldsymbol{o}$ to be close in the embedding space. For example, we would like the vectors $\boldsymbol{Barack} + \boldsymbol{marriedTo}$ and $\boldsymbol{Michelle}$ to be close. There are

two ways to define "close": The vectors can have a small vector difference (which is what translational models aim at) or they can have a small angle between them (which is what semantic-matching models do). The simplest translational model is TransE [8]. Its scoring function is simply the opposite of the distance between $\boldsymbol{s}+\boldsymbol{r}$ and $\boldsymbol{o}$: $f_{\boldsymbol{r}}(\boldsymbol{s},\boldsymbol{o}) = -||\boldsymbol{s}+\boldsymbol{r}-\boldsymbol{o}||$ (where $||\cdot||$ is either the 1-norm or the 2-norm). Maximizing this scoring function for true facts and minimizing it for negative facts leads to embeddings that should verify the simple arithmetic equation $\boldsymbol{s}+\boldsymbol{r} \approx \boldsymbol{o}$.

Let us now see how a network can be made to compute this score. As an example, let us embed in $\mathbb{R}^3$ the KB of Figure 1 with TransE and the 2-norm. The network to this end is shown in Figure 4.

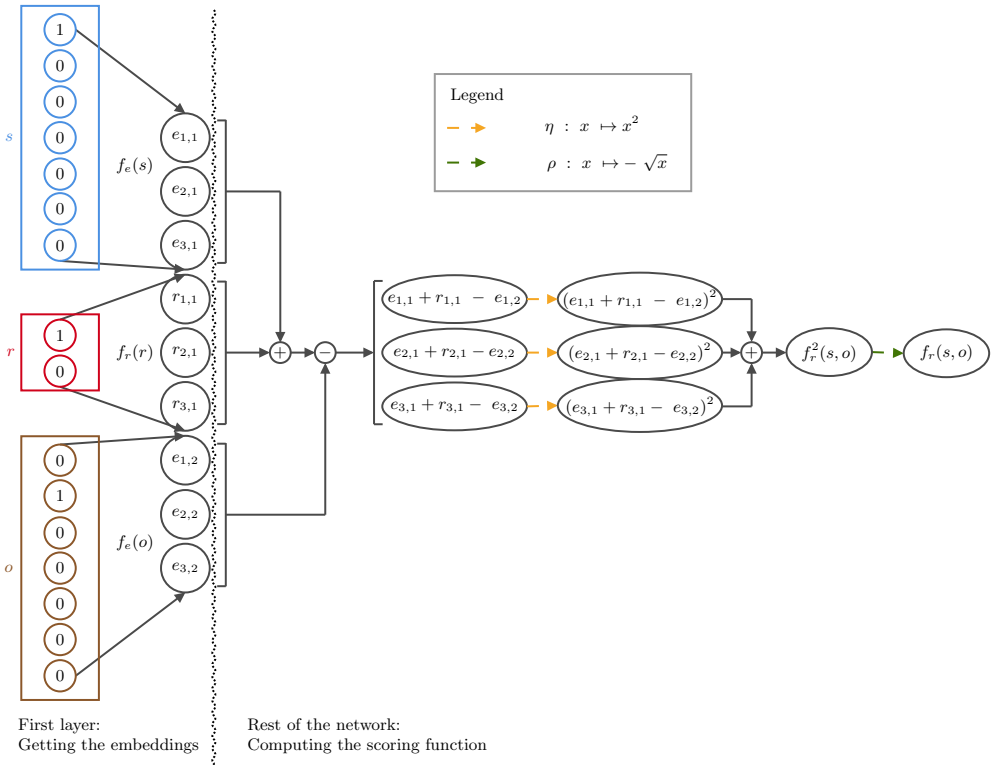

Fig. 4. Graphical representation of the TransE Model (with 2-norm) applied to the fact *marriedTo(Barack,Michelle)* for Figure 1, with $s = (1\,0\,0\,0\,0\,0\,0)^T$, $o = (0\,1\,0\,0\,0\,0\,0)^T$ and $r = (1\;0)^T$.

The network takes as input a fact $r(s,o)$, i.e., the concatenation of the one-hot encodings of s, r, and o. The first layer computes the embeddings of these items. The trick is that we will use not a single weight matrix for the first layer,

but two: One weight matrix $E \in \mathbb{R}^{3\times 7}$ to compute the embedding of entities s and o and one weight matrix $R \in \mathbb{R}^{3\times 2}$ to compute the embedding of relations r. Thus, when we train the network, we will learn the same matrix (and thus the same embeddings) for entities independently of their roles (as subject or object of the facts). We use no activation function in the first layer (or the identity function but it is not really an activation function as it is linear). Thus, the first layer computes simply, for a given fact $r(s, o)$, the embeddings $E \cdot h(s)$, $R \cdot h(r)$ and $E \cdot h(o)$ where h is the one-hot encoding function.

The next layer of the network will add the embeddings of s and r, i.e., it will take as input two 3-dimensional vectors, and produce as output a single 3-dimensional vector. This can be done by a simple matrix multiplication of the concatenated vectors with a fixed matrix, as shown here:

$$\begin{pmatrix} 1\,0\,0\,1\,0\,0 \\ 0\,1\,0\,0\,1\,0 \\ 0\,0\,1\,0\,0\,1 \end{pmatrix} \cdot \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \end{pmatrix} \tag{7}$$

Thus, this hidden layer of the network has no parameter to be trained – it just performs a sum operation with a fixed weight matrix. Therefore, Figure 4 just shows a $\bigoplus$. We use the same trick again to compute the difference between the sum of the embeddings of s and r (which is a vector in $\mathbb{R}^3$), and the embedding of o (also a vector in $\mathbb{R}^3$).

The next layer receives as input the 3-dimensional vector $\boldsymbol{s}+\boldsymbol{r}-\boldsymbol{o}$, and it has to produce the value $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o}) = -||\boldsymbol{s} + \boldsymbol{r} - \boldsymbol{o}||_2$. For this purpose, we first have to square every component of the vector. This can be done by a hidden layer with 3 input nodes and 3 output nodes. The weight matrix is just the identity matrix: It has only zeroes, and one's in the diagonal, i.e., it just passes every value from an input node to the corresponding output node. This matrix also remains constant during the training. The activation function η just squares the value for each node. The next hidden layer receives a vector of size 3, and adds up the components (as we have done before). The final layer just applies the activation function $\rho : x \mapsto -\sqrt{x}$.

Thus, the model computes, for an input fact $r(s, o)$, the score $f_{\boldsymbol{r}}(\boldsymbol{s}, \boldsymbol{o}) = -||\boldsymbol{s} + \boldsymbol{r} - \boldsymbol{o}||_2$. It suffices to train this model on positive and negative facts to adjust the weights so that this score is high for positive facts, and low for negative ones.

Limitations of the TransE Model. Although TransE yields promising results, it does not work well for one-to-many, many-to-one or many-to-many relations. As an example, consider the relation *wrote*. This is a one-to-many relation because one author can write several books: the facts *wrote(Albert Camus, La Peste)* and (*wrote(Albert Camus, L'étranger)* are both true. The TransE approach tends to return the same embeddings for the books of an author (i.e.,

La Peste and *L'étranger*), because they are almost interchangeable for the loss function, they have very few facts to distinguish them.

This limitation has been overcome by adding a relation-specific projection step. Entities should be projected into a relation-specific subspace before the translation happens. Let p_r be a relation-specific projection, i.e., a linear function from entity embeddings to vectors of possibly smaller size (projecting into a subspace). Then we want $p_r(\boldsymbol{s})+\boldsymbol{r}$ and $p_r(\boldsymbol{o})$ to be close (in distance or in similarity). Since a projection is a linear operation, it just adds another layer in the network after the first one (which returns the embeddings) and before the computation of the scoring function starts.

If, in our previous example, all entities are projected into a subspace that is specific to *wrote*, the projections of the books could all be the same. We can thus arrive at the desired arithmetic relations between the projected entities without forcing all books to have the same embedding. This idea has given rise to various refinements of the TransE model, depending on the projections considered. Here are a couple of the methods that followed TransE: TransH (projections on hyperplanes), TransR (projections on any type of subspace), TransD (projections with dynamic mapping matrices built both from entities and relations) [79,43,35].

We mainly presented translational models because they are the most intuitive ones, but semantic models achieve good results as well. The first such model (RESCAL [54]) lacks some representation capacity (as TransE) and was later refined to more complex models such as DistMul, HolE, ComplEx, ANALOGY [89,53,75,45].

4.3.3 Deep Models

Deeper architectures have also been introduced with the hope that hidden layers can capture more complex interaction patterns between entities and relations (and then estimate more complex scoring functions). In such models, the first part of the network (which, in shallow networks, just maps facts to their embeddings or their projections) now adds additional layers (possibly numerous) that receive as inputs the embeddings, and produce as outputs some extracted features. The second part of the network now computes the scoring function from the features extracted by the first part of the network and not directly from the embedding (or its projection) as in shallow models. The scoring function also becomes a parameter of the model (to be trained) and is not defined a priori anymore (in TransE for example it was only a distance between $\boldsymbol{s}+\boldsymbol{r}$ and $\boldsymbol{o}$). Note that we often loose the interpretability of the scoring function in this process. Examples of such methods are SME, NTN, and MLP [7,64,15] and more recent ones that include convolutional structures ConvE, ConvKB [14,52].

4.3.4 Fact Prediction

Commonly, authors compare the performance of their embedding methods on two tasks: fact checking and link prediction. Fact checking is simply deciding whether a given fact is true or false. To see how the embedding performs on this task, we train the network on a portion of the KB (i.e., on a subset of the facts) in which all entities and relations appear at least once. Then, we check for each fact from the KB that the network has not seen during training whether the network computes a score that is higher than a threshold (i.e., whether the network correctly assumes the fact to be true). The thresholds are determined on a validation set extracted from the training set.

Link prediction is a bit more complex. Again, we train the network on a portion of the KB. For a given fact $r(s, o)$ that the network did not see during training, the value of the scoring function $f_r(s, e)$ (resp. $f_r(e, o)$) is computed with the model for all entities e. This allows ranking candidate entities by decreasing order of scoring function. Then we count the share of unseen facts that the model manages to recover when the object (resp. subject) is hidden. Such evaluations are usually done under the Closed World Assumption (Section 2.4.1): If the network predicts a fact that is not in the KB, this is counted against the network – although the fact may be true but just unknown to the KB.

Thus, link prediction amounts to predicting facts – much like rules predict facts (Section 3.1). The difference is that rules are explicit: they tell us which circumstances lead to a prediction. Networks are not: they deduce new facts from the overall similarity of the facts. Another difference is that a rule does not know about the other rules: The confidence of a prediction does not increase if another rule makes the same prediction. Networks, in contrast, combine evidence from all types of correlations, and may thus assign a higher score to a fact for which is has more evidence.

4.4 Challenges in Representation Learning

While representation learning for knowledge graphs has made big advances these recent years, some challenges remain to be tackled:

Generalization of Performances. Current models tend to have performances that do not generalize well from one dataset to the other. Most methods are heuristics executing a more or less intuitive approach. A theory that could explain the variation of performance is missing.

Negative Sampling. Finding realistic negative facts remains a challenge in knowledge base embedding – much like in rule mining (Section 3.5). Here, we could use logical constraints. For example, if we know that Lisa cannot have more than two parents, then we could use *hasChild(Michelle, Lisa)* as a negative fact.

Dealing with Literals. Most current methods consider literals as monolithic entities. Thus, they are unable to see, e.g., that the date "2019-01-01" is close to the date "2018-12-31", or that the number "99" is close to the number "100". Such knowledge could lead to more accurate fact scoring functions.

Scalability. The development of massive KBs such as Wikidata requires algorithms to be able to scale. There is still room for improvement here: Embedding methods are usually tested on the FB15k dataset, which counts only 500 thousand facts – while Wikidata counts more than 700 million.

5 Conclusion

In this article, we have investigated how entities, relations, and facts in a knowledge base can be represented. We have seen the standard knowledge representation model of instances and classes. We have also seen an alternative representation of entities, as embeddings in a vector space. We have then used these representations to predict new facts – either through logical rules (by the help of rule mining), or through link prediction (with the help of neural networks).

Many challenges remain: the knowledge representation of today's KBs remains limited to subject-relation-object triples (Section 2.6). In Rule Mining, we have only just started looking beyond Horn Rules (Section 3.5). In Knowledge Base Embeddings, we have to learn how to generate more realistic negative examples (Section 4.4). This is one of the areas where the Semantic Web community and the Machine Learning community can have fruitful interchanges.

References

1. Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, volume 1215, 1994.
2. Alfred V. Aho, Michael R Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2), 1972.
3. Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook*. Springer, 2003.
4. Meghyn Bienvenu, Fabian M. Suchanek, and Daniel Deutch. Provenance for web 2.0 data. In *SDM workshop*, 2012.
5. Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the Web. In *WWW*, 2008.
6. Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *SIGMOD*, 2008.
7. Antoine Bordes, Xavier Glorot, Jason Weston, and Yoshua Bengio. A semantic matching energy function for learning with multi-relational data. *Machine Learning*, 94(2), Feb 2014.
8. Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. Translating Embeddings for Modeling Multi-relational Data. In *NIPS*, 2013.
9. A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka Jr., and T. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.
10. Jeremy J Carroll, Christian Bizer, Pat Hayes, and Patrick Stickler. Named graphs, provenance and trust. In *WWW*, 2005.
11. Yang Chen, Daisy Zhe Wang, and Sean Goldberg. Scalekb: Scalable learning and inference over large knowledge bases. In *VLDBJ*, 2016.

12. Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In *Probabilistic Inductive Logic Programming*. Springer, 2008.
13. Luc Dehaspe and Luc De Raedt. Mining association rules in multiple relations. In *International Conference on Inductive Logic Programming*, 1997.
14. Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2d Knowledge Graph Embeddings. In *AAAI*, 2018.
15. X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang. Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In *SIGKDD*, 2014.
16. Minh Duc Tran, Claudia d'Amato, Binh Thanh Nguyen, and Andrea G. B. Tettamanzi. Comparing rule evaluation metrics for the evolutionary discovery of multi-relational association rules in the semantic web. In *Genetic Programming*, 2018.
17. Oren Etzioni, Michael Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in knowitall:. In *WWW*, 2004.
18. Nicola Fanizzi, Claudia d'Amato, Floriana Esposito, and Pasquale Minervini. Numeric prediction on owl knowledge bases through terminological regression trees. *International Journal of Semantic Computing*, 6(04), 2012.
19. C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.
20. David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3), 2010.
21. Michael David Fisher, Dov M Gabbay, and Lluis Vila. *Handbook of temporal reasoning in artificial intelligence*. Elsevier, 2005.
22. Mohamed H. Gad-Elrab, Daria Stepanova, Jacopo Urbani, and Gerhard Weikum. Exception-enriched rule learning from knowledge graphs. In *ISWC*, 2016.
23. Luis Galárraga, Simon Razniewski, Antoine Amarilli, and Fabian M. Suchanek. Predicting completeness in knowledge bases. In *WSDM*, 2017.
24. Luis Galárraga and Fabian M. Suchanek. Towards a numerical rule mining language. In *AKBC workshop*, 2014.
25. Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Amie: Association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*, 2013.
26. Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast rule mining in ontological knowledge bases with amie+. In *VLDBJ*, 2015.
27. Lise Getoor and Christopher P Diehl. Link mining: a survey. *ACM SIGKDD Explorations Newsletter*, 7(2), 2005.
28. Claudio Gutierrez, Carlos A Hurtado, and Alejandro Vaisman. Introducing time into rdf. *IEEE Transactions on Knowledge and Data Engineering*, 19(2), 2007.
29. James Hawthorne. Inductive logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2018.
30. Sebastian Hellmann, Jens Lehmann, and Sören Auer. Learning of owl class descriptions on very large knowledge bases. *J. on Semantic Web and Information Systems*, 5(2), 2009.
31. Leah Henderson. The problem of induction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, 2019.
32. Vinh Thinh Ho, Daria Stepanova, Mohamed H. Gad-Elrab, Evgeny Kharlamov, and Gerhard Weikum. Rule learning from knowledge graphs guided by embedding models. In *ISWC*, 2018.

33. Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. Yago2: A spatially and temporally enhanced knowledge base from wikipedia. In *Artificial Intelligence* , 2013.
34. Katsumi Inoue. Induction as consequence finding. *Machine Learning*, 55(2), 2004.
35. Guoliang Ji, Shizhu He, Liheng Xu, Kang Liu, and Jun Zhao. Knowledge Graph Embedding via Dynamic Mapping Matrix. In *ACL*, Beijing, China, 2015.
36. Tim Kimber, Krysia Broda, and Alessandra Russo. Induction on failure: Learning connected horn theories. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 2009.
37. Markus Krötzsch, Maximilian Marx, Ana Ozaki, and Veronika Thost. Attributed description logics: Reasoning on knowledge graphs. In *IJCAI*, 2018.
38. Jonathan Lajus and Fabian M. Suchanek. Are all people married? determining obligatory attributes in knowledge bases. In *WWW*, 2018.
39. Ni Lao, Tom Mitchell, and William W Cohen. Random walk inference and learning in a large scale knowledge base. In *EMNLP*, 2011.
40. Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia - a large-scale, multilingual knowledge base extracted from wikipedia. *Semantic Web Journal*, 6(2), 2015.
41. Douglas B Lenat and Ramanathan V Guha. *Building large knowledge-based systems; representation and inference in the Cyc project*. Addison-Wesley Longman Publishing Co., Inc., 1989.
42. Adam Lerer, Ledell Wu, Jiajun Shen, Timothee Lacroix, Luca Wehrstedt, Abhijit Bose, and Alex Peysakhovich. PyTorch-BigGraph: A Large-scale Graph Embedding System. In *Conference on Systems and Machine Learning*, 2019.
43. Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. Learning Entity and Relation Embeddings for Knowledge Graph Completion. In *AAAI*, 2015.
44. H. Liu and P. Singh. Conceptnet. *BT Technology Journal*, 22(4), October 2004.
45. Hanxiao Liu, Yuexin Wu, and Yiming Yang. Analogical inference for multi-relational embeddings. In *ICML*, 2017.
46. Eric Margolis and Stephen Laurence. Concepts. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford, 2014.
47. Maximilian Marx, Markus Krötzsch, and Veronika Thost. Logic on mars: Ontologies for generalised property graphs. In *IJCAI*, 2017.
48. André Melo, Martin Theobald, and Johanna Völker. Correlation-based refinement of rules with numerical attributes. In *FLAIRS*, 2014.
49. Stephen Muggleton. Inverse entailment and progol. *New generation computing*, 13(3-4), 1995.
50. Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19, 1994.
51. Stephen Muggleton and Cao Feng. *Efficient induction of logic programs*. 1990.
52. Dai Quoc Nguyen, Tu Dinh Nguyen, Dat Quoc Nguyen, and Dinh Phung. A Novel Embedding Model for Knowledge Base Completion Based on Convolutional Neural Network. In *NAACL*, 2018.
53. Maximilian Nickel, Lorenzo Rosasco, and Tomaso Poggio. Holographic Embeddings of Knowledge Graphs. In *AAAI*, 2016.
54. Maximilian Nickel, Volker Tresp, and Hans-Peter Kriegel. A Three-way Model for Collective Learning on Multi-relational Data. In *ICML*, 2011.
55. Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. Robust discovery of positive and negative rules in knowledge bases. In *ICDE*, 2018.

56. Thomas Pellissier Tanon, Daria Stepanova, Simon Razniewski, Paramita Mirza, and Gerhard Weikum. Completeness-aware rule learning from knowledge graphs. In *ISWC*, 2017.
57. Gordon Plotkin. Automatic methods of inductive inference. 1972.
58. S. Ponzetto R. Navigli. BabelNet: The automatic construction, evaluation and application of a wide-coverage multilingual semantic network. *Artificial Intelligence*, 193, 2012.
59. Oliver Ray, Krysia Broda, and Alessandra Russo. Hybrid abductive inductive learning: A generalisation of progol. In *International Conference on Inductive Logic Programming*. Springer, 2003.
60. Simon Razniewski, Fabian M. Suchanek, and Werner Nutt. But what do we actually know? In *AKBC workshop*, 2016.
61. Bertrand Russell. *The Problems of Philosophy*. Barnes & Noble, 1912.
62. S. Russell and P. Norvig. *Artificial Intelligence: a Modern Approach*. Prentice Hall, 2002.
63. Ehud Y Shapiro. *Inductive inference of theories from facts*. Yale University, Department of Computer Science, 1981.
64. Richard Socher, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning With Neural Tensor Networks for Knowledge Base Completion. In *NIPS*. 2013.
65. Arnaud Soulet, Arnaud Giacometti, Beatrice Markhoff, and Fabian M. Suchanek. Representativeness of Knowledge Bases with the Generalized Benford's Law. In *ISWC*, 2018.
66. J. F. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations*. Brooks/Cole, 2000.
67. Steffen Staab and Rudi Studer, editors. *Handbook on Ontologies*. International Handbooks on Information Systems. Springer, 2004.
68. Daria Stepanova, Mohamed H Gad-Elrab, and Vinh Thinh Ho. Rule induction and reasoning over knowledge graphs. In *Reasoning Web International Summer School*. Springer, 2018.
69. Fabian M. Suchanek, Serge Abiteboul, and Pierre Senellart. Paris: Probabilistic alignment of relations, instances, and schema. In *VLDB*, 2012.
70. Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago – a core of semantic knowledge. In *WWW*, 2007.
71. Fabian M. Suchanek and Nicoleta Preda. Semantic Culturomics . In *VLDB short paper track*, 2014.
72. Niket Tandon, Gerard de Melo, Abir De, and Gerhard Weikum. Knowlywood: Mining Activity Knowledge From Hollywood Narratives. In *CIKM*, 2015.
73. Niket Tandon, Gerard de Melo, Fabian M. Suchanek, and Gerhard Weikum. WebChild: Harvesting and Organizing Commonsense Knowledge from the Web . In *WSDM*, 2014.
74. Matus Telgarsky. Representation Benefits of Deep Feedforward Networks. *arXiv [cs]*, September 2015.
75. Théo Trouillon and Maximilian Nickel. Complex and Holographic Embeddings of Knowledge Graphs: A Comparison. *arXiv [cs, stat]*, 2017.
76. Octavian Udrea, Diego Reforgiato Recupero, and VS Subrahmanian. Annotated rdf. *ACM Transactions on Computational Logic*, 11(2), 2010.
77. Denny Vrandečić and Markus Krötzsch. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10), 2014.
78. Peifeng Wang, Shuangyin Li, and Rong Pan. Incorporating GAN for Negative Sampling in Knowledge Representation Learning. In *AAAI*, 2018.

79. Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. Knowledge Graph Embedding by Translating on Hyperplanes. In *AAAI*, 2014.
80. Chris Welty, Richard Fikes, and Selene Makarios. A reusable ontology for fluents in owl. In *FOIS*, 2006.
81. Alfred North Whitehead and Bertrand Russell. *Principia mathematica*. 1913.
82. Word Wide Web Consortium. RDF Primer, 2004.
83. Word Wide Web Consortium. RDF Vocabulary Description Language 1.0: RDF Schema, 2004.
84. Word Wide Web Consortium. SKOS Simple Knowledge Organization System, 2009.
85. Word Wide Web Consortium. OWL 2 Web Ontology Language, 2012.
86. Word Wide Web Consortium. SPARQL 1.1 Query Language, 2013.
87. Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Qiuyue Wang, and Gerhard Weikum. Relationship Queries on Extended Knowledge Graphs. In *WSDM*, 2016.
88. Akihiro Yamamoto. Hypothesis finding based on upward refinement of residue hypotheses. *Theoretical Computer Science*, 298(1), 2003.
89. Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. Embedding Entities and Relations for Learning and Inference in Knowledge Bases. In *ICLR*, 2014.
90. Kaja Zupanc and Jesse Davis. Estimating rule quality for knowledge base completion with the relationship between coverage assumption. In *WWW*, 2018.

A Computation of Support and Confidence

Notation. Given a logical formula ϕ with some free variables $x_1, \ldots, x_n$, all other variables being by default *existentially* quantified, we define:

$$\#(x_1, \ldots, x_n) : \phi \quad := \quad |\{(x_1, \ldots, x_n) \ : \ \phi(x_1, \ldots, x_n) \text{ is } \mathit{true}\}|$$

We remind the reader of the two following definitions:

Definition 14 (Prediction of a rule): *The predictions P of a rule $\boldsymbol{B} \Rightarrow h$ in a KB $\mathcal{K}$ are the head atoms of all instantiations of the rule where the body atoms appear in $\mathcal{K}$. We write $\mathcal{K} \wedge (\boldsymbol{B} \Rightarrow h) \models P$.*

Definition 19 (Support): *The support of a rule in a KB is the number of positive examples predicted by the rule.*

A prediction of a rule is a positive example if and only if it is in the KB. This observation gives rise to the following property:

Proposition 34 (Support in practice): *The support of a rule $\boldsymbol{B} \Rightarrow h$ is the number of instantiations of the head variables that satisfy the query $\boldsymbol{B} \wedge h$. This value can be written as:*

$$\mathit{support}(\boldsymbol{B} \Rightarrow h(x, y)) = \#(x, y) : \boldsymbol{B} \wedge h(x, y)$$

Definition 20 (Confidence): *The confidence of a rule is the number of positive examples predicted by the rule (the support of the rule), divided by the number of examples predicted by the rule.*

Under the CWA, all the predicted examples are either positive examples or negative examples. Thus, the standard confidence of a rule is the support of the rule divided by the number of prediction of the rule, written:

$$\mathit{std\text{-}conf}(\boldsymbol{B} \Rightarrow h(x, y)) = \frac{\#(x, y) : \boldsymbol{B} \wedge h(x, y)}{\#(x, y) : \boldsymbol{B}}$$

Assume h is more functional than inverse functional. Under the PCA, a predicted negative example is a prediction $h(x, y)$ that is not in the KB, such that, for this x there exists another entity y' such that $h(x, y')$ is in the KB. When we add the predicted positive examples, the denominator of the PCA confidence becomes:

$$\#(x, y) : (\boldsymbol{B} \wedge h(x, y)) \vee (\boldsymbol{B} \wedge \neg h(x, y) \wedge \exists y'.h(x, y'))$$

We can simplify this logical formula to deduce the following formula for computing the PCA confidence:

$$\mathit{pca\text{-}conf}(\boldsymbol{B} \Rightarrow h(x, y)) = \frac{\#(x, y) : \boldsymbol{B} \wedge h(x, y)}{\#(x, y) : \boldsymbol{B} \wedge \exists y'.h(x, y')}$$