Universität des Saarlandes
Max-Planck-Institut für Informatik
AG5

Question Generation from Knowledge Graphs

Masterarbeit im Fach Informatik
Master's Thesis in Computer Science
von / by

Dominic Seyler

angefertigt unter der Leitung von / supervised by

Prof. Dr. Klaus Berberich
Mohamed Yahya

begutachtet von / reviewers

Prof. Dr. Klaus Berberich
Prof. Dr. Gerhard Weikum

Oktober / October 2015

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, den 14. Oktober 2015,

(Dominic Seyler)

The power to question is the basis of all human progress.

- Indira Gandhi

Abstract

In this thesis we present a novel approach for generating natural language questions, using factual information from a knowledge graph and automatically assessing their difficulty. Our work elicits a further utilization of the knowledge captured in knowledge graphs that could find applications in research, education and leisure. In general, coming up with question manually can be a resource consuming endeavor. An automatic approach can therefore provide an alternative that substantially reduces the required effort. Established methods for question generation have used document corpora as their main source of information. However, the utilization of knowledge graphs for this purpose has received far less attention. Furthermore, to the best of our knowledge, no previous work has examined question difficulty in the context of question generation.

We specify a framework for a system staged in quiz setting that can test the domain knowledge of players. The framework is implemented as an end-to-end system that expects a human to specify a topic and a difficulty level. The resulting output is a question in natural language, that abides the input criteria. The challenges we address along the way include the principled selection of the contents of the question, the verbalization of these contents into natural language, and the creation of an automated question difficulty estimation scheme. We empirically show the effectiveness of our approach and conduct user studies to demonstrate the correlation between our automated difficulty judgments and those made by human annotators.

Acknowledgements

First and foremost, I would like to sincerely thank my supervisors Prof. Dr. Klaus Berberich and Mohamed Yahya for their substantial guidance and support. Their advice was essential for the success of my thesis and for this project. Moreover, our countless meetings were fruitful and productive, and it was more than a pleasure to collaborate with them.

Furthermore, I would like to express my highest gratitude towards Prof. Dr. Klaus Berberich and Prof. Dr. Gerhard Weikum for reviewing my thesis and for giving me the opportunity to work alongside the world-class scientists of the Databases and Information Systems department at the Max Planck Institute for Informatics. In this context, I would also like to thank my colleagues at the department for creating a friendly and open-minded working atmosphere.

Last, but not least, I would like to thank all participants of our user study, whose valuable feedback was elementary to the evaluation of our approach.

Contents

Chapter 1

Introduction

1.1 Motivation

A considerable amount of research has been invested into the extraction of factual knowledge from unstructured web resources. These efforts resulted in the creation of knowledge graphs, which provide this information in a machine-interpretable format. Among these are DBpedia, Freebase, and YAGO, which capture world knowledge in a broad range of domains. Given this topical diversity, there is great potential for the creation of a system that can facilitate this knowledge for educational purposes. As part of the learning process, the system could generate questions of a certain topic that is adequate to the learner's information need and expertise level. By using automatically generated questions as a medium for knowledge acquisition, a novel utilization for knowledge graphs could be created.

When crafting new questions it would be intriguing to study their properties. A property that comes to mind immediately is question difficulty. Even though difficulty depends on multiple factors of the individual who is supposed to answer the question, there is potential to inspect the characteristics of questions that influence difficulty positively or negatively. For example, consider the questions *Who created the painting Portrait of a Musician?* and *Who created the painting Mona Lisa?* Unless one is an Lenoardo da Vinci expert, it is very hard to relate him to the painting *Portrait of a Musician*. On the other hand, when talking about the creator of *Mona Lisa*, da Vinci comes to mind more effortlessly. If it could formally be captured what makes the former question harder than the other, this notion of difficulty could be integrated in a system to enable it making difficulty estimates.

There are multiple key-applications that could benefit from a system that automatically generates questions with difficulty estimates. As in the beginning, one field of application

could be in an educational setting, such as the automatic generation of tests and exams to measure the learning success of students. Another application is in professional education settings, such as the training of employees based on structured data about products, customers, or an organization. Furthermore, question generation could be of great use in the field of fraud detection on crowdsourcing platforms, e.g., Amazon Mechanical Turk. Since the correct answer to each question is known beforehand, it would be possible to discriminate between users who just click through the data and users who choose answers that are correct or closely related to the correct answer. Finally, the large number of meaningful queries that are generated by such a system could be used to drive natural language generation research, which focuses on questions.

In recent years, machine-interpretable knowledge resources have been used to automatically answer natural language questions on various domains. The most prominent example is the IBM Watson system [1] that participated in the popular TV-quiz-show Jeopardy! There, the system was able to beat two of the most successful human competitors in the show's history. However, in this work we address the reverse problem of generating natural language questions from knowledge graphs. Similar to the IBM Jeopardy! challenge, we stage our problem in the setting of a quiz game. Thus, our main objective is to "come up" with a natural language quiz question for a specific topic (e.g., Entertainment) and a specific difficulty (e.g., easy or hard).

1.2 Problem Definition

A main goal for the this thesis is to leverage the structured information from the knowledge graph to craft meaningful questions. Therefore, the task comprises the selection of the question's content, meaning which clues are contained in the question, and the question's answer. We decided to choose a structural query representation as the preliminary formulation of the question. Using this representation of the query enables us to develop a method to verbalize it into natural language, which is required for users to interpret the system's output. In addition to the crafting the question, we investigated the notion of question difficulty. In general question difficulty is subjective, since it depends on the individual who is supposed to answer the question. Thus, a further goal is to find a way to standardize the notion of difficulty and enable the system to automatically judge a question's difficulty.

1.3 Contributions

As stated above, the challenges we address along the way include the *generation* of the contents of the question, the *verbalization* of these contents for humans and the *judgment* of question difficulty. Correspondingly, our contributions fall into the following research areas:

- *Question Generation*: We propose a novel approach to generate a question, which has a unique answer, using semantic information from the knowledge graph. Our approach uses a SPARQL query as an intermediate representation of the question, and for checking if it has a unique answer.
- *Query Verbalization*: We elaborate on a pattern-based technique for verbalizing SPARQL queries, using lexical resources. The resulting natural language mimics the style of Jeopardy! clues. To cater to verbalization variety, we expanded the standard set of paraphrases for relations and created a method to distinguish important types for an entity.
- *Question Difficulty Estimation*: We designed, implemented and evaluated a question difficulty classifier trained on Jeopardy! data. The classifier's features are based on statistics computed from the knowledge graph and Wikipedia. With empirical studies and a human experiment, we were able to show that we achieve good performance on our training data (66% correctly classified) and that human evaluators moderately agree with our difficulty estimates in terms of relative and absolute difficulty judgments.

1.4 Outline of the Thesis

The remainder of this thesis is structured as follows. In Chapter 2, we provide technical background on knowledge bases and the Semantic Web. Moreover, we give a short introduction to machine learning and regression problems. We close the background chapter with a discussion about techniques to process large datasets. In Chapter 3, we present a summary of previous work in the areas of question generation, difficulty estimation of text and questions, verbalization techniques and work that focuses on the analysis of Jeopardy! questions. In the subsequent chapter, we present our approach for generating questions using a knowledge graph (Chapter 4). This section is divided by the problem domains of *question generation*, *query verbalization* and *difficulty estimation*. Chapter 5 gives a high-level overview of the system and discusses the implementation of the prototype. The implementation comprises a web interface for generating questions,

which was part of the poster presentation at the 2015 World Wide Web Conference in Florence, Italy. Another web interface was created as part of a user study to analyze the performance of the difficulty classifier. In Chapter 6, we empirically evaluate the performance of our difficulty estimation scheme on test data. In addition to the study, we perform an extensive user experiment to evaluate our agreement with human question difficulty judges. The closing chapter concludes the thesis and gives an outlook on future work.

Chapter 2

Technical Background

This chapter describes background knowledge that is necessary to further understand the concepts and algorithms introduced in this thesis. Section 2.1 introduces the *semantic web* framework and some of its relevant components. Section 2.2 gives an introduction to *knowledge bases* and presents two particular instances that form the data backbone of our system. Section 2.3 describes *machine learning* and the sub-task of regression analysis. Special focus is cast on *logistic regression* and model validation for classifiers. The final Section 2.4 elaborates on the *MapReduce* programming model and introduces two state-of-the-art systems based on MapReduce.

2.1 Semantic Web

The semantic web - also known as the web of data - was standardized by the World Wide Web consortium to extend the World Wide Web (WWW). It "provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries" [2]. Berners-Lee et al. [3] envisioned the semantic web as the future of the World Wide Web and the next step of its evolution. They state that the goal of the semantic web is to "bring structure to the meaningful content of web pages", therefore making it machine-interpretable. The structure is achieved by embedding machine-readable metadata into a web page, which enables automated agents to make sense of the information and perform tasks on behalf of users. The format of the metadata is described by a set of standards. The standards relevant for this thesis are: *URI* which identifies an abstract or physical resource, *RDF* which is used to describe properties of resources and *SPARQL* to formulate queries over these properties. The following sub sections will elaborate on these standards in more detail.

2.1.1 URI

A *Unified Resource Identifier* (URI) uniquely identifies a physical or abstract resource and is represented as a string of characters. It consists of a hierarchical sequence of components which are referred to as the *scheme*, *authority*, *path*, *query*, and *fragment*. Masinter et al. [4] define the components of an URI as follows:

- *scheme*: determines how the URI needs to be interpreted (e.g. HTTP)
- *authority*: indicates the responsibility of a certain party for this URI (e.g. a host)
- *path*: contains hierarchical data to identify the resource within the scope of the URI's scheme and naming authority
- *query*: provides non-hierarchical information to further identify the resource
- *fragment*: enables indirect identification of a secondary resource within the resource itself.

Their work also gives an example for an URI and its components:

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
```

2.1.2 RDF

The *Resource Description Framework* (RDF) is used to describe properties of resources. The data model provided by RDF describes these properties in the form of subject-predicate-object triples. The *subject* is the resource that is being described. The *object* can be another resource or a fixed value, called literal (e.g., an integer). The *predicate* indicates the kind of the relation between the subject and the object and is represented using a property (e.g., the property `rdf:type` indicates that a resource is an instance of a class). A set of these triples form a labeled, directed multigraph that can be queried using the *SPARQL* query language (Section 2.1.3).

The framework also provides the possibility to use namespaces to represent common URI prefixes in a more compact way. For example,
using the notation `@prefix wiki: http://en.wikipedia.org/wiki/`,
the URI `http://en.wikipedia.org/wiki/Alan_Turing`

subject	predicate	object
Alan_Turing	worksAt	University_of_Cambridge
Alan_Turing	livesIn	United_Kingdom
Robin_Grandy	hasAcademicAdvisor	Alan_Turing
Robin_Grandy	livesIn	United_Kingdom
University_of_Cambridge	isLocatedIn	United_Kingdom

Table 2.1: Example RDF graph in triple representation

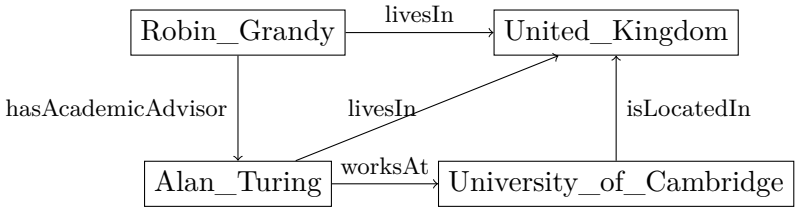

Figure 2.1: Example RDF graph from Table 2.1 in graphical representation

results in the much shorter `wiki:Alan_Turing`. An example of a RDF graph in triple representation can seen in Table 2.1 (prefixes have been omitted). Figure 2.1 depicts the graphical representation of the example RDF graph.

2.1.3 SPARQL

The *SPARQL Protocol and Query Language* (SPARQL) can be used to retrieve or manipulate data in the RDF graph. Prud'hommeaux and Seaborne [5] state that "SPARQL contains capabilities for querying [...] graph patterns along with their conjunctions and disjunctions. [...] The results of SPARQL queries can be results sets or RDF graphs." Conjunctions are expressed by the use of common variables and are denoted by a leading question mark. Disjunctions provide the capability to retrieve a matching subgraph if at least one of multiple graph patterns matches. In SPARQL a disjunction is expressed using the `UNION` keyword.

An example query, for the RDF graph given in Table 2.1, can be found in Figure 2.2. The query roughly translates to *Who lives in the United Kingdom and works at a place that is located in the United Kingdom.* The sole result for the query is the resource `yago:Alan_Turing`. In this example only one person matches the given criteria which showcases something that is true in general: No knowledge base can be complete. Figure 2.3 depicts the graph pattern with variables `?p` and `?u` that is matched against the RDF graph.

```
@prefix yago:http://yago-knowledge.org/resource/

SELECT ?p WHERE {
    ?p yago:livesIn yago:United_Kingdom .
    ?p yago:worksAt ?u .
    ?u yago:isLocatedIn yago:United_Kingdom
}
```

Figure 2.2: Example SPARQL query

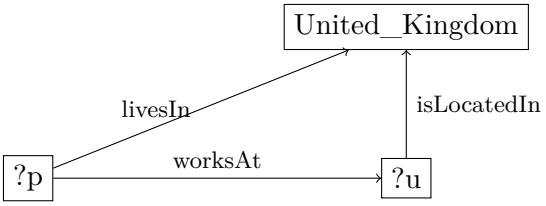

Figure 2.3: Graph pattern of example query

subject	predicate	object
soccer_player	rdfs:subClassOf	player
Ronaldo	rdf:type	soccer_player
Ronaldo	playsFor	Barcelona Football Club

Table 2.2: SPO triples of knowledge base relations

2.2 Knowledge Bases

A knowledge base is a, centrally accessible, aggregation of information. For example, a public library, a domain specific database or an online encyclopedia, such as Wikipedia, can all be generally regarded as knowledge bases [6]. In recent years the term has been used especially to refer to a database that stores information in an *ontological* representation. These knowledge bases store knowledge about classes and their relations (e.g., `soccer_player` is a subclass of `player`) and combine them with instance-level knowledge (e.g., `Ronaldo` is a `soccer_player`). In addition to instance-class affiliations, they store information about the relations between entities (e.g., `Ronaldo` plays for `Barcelona_Football_Club`). Table 2.2 shows these relations as SPO triples in a knowledge base. Subclass-class relation are denoted with `rdfs:subClassOf` and type relations are denoted with `rdf:type`.

Knowledge bases serve different purposes across various domains. They can contain *lexical information* (e.g., Wordnet [7]), which is utilized in the field of linguistics. They can contain *common sense* knowledge (e.g., WebChild [8]), which can be used for reasoning and question answering in artificial intelligence. Knowledge bases that contain very broad knowledge and are not restricted to a particular domain are called *general-purpose*

knowledge bases. Examples for these systems comprise YAGO [9], Freebase [10], DBpedia [11] and many others.

The *Linked Open Data Project* was created as an effort to establish a connection between these different systems. As stated in [12], its purpose is to "connect related data that wasn't previously linked, or using the Web to lower the barriers to linking data currently linked using other methods". The main idea is that two entities from two different knowledge bases which refer to the same physical entity can be connected with the `owl:sameAs` link. Using this method it can be expressed that the entity for Alan Turing in Freebase: `http://www.freebase.com/m/0n00` is the same as the YAGO entity `http://yago-knowledge.org/resource/Alan_Turing`.

2.2.1 Wikipedia

Wikipedia is a non-profit, free internet encyclopedia and is the result of collaborative work of more than 24 million volunteers. Started in January 2001, Wikipedia has grown continuously until reaching almost 5,000,000 English articles in 2015. As of November 2014, Wikipedia has articles in 288 Languages and about 69,000 active editors [13]. The encyclopedia is built on an open concept where originally every user, registered or not, could contribute by adding or editing articles. Over time this openness had to be constrained when the massive growth and popularity of the website attracted vandalism. As of today only registered users can add or edit articles that are not specially protected.

Wikipedia plays an important role as a data source when constructing knowledge bases. Its large amount of structured data, such as info boxes and category pages, can be utilized to extract clean and comprehensive facts about entities and their relations. By making use of semantic patterns mentions of entities can be spotted in unstructured text and used as evidence in disambiguation tasks. The underlying link structure, which expresses which articles are connected, can be exploited to gain valuable information about the entities that belong to the corresponding article.

2.2.2 WordNet

WordNet, as presented in Miller [7], is a lexical knowledge base that defines semantic relations between nouns, verbs, adjectives, and adverbs. These relations include synonymy and hyponymy. Synonymy makes it possible to group words into synonym sets (called synsets), where each synset expresses a distinct concept, or word sense. For example, the noun *bank* can be associated with multiple synsets: (1) "a building in which the business

of banking transacted", (2) "sloping land (especially the slope beside a body of water)", etc. In this thesis we exploit synsets to find synonymous words for types (Section 4.2.3).

Additionally, WordNet defines a hyponymy relation between nouns that organizes the meanings into a hierarchical structure. For example, the noun pairs *maple* and *tree* form a sub-class class relationship, since maple is a more precise variant of a tree. Yago (Section 2.2.3) makes use of the hyponymy relation when constructing the class taxonomy.

2.2.3 YAGO

As already mentioned, YAGO is a general-purpose ontology which contains world-knowledge of more than 4 million entities and has more than 120 million facts about these entities. Manual verification confirmed an accuracy of more than 95% which makes its quality comparable to an encyclopedia [9]. The second version of YAGO, presented in Hoffart et al. [14], has facts and events anchored in time and space. These events have been automatically extracted from various web resources, namely: Wikipedia, Geonames and WordNet.

In YAGO, Wikipedia pages are represented as entities and the entity's classes are retrieved from Wikipedia categories. The classes from Wikipedia are then intertwined with the WordNet class hierarchy to form a rich class taxonomy. The taxonomy captures how different classes are associated to each other. On the top level is `owl:Thing`, which all classes are subclass of. The further one descends in the hierarchy the more specific the classes become.

YAGO complies with the RDF standard and therefore stores its facts as subject-predicate-object (SPO) triples. In YAGO-terminology resources are known as entities whereas predicates are known as relations. Therefore, the expression of two entities forming a certain relation with each other is called a fact. An example of a fact with two entities stating that Alan Turing works at the University of Cambridge is expressed in YAGO as:

```
yago:Alan_Turing yago:worksAt yago:University_of_Cambridge
```

An example where an entity and a literal form a fact is expressed as:

```
yago:Alan_Turing yago:wasBornOnDate yago:1912-06-23
```

Additionally, entities are associated with one or more classes. This is expressed using the `rdf:type` relation. Classes in YAGO can be either Wikipedia classes, which is denoted by the prefix *wikicat*, or they can be WordNet classes, which is denoted by the prefix *wordnet*. For example the fact that Alan Turing is an English computer scientist is denoted as follows:

```
yago:Alan_Turing rdf:type yago:wikicat_English_computer_scientists
```

It has to be noted that only the immediate Wikipedia class memberships are considered. All super-classes of `English_computer_scientists` are not explicitly captured in YAGO. However, they can be derived from the class taxonomy.

2.2.4 AIDA

AIDA [15] is a framework for *entity linking* in natural language text or tables. Entity linking is the task of finding entity mentions, resolving their ambiguity and linking them to a known knowledge base identifier. AIDA maps mentions of ambiguous names onto canonical entities (e.g., individual people or places) that are registered in the YAGO knowledge base. This is done by harnessing context from knowledge bases in combination with the utilization of prior approaches. These approaches make use of "three measures: the prior probability of an entity being mentioned, the similarity between the contexts of a mention and a candidate entity, as well as the coherence among candidate entities for all mentions together". Using these metrics the system generates a dense subgraph which determines the best entity-mention mapping.

2.3 Machine Learning

Machine learning is a sub-field in computer science that explores algorithms that are able learn from a given training data set and use this learned "knowledge" to make predictions on unseen data. A similar definition of machine learning is given by Flach [16] where it is stated that "Machine learning is the systematic study of algorithms and systems that improve their knowledge or performance with experience."

According to Russell and Norvig [17], machine learning can be categorized into three broad learning tasks. In *supervised learning*, when training the system, the desired output for a given input is known and the system tries to derive a general rule that maps input of the training data to the output. In contrast to supervised learning, the input data is not labeled in *unsupervised learning* and therefore the system has to discover a hidden structure in the data independently. In *reinforcement learning* the system tries to reach a given goal without any external re-assurance if it has come closer or further from the goal. The only feedback that the system receives is whether it has reached the goal, or not, when arriving at a terminal state.

Apart from the learning task, machine learning systems can be further categorized depending on the output they produce. Bishop [18] identifies the following categories:

- *Classification*: The input data can be grouped into two or more classes. To achieve this, the classifier has to be trained beforehand and the system has to learn a model that enables it to map unseen inputs to one of these classes. In the case only two classes exist, the classifier is called *binary*, if more than two classes exist the classifier is called *dichotomous*.
- *Regression*: Here the outcome is a continuous value as opposed to a discrete class in the classification task.
- *Clustering*: In this task, similar to classification, we try to group the data into multiple classes but with the significant difference that these classes are not known beforehand.
- *Density Estimation*: Given the input data, the system tries to find an unknown probability density function that the data underlies.
- *Dimensionality Reduction*: Given input data with high dimensionality, the system maps the input a representation in lower dimensional space.

Following Flach [16], there are three main components for machine learning: *tasks*, *features*, and *models*. A *task* refers to the high-level problem that is being solved, e.g. distinguish genuine and spam email. Whereas a *feature* is a set of descriptors for an object that is being classified, e.g. "Number of characters in an email". The *model* is an mathematical representation of the relationship of the features with the class output, e.g. linear regression model. After choosing the model type, a learning algorithm is required that uses training data to build the model.

The most prominent machine learning method in this thesis is *classification*. As stated above, in this method the system tries to find the most suitable class label, given a set of observations. In Figure 2.4 the data flow of the classification task is depicted. The process starts on the right-hand side of the diagram. There, feature extraction is performed on the training data. The output of the training data is then used as input for the learning algorithm, which trains the mathematical model. After the model is trained, feature extraction is also performed on the unseen data. Using the trained model, the classifier can finally use the output of the features to classify the unseen data. A common example for such as system is a spam mail classifier which uses different observations in the email text or the email meta-data to decide if an email is spam or ham. The training data to this classifier is a set of hand-labeled emails. Using the input, the classifier can

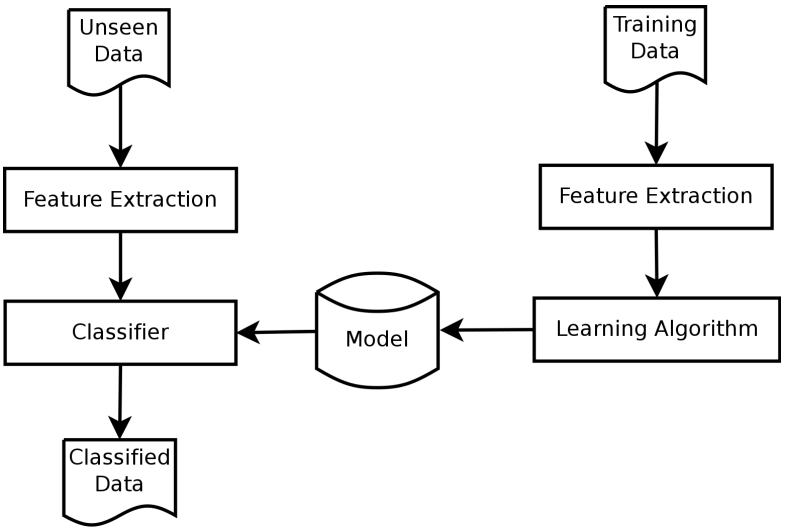

Figure 2.4: Data flow of a classification task in machine learning

then build a mathematical model which makes it possible for unseen mail to predict whether it is spam or ham. This training can be interpreted as the machine's experience. Thus, the more training data is available, the better the classifier performs.

2.3.1 Regression Analysis

To give a general introduction to all above mentioned machine learning problems and methods is beyond of the scope of this thesis. We therefore focus on a sub-task of machine learning called *regression analysis*. In regression analysis we try to model the relationship between a dependent variable (the "outcome" or "class") and multiple independent variables (often called "predictors", "attributes" or "features")[1]. Regression analysis is therefore especially useful when the classes and features are numeric.

[1]Because of the interchangeability of these terms we will strictly use the terms "class" and "features" in the course of this thesis which are also most commonly used in machine learning literature.

2.3.2 Linear Regression

Regression analysis can be best understood when looking at the linear regression model. Linear regression is the standard model that can be used if the class and features are numeric. The intuition behind the method is to express the class as a weighted linear combination of the features, as follows:

$$\hat{y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_N x_N \tag{2.1}$$

where $\hat{y}$ is the class value that is being predicted, β_i the i^{th} feature weight and x_i the i^{th} feature value, for all features $1...N$. To calculate these weights we make use of training data, which consist of labeled training instances[2]. Let (i) denote the i^{th} instance we predict the output value as follows, where $x_0^{(i)}$ is always set to one:

$$\beta_0 x_0^{(i)} + \beta_1 x_1^{(i)} + \beta_2 x_2^{(i)} + ... + \beta_N x_N^{(i)} = \sum_j^N \beta_j x_j^{(i)} \tag{2.2}$$

After obtaining the prediction, it is compared to the actual value of the class. The regression method now minimizes the difference between the predicted and the actual value by adjusting the weights β_j. This is done by minimizing the square of these differences (or errors) which can be expressed as the following equation:

$$\sum_{i=1}^{n} (y^{(i)} - \sum_{j=0}^{N} \beta_j x_j^{(i)})^2 \tag{2.3}$$

where $y^{(i)}$ is the actual class value of instance i, for all instances 1 through n and all features 1 through N. The minimization of this function can be done using the linear least squares approach and will result in feature weight estimates $\hat{\beta}_j$, $j = 1, ..., N$. The least squares approach, also called *ordinary least squares*, is the most studied and widely applied method for the estimation of regression parameters. A detailed description is given in Chapter 2.2 in Groß [19], for the interested reader.

[2]Since the classes of the training data is visible to the classifier while training, regression analysis falls into the category of supervised training algorithms.

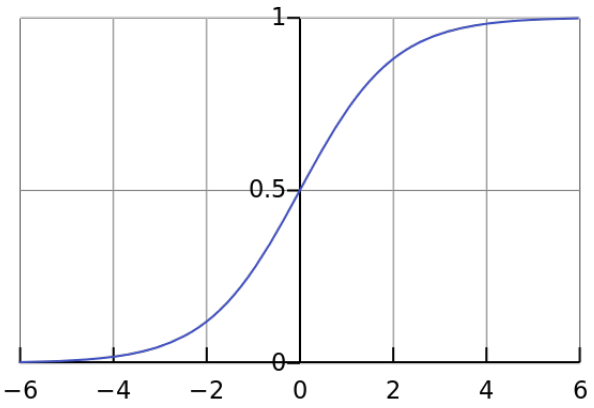

Figure 2.5: The logistic function [20]

2.3.3 Logistic Regression

As seen above, when dealing with numeric classes the linear regression model is a natural choice. But when dealing with discrete classes we are required to use the logistic regression model[3]. In this model, instead of predicting the numerical class value directly, a probability of an instance belonging to a class is estimated. For this reason the method takes advantage of the properties of the logistic function which, for any input form negative to positive infinity, takes always an output between one and zero and can therefore be interpreted as a probability. The graph for the logistic function can be seen in Figure 2.5.

To further explain logistic regression we first have to define the logistic function as follows:

$$\sigma(t) = \frac{e^t}{e^t + 1} = \frac{1}{1 + e^{-t}} \tag{2.4}$$

where t is a linear function of feature weights β_i and feature values x_i, $i = 1, ..., N$ where N is the number of features. t can therefore be expressed as:

$$t = \sum_i \beta_i x_i \tag{2.5}$$

Plugging this into the logistic function we obtain the logistic regression function:

$$F(x) = \frac{1}{1 + e^{-(\sum_i \beta_i x_i)}} \tag{2.6}$$

[3]The term logistic regression is usually used when dealing with binary classes. When the data can be divided among more than two classes the term multinational logistic regression is used. In this thesis we consider only binary class problems.

Similarly to linear regression the method needs to find the weights β_i that best fit the training data. As discussed in Section 2.3.2, linear regression makes use of the least squares method to estimate the unknown parameters β_i. However, since the outcome of logistic regression is dichotomous, the least squares approach is not applicable [21]. The method that yields the least squares function, when applied to the linear regression model, is called *maximum likelihood*. For a fixed sample, the maximum likelihood method selects the underlying model's parameters such that it maximizes the likelihood function. Thus, the method chooses the parameters such that it maximizes the probability of obtaining the observed data with the model at hand. To construct the likelihood function $l(\beta)$ in the case of logistic regression, we assume that observations are independent. This assumption leads to the following product:

$$l(\beta) = \prod_{i=1}^{n} \sigma(x_i)^{y_i}[1 - \sigma(x_i)]^{1-y_i} \tag{2.7}$$

where y_i are either 0 or 1 according to the outcome of our model. In maximum likelihood estimation we are only interested in the parameters β_i that maximize $l(\beta)$. Since it is mathematically more convenient to work with the logarithm of $l(\beta)$, we can do so and still obtain correct weights β_i. This expression is referred to as log-likelihood and is given as:

$$L(\beta) = \sigma(l(\beta)) = \sum_{i}^{n}(1 - y_i)\sigma(1 - F(x_i)) + y_i\sigma(F(x_i)) \tag{2.8}$$

Now the weights β_i need to be chosen such that the log-likelihood is maximized. This is done by differentiating $L(\beta)$ with respect to $\beta_1, \beta_2, ..., \beta_N$ and setting the resulting expressions equal to zero. The resulting equations are referred to as *likelihood equations*. As mentioned earlier, in linear regression the likelihood equations can be solved using the least squares approach. However, in logistic regression these equations are non-linear. Finding their solution requires iteratively solving a sequence of weighted least-squares regression problems until the log-likelihood converges. The convergence happens usually within a few iterations. For details of this method we would like to refer the interested reader to McCullagh et al. [22].

Using the logistic regression function for binary classification can be easily visualized by imagining a decision boundary at probability 0.5. The classification now works as follows: If $F(x) < 0.5$ the instance belongs to class 0 and if $F(x) > 0.5$ to class 1. In case $F(x) = 0.5$ we have no clear classification and we can classify the instance as belonging to either class.

2.3.3.1 Ridge Regression

In logistic regression, ridge estimators can be used to improve the weight estimates for features and thereby reducing the error made by future predictions. The effect of ridge regression is, that the higher the *ridge parameter* is chosen, the more the feature weights shrink towards zero. This property avoids overfitting and is especially useful when dealing with a dataset were the number of features is relatively large compared to the number of observations. Le Cessie et al. [23] show an approach to extend ridge regression theory in standard linear regression to logistic regression. Their approach maximizes the log-likelihood of the logistic regression model (see Equation 2.8) by introducing a penalty of the form:

$$l^{\lambda}(\beta) = l(\beta) - \lambda||\beta||^2 \tag{2.9}$$

where $l(\beta)$ is the unrestricted log-likelihood function and $||\beta|| = (\sum \beta_j^2)^{\frac{1}{2}}$, the l^2-norm of feature weight vector β. The ridge parameter (λ) controls the amount of shrinkage of β towards 0. Thus, if $\lambda = 0$ we perform ordinary maximum likelihood estimation, whereas $\lambda \to \infty$, all β_j will tend to 0. A large number of features will give rise to unstable feature weight estimates β_j. Shrinking them towards 0, while allowing for a little bias, stabilizes the model and provides estimates with smaller variance [23].

2.3.3.2 Odds Ratio

After fitting a logistic regression we need to somehow asses the correctness of the estimated feature weights and interpret their values. As stated by Hosmer et al. [21], the main question that is being addressed is "What do the estimated coefficients in the model tell us about the research questions that motivated the study?". Looking at the plain feature weights only does not necessarily answer this question. For better interpretability of the model's parameters *odds ratios* have been introduced in literature. In the simple case with only one feature x that can take values 0 or 1, the odds ratio quantifies how strongly the presence of x is associated with the outcome of the classification. Following the example of [21], let the classification outcome y be the presence or absence of heart disease and feature x denote whether or not the person engages in regular exercise. An odds ratio of 0.5 would indicate that the odds of having heart disease ($y = 1$) if exercising ($x = 1$) is only one half the odds of heart disease ($y = 1$) when not exercising ($x = 0$). Thus, the odds ratio gives us a numerical estimate of relatedness between feature x and the outcome y. For this reason the odds ratio has proven to be a powerful analytic research tool when dealing with logistic regression.

2.3.4 Normalizing and Standardizing Data

Before starting any data mining task data pre-processing should be considered as a highly important first step. When dealing with numerical features normalization is a common method that can be used to transform the range of every feature into a certain interval (usually [0,1]). A formula for normalization is given in [24] as:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{2.10}$$

with observation x, the normalized value x_{norm} and the minimum and maximum value for the particular feature x_{min} , x_{max} respectively. Normalization is very useful for regression problems since the magnitude of feature weights can be easily interpreted by looking at their numerical value. A disadvantage of normalization is that it is prone to outliers. Another method that does not suffer from outliers is standardization. Here the goal is to transform the data in a way that it has zero mean and unit variance. From [24] we obtain the standardization formula as:

$$x_{std} = \frac{x - \mu}{\sigma} \tag{2.11}$$

with the standardized value x_{std}, normal distribution parameters mean μ and variance σ. In most cases, mean and variance are usually unknown for a given data set and can be estimated using the sample mean $\hat{\mu}$ and sample variance $\hat{\sigma^2}$. The sample mean can be obtained as the arithmetic mean of the statistical sample, as follows:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.12}$$

where n is the number of observations and x_i is the i^{th} observation from the sample. The sample variance $\hat{\sigma^2}$ can be obtained from the sample as:

$$\hat{\sigma^2} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 \tag{2.13}$$

Here again, $\hat{\mu}$ denotes the mean estimate. $\hat{\sigma^2}$ is biased by a factor of $\frac{n-1}{n}$ and is therefore called *biased sample variance*. If this bias is corrected, the the result is referred to as *unbiased sample variance* and is denoted as:

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \hat{\mu})^2 \tag{2.14}$$

For very large sample sizes using the uncorrected variance estimate is generally accepted, whereas for smaller samples using the unbiased sample variance is favorable.

2.3.5 Cross Validation

When evaluating a classifier there are multiple test options: One way is to *use the training set* for evaluation. The results of this method are not very meaningful, since the classifier obviously performs best on the data it has been trained with. On the other hand, what is of interest is how the classifier performs on unseen data. Another method is to *split* the whole data set into *training* data and *test* data[4]. Here the classifier learns form the training data and can be evaluated on the unseen test data. Drawbacks of this method are however, that if your data set is rather small loosing a part as test data can harm performance of the classifier significantly. An even bigger problem is the spread of the training and test set. If the sets are chosen unluckily the result of the evaluation may not be representative of the classifier's true performance. Furthermore, it is generally hard to determine if these sets are representative or not. To avoid this dilemma we can split the data multiple times into differing training and test sets and perform the whole evaluation procedure multiple times. We can then average the performance to obtain a less biased evaluation result. This is the basic idea behind *cross-validation*. More specifically, for cross-validation the number of times the split-test-evaluate iteration is performed is chosen beforehand and is called the number of *folds*. The data is then randomly split into the number of folds partitions and for each iteration a different partition is chosen as the test data. The remaining partitions form the training data. For example, if the number of folds is set to ten, the data is split into ten equal random sets. Then ten iterations are performed were each time the classifier is trained on $\frac{9}{10}$ of the data and evaluated on $\frac{1}{10}$. This procedure ensures that every instance in the data set has been used exactly once for testing. Finally, the average of the ten iterations yields the classifier's performance estimate.

2.3.6 Weka

For the exploration of the data sets we used the machine learning software tool WEKA, which is developed at the University of Waikato, New Zealand. Witten et al. [25] state that Weka "is a collection of state-of-the-art machine learning algorithms and data preprocessing tools." They argue that "it provides extensive support for the whole process of experimental data mining, including preparing the input data, evaluating learning schemes statistically, and visualizing the input data and the result of learning." The advantage of WEKA is the ease of its use and the access to implementations of the majority of machine learning algorithms. After transforming the data into a WEKA compatible format it is possible, for example, to preprocess the dataset by applying

[4] A common way to split is $\frac{2}{3}$ training and $\frac{1}{3}$ test data.

normalization to it. After choosing a learning method, the data can then be used to train and evaluate the resulting classifier and its performance. All of this functionality is provided in the user interface and can be accessed without writing any source code. Additionally to the user interface, WEKA provides an API to access it directly from Java code which was very useful when integrating the classifier into our system.

2.4 Processing Large Datasets

The advances in digital and mobile communication have lead to the availability of data sets so large and complex, they have become hard to process on standard statistical software [26]. This trend is also amplified by the world's capability to store increasingly larger amounts of digital data [27]. Since these massive datasets require high amounts of computing power and storage space, processing is usually done by spreading the workload between multiple systems. The bigger the cluster of these systems becomes, the higher is the risk of hardware failure and the system's fault tolerance becomes increasingly important. An approach that is scalable and fault tolerant is the MapReduce programming model. The following sub-sections elaborate on the basic concepts of MapReduce and one of its implementations (Apache Hadoop) and a high-level framework (Apache Pig) which is based on the Hadoop implementation. Finally, we introduce the ClueWeb research dataset that consist of about 733 million English web pages.

2.4.1 MapReduce

Dean and Ghemawat [28] introduce the MapReduce programming model which enables users to process and generate large datasets on large-scale computer clusters. These clusters can be made up of commodity hardware, as opposed to highly expensive mainframe computers. The MapReduce system automatically parallelizes the computation among the cluster and takes care of machine failures an inter-machine communication. Inspired by the map and reduce primitives, which are used in functional programming, the authors designed a new abstraction that hides the details of parallelization.

As implied by the name, a user has to specify two functions: a *map* function and a *reduce* function. The map function takes an input key/value pair and outputs a set of intermediate key/value pairs. For each key, the system then groups together all intermediate values and passes them to the reduce function. In the reduce function the user can access each key with an associated list of values to reduce the list to a smaller set of values. Following this model a simple word count program can be written where the map function is called for every document in a collection. The function then emits

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, ``1'');

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Figure 2.6: MapReduce word count problem in pseudo code

every word in the document and its count. In this case, the reduce function's input consist of the key which is a certain word and a list of word occurrences. All the reduce function has to take care of now is going through the list of counts and add them to a global count. Finally the global count's value is emitted. Figure 2.6 presents pseudo code taken from Dean and Ghemawat [28] which implements the word count problem.

2.4.2 Hadoop

Apache Hadoop is an open-source, Java-based implementation of the MapReduce model. It uses the Hadoop Distributed File System (HDFS) to store the data efficiently. HDFS tries to maximize data locality by assigning the workload to the servers in a cluster where the data needed for the tasks are stored [29]. This is done by breaking down the data into small blocks and distribute them throughout the Hadoop cluster. Each node therefore works only on a smaller subset of the dataset which provides the necessary flexibility and scalability for big data processing.

The architecture of a Hadoop cluster is made up of one master node, which takes care of distributing tasks and data to one or more worker nodes. The procedure works as follows: After the client application submits a job to the master node it breaks down the job into multiple tasks which are then distributed among the worker nodes. To take advantage of locality, the tasks are distributed to keep the work close to were the data is located. If a tasks fails or times out it is re-scheduled. In case the task fails multiple times on the same node it is moved to a different node, thereby making the system more fault-tolerant.

2.4.3 Pig

Olston et. al [30] present the *Pig* system for ad-hoc analysis of massively large datasets and its accompanying language *Pig Latin*. The authors identify the main weakness of the MapReduce model as being too rigid, since it strictly allows for only one input and two-stage data flow. Their proposed language is a mixture of "the spirit of SQL, and low-level, procedural programming". Pig Latin programs are made up of a sequence of steps, where each step is a single transformation of the data. Therefore, "writing a Pig Latin program is similar to specifying a query execution plan", which helps the developer to better understand the data flow. Pig Latin also supports custom data processing needs by letting the user create so called *user-defined functions* (UDFs). A UDF can customize all aspects of data processing step in Pig Latin. The functions follow Pig's fully nested data model by taking non-atomic parameters as input and output non-atomic values. To achieve a high amount of parallelism, the Pig platform is built on top of Hadoop and translates each Pig script into a sequence of MapReduce programs.

2.4.4 ClueWeb

As closure of this chapter, we present the ClueWeb [31] dataset, which is a large research dataset that consists of 733,019,372 English web pages. These web pages where collected as part of a web crawl between February 10, 2012 and May 10, 2012. In addition to the crawling procedure, web pages where filtered and organized into a format that is advantageous for research. The dataset captures only raw text from these web pages and ignores multimedia content, e.g., audio or video files. Handling the size of the data has proved to be a challenging task for researchers and requires the use of scalable systems like MapReduce. However, since it contains large amounts of text, it has become a valuable linguistic resource.

Chapter 3

Related Work

To the best of our knowledge, there exists no prior work which encompasses all research topics that we address in this paper. Therefore, we discuss related work divided into the following topics in this chapter: The first section elaborates on efforts in the field of question generation (Section 3.1). There, we examine knowledge-graph-based, text-based and crowdsourcing approaches to generate fill-in-the-blanks questions, multiple choice questions or type-based question templates. In the second section we focus on estimating the difficulty of a question (Section 3.2). One of the approaches utilizes a community answering service (Stack Overflow) to infer hardness of a question based on competition between users. The second part of this section focuses on work that estimates reading difficulty for natural language text. Section 3.3 highlights techniques for the verbalization of queries. We discuss approaches that verbalize SPARQL and SQL statements to guide users when formulating queries in these languages. In the final section we present work that focuses on the analysis of Jeopardy! questions (Section 3.4), which originated as part of the Watson project, developed by IBM.

3.1 Question Generation

Following Rus et al. [32], the task of question generation is defined as the automatic generation of questions from various input sources. Sources can be raw text, a database or some form of semantic representation. The authors identify two core aspects for question generation: the question's goal and its importance. They further argue that the "goodness" of a question can only be determined by looking at the context the question was posed in. Thus, it is required to find information about the question's goals and what counts as important regarding the current context. While examining related work, we found that in practice many proposed approaches cannot be strictly categorized by

a single input source, since these papers make use of a combination of various input sources.

Sakaguchi et al. [33] focus on the problem of removing words from a sentence to create fill-in-the-blanks quizzes for language learning. For the removed words they create distractors[1] and evaluate them in terms of *reliability* and *validity*. A distractor has to be reliable; meaning that it cannot be replaced with the answer, thereby avoiding multiple correct answers to a question. Furthermore, the distractors have to be valid; meaning that they are "close enough" to the correct answer, such that they distract the learners that do not know the correct answer. Their proposed method first finds the word to be left out by looking at error-correction pairs extracted from a large English learner corpus and selecting the verbs where a *semantic confusion* was made. Then, they calculate the conditional probability $P(w_e|w_c)$ that a word w_c is misused as w_e and compute a confusion matrix based on these probabilities. Given a sentence, the verbs appearing in the confusion matrix are identified and made blank. To generate the distractors, the authors train multiple classifiers for each target word using the error-correction pairs. These classifiers are based on the discriminative Support Vector Machine model and are trained by looking at 5-gram lemmas and their dependency types with the target word. Each trained classifier for a target word works by taking a sentence as input and outputting a verb as the best distractor given the 5 word context. Finally, the approach is evaluated in terms of effectiveness, by conducting a user study with English native speakers and comparing the ratio of appropriate distractors with two baselines. They show that their discriminative models perform better than their baselines that use a generative model. Furthermore, they show the validity of their distractors by measuring high correlation between the performance of non-english speakers on a test generated by their system and the participant's TOEIC[2] scores.

Narendra et al. [34] propose an end-to-end system for the automatic generation of fill-in-the-blanks questions from a given text. Their method retrieves a text document from the Cricket domain as input and outputs a sentence with a blank and four answer options. One of the options is the correct answer, whereas the remaining three options are distractors. For a given document, their approach performs three stages of processing until the question is generated: In the first stage a relevant and informative sentence is selected to represent the question's sentences. For this task the authors use an off-the-shelf extractive summarizer and use the top ten percent of the summarizer's output. In the second stage, their approach selects keywords that are used as the blank in the question. Keywords can be either named entities, pronouns or constituents. Additionally, they define a list of observations to help prune the list of candidate keywords, which

[1] A distractor is an incorrect option in a multiple choice question.
[2] https://www.ets.org/toeic

encompass relevancy of a token and the position of the preposition, among others. In the final stage, the researchers use an approach backed by a knowledge graph to generate the question's distractors. The knowledge graph is only involved in distractor generation when the selected keyword is a named entity. In the case that a named entity is not a person, their algorithm selects a fact from the knowledge graph at random. In case that it is a person, the algorithm selects facts depending on the Cricket team the person plays in. This helps to generate distractors of players whose properties are close to the answer's properties.

The approach introduced by Labutov et al. [35] focuses on generating high-level comprehension questions rather than factoid questions. What distinguishes their approach is that it does not require the system to deeply understand the text, as the generation of question templates is accomplished by crowd workers. The approach generates questions by representing the source text in an ontology. The ontology is built as the Cartesian product of Freebase article categories and article section names, derived from Wikipedia. The authors call these mappings *category-section* pairs. For instance, the category *Person* and the section *Early life* form such a pair. Using these pairs from the ontology, crowd workers are asked to create high-level templates in the next step. For the above mentioned category-section pair a crowd worker may create the question template *Who were the key influences on <Person> in their childhood?* To ensure the generated questions are high-level and relevant, the authors build a classifier that ranks each question according to its relevance to the given text.

A method that utilizes semantics provided by an ontology was introduced by Al-Yahya [36] as the OntoQue engine. The author's system is backed by an ontology with roughly 300 RDF triples to generate multiple-choice, true/false, and fill-in-the-blank questions. OntoQue generates questions by iterating over RDF statements that contain entities, such that every statement can be turned into a single question. RDF triples that are not meaningful for questions are sorted out. Fill-in-the-blank questions are generated by leaving out either the subject or the object of a triple. For true/false questions either the subject or object is replaced by an entity belonging to the same class as the entity of the correct answer. Distractors for multiple-choice questions are generated by either considering entities that share the same class-membership as the answer or enumerating all individuals in the knowledge graph and collecting all assertions where the individual is either subject or object. Furthermore, the author utilizes the `rdfs:label` property to access the surface form for an entity. The system was evaluated by the author, by categorizing the generated questions as *good* or *bad*, and measuring precision.

3.2 Difficulty Estimation

In this section we discuss related work in two problem domains related to the estimation of difficulty in language. The first domain deals with the prediction of question difficulty in the context of community question answering services, such as StackOverflow[3] and Yahoo! Answers[4]. The second domain deals with the prediction of reading difficulty of natural language text. Even though approaches of said domain do not deal specifically with questions, the discussion of this work gives insights into the prediction models used for the estimation of difficulty in language related problems.

Liu et al. [37] addresses the problem of estimating question difficulty in community question answering services. They use a competition-based approach, which models question difficulty by taking the user expertise level into consideration. In their work they make two assumptions: First, the difficulty of a certain question is higher than the expertise score of it's asker. Second, the user's expertise, who has given the best answer, is higher than that of the asker and all other users who gave lower ranked answers. Question difficulty is then determined by looking at the pairwise comparisons for a "two-player" competition with one winner and one loser. Competitions can be of the following kind:

- competition between question and the question's asker
- competition between the question's asker and the best answerer
- competition between the best answerer and the question's asker
- multiple competitions between the best answerer and all other answerers

Now, the problem of estimating question difficulty can be cast into the problem of learning the relative skills of each player by looking at the results of the two-player competitions. If we regard the question as a participant in the competition, the question's difficulty can then be retrieved as it's skill score. Skills scores learned for all other users reflect their expertise scores. To learn the relative scores, the authors adapt the *TrueSkill* ranking model [38]. To evaluate the approach of Liu et al. [37], 300 question pairs are sampled from StackOverflow and experts are asked to compare their relative difficulty. Then, the authors measure the accuracy of their system as the number of correct pairwise comparisons divided by the total number of pairwise comparisons. Finally, their method is compared to a PageRank-based approach [39], where the difficulty of tasks in crowdsourcing contest services is estimated. The approach models the problem as

[3] http://stackoverflow.com
[4] http://answers.yahoo.com

a graph, where an edge between two tasks encodes that one task is harder than the other. Then they interpret the PageRank score of each task as the difficulty measure. In Liu et al. [37], the authors find significant performance improvements compared to the PageRank-based method, in terms of accuracy.

There is a body of work done in estimating reading difficulty of texts. Therefore we place emphasis on the most related approaches in this field. Collins-Thompson et al. [40] created an approach which uses statistical language models to asses reading difficulty. The method uses a smoothed unigram language model based on a variation of the multinominal naïve Bayes classifier. The semantic difficulty of a given text T is predicted as the likelihood that T was generated by a language model that is representative for a certain school grade level. These language models are trained from authoritative sources and educational websites that have grade levels assigned to them. Their work shows that particular words are very decisive for a certain grade level. For instance, the authors found that the words *grownup*, *ram* and *planes* where most representative for grade level 1, whereas the words *essay*, *literary* and *technology* were most indicative for grade 12. One drawback of their method is that it considers lexical features only and does not incorporate features based on grammar. In contrast, Heilman et al. [41] present work that shows how reading difficulty estimation can be improved by considering a combination of lexical and grammatical features. In their approach, the authors consider the relative frequencies of a set of morphologically stemmed word unigrams, which constitute the lexical features. As grammatical features, the approach computes the relative frequencies of sub-trees of syntactic parse trees up to a certain depth. Using these features the researches experiment with three linear and log-linear models, namely *linear regression*, *proportional odds model* and *multi-class logistic regression*. These models were evaluated on documents of a web corpus, where each document had a grade level assigned to it. As a result, they found that the proportional odds model performs best for predicting reading difficulty.

3.3 Query Verbalization

Query verbalization is a sub-problem of natural language generation (NLG). NLG often finds application in dialog systems, where a novice user that is not familiar with the format of the underlying data, expects some sort of human interpretable representation. In the context of query verbalization, the main purpose is to assist users formulating queries by translating their query into natural language. This enables users to compare their query's reflection of their information need more easily, since a textual representation of the query is assumed to be interpretable by humans more effortlessly.

An approach to verbalize SPARQL queries into natural language was proposed by Ngomo et al. [42]. Their approach is intended to assist novice and professional users in creating SPARQL queries, by making the assumption that users better understand their query if it is written in natural language. Their approach works by using three steps (preprocessing, processing and postprocessing) to verbalize and refine the natural language representation incrementally. In the preprocessing step, a combination of types is assigned to each projection variable in the query. The query is then normalized by transforming all nested `UNION` statements into disjunctive normal form. In the processing step, each triple pattern is verbalized separately by a rule based approach, which maps each triple to a sequence of Stanford dependencies. Furthermore, the `WHERE` clause of the query is verbalized by transforming the extracted types from the preprocessing phase elements using a set of pre-defined rules. In the postprocessing step, the verbalization is transformed into a more natural phrase by identifying and removing redundancies. To evaluate their approach, the authors implemented a prototype called SPARQL2NL and performed a user study. Participants where asked to evaluate the verbalization according to the machine translation metrics *adequacy* and *fluency*, introduced in [43]. They also measured the task completion time and inferred that their approach reduced the time for experts and novices alike. Finally, they compared their approach to Ell et al. [44] and showed an improvement in terms of adequacy and fluency.

Koutrika et al. [45] presents an approach for the verbalization of SQL queries. Their method represents structured SQL queries as directed graphs, where edges are annotated with extensible template labels to capture the semantics of a query. Each edge can represent one of the following relations: (1) *membership* of an attribute to a relation, (2) a *predicate* or (3) a *selection*. After building the query graph, the natural language phrases are composed by different graph traversal strategies (namely, InAF and MRP). In *InAF* the algorithm starts from the query subject and performs a depth-first search through the query graph until all relations have been reached. *MRP* also traverses the query graph in depth first search, but the actual direction of the translation is chosen according to the current state of the algorithm. Furthermore, their work also encompasses an algorithm for selection of the best templates for a given query, by building the best combination of automatically created and manually specified templates on-the-fly. Finally, the authors measure the effectiveness of their method by selecting two teams of SQL experts which evaluate a set of automatically generated query explanations. One expert team is asked to write a query explanation, whereas the other expert team evaluates the performance of each algorithm, by comparing the output with the user generated content.

3.4 Jeopardy! Question Analysis

As part of IBM Research's effort to push the boundaries in deep question answering, the company accepted the challenge to build a system that could answer questions of the American TV quiz show Jeopardy! in real time. Their work resulted in the creation of the *Watson* system, which was able to automatically answer Jeopardy! questions at a level competitive with human champions [1]. A large part of the work focused on analyzing Jeopardy! questions. Lally et al. [46] perform an extensive effort to analyze Jeopardy! questions in order to find the best approach for answering them. They state that it is required to find the focus of a question, which is the reference to the answer. Consider the example question:

> *He* was a bank clerk in the Yukon before *he* published "Sons of a Sourdough" in 1907.

Here, the focus is the pronoun *he*. Finding the question's focus is essential in later stages to align the question with a supporting passage in a document. Furthermore, they propose an approach to detect lexical answer types (LATs). An LAT is a term in the question that indicates the type of the entity that is being asked for. For instance, the question:

> Henry VIII destroyed the Canterbury Cathedral Tomb of this *saint* and *chancellor* of Henry II.

indicates that the answer entity is of type *saint* and *chancellor*. These LATs are used in Watson to determine if the answer type matches the types of multiple answer candidates, thereby eliminating entities with non-matching types. In addition to focus and answer type detection, they build a question classifier that can determine to which *question category* a question belongs. They identify seven broad question categories: *factoid*, *definition*, *multiple-choice*, *puzzle*, *common bonds*, *fill-in-the-blanks* and questions about *abbreviations*. Identifying these question classes is an important step for later stages in the answering process, since it determines which answer strategy or machine learning model is used. They find, that even though all questions of all categories appear frequently in Jeopardy!, the major percentage of questions are *factoid*. Questions are factoid when their answer is based on factual information about one or more individual entities[5]. Kalyanpur et al. [47] focus on these factoid questions and decompose them into atomic parts, where each part contains only one fact. In Jeopardy! it is quite common that a single question contains multiple facts. Consider for example the question:

[5]Since these questions are by far the most common, in our work, we focus on factoid questions only.

> This company with origins dating back to 1876 became the first U.S. company to have 1 million stockholders in 1951.

This question contains two facts: (1) *a company with origins dating back to 1876* and (2) *a company that became the first U.S. company to have more than 1 million stockholders in 1951*. The authors hypothesize that decomposing the question into single facts can help in the evidence collection process. The hypothesis is based on the assumption that it is more likely to find evidence supporting a single fact, than evidence that supports both facts at the same time. The work focuses on two types of decomposable questions: *parallel* and *nested* decomposable. The example question, stated above, is categorized as parallel decomposable, since its facts are mutually independent and relate to the answer entity. However, decomposable questions contain facts about an entity related to the correct answer and a separate fact that links the entity to the correct answer. The question:

> A controversial 1979 war film was based on a 1902 work by this author.

can be decomposed into two separate clues. The first clue asks about Francis Coppola's *Apocalypse Now*, which is related to the question's answer. It basically serves as a side clue. The second clue given, asks about the novella *Heart of Darkness* by Joseph Conrad, who is the actual answer to the question.

Chapter 4

Question Generation from Knowledge Graphs

In this chapter, we present our approach for using factual knowledge from the YAGO knowledge graph to automatically generate questions. Our system expects a difficulty level (easy or hard) and a topic (e.g., sports) as user input. The system's output will be a question in natural language, that abides the input properties. On a high level, the system can be split into three separate units: The first unit is responsible for the selection of the question content, meaning: What/Who is the question's target (the answer) and which clues are given to the user (a subset of facts about the answer). Furthermore, the unit checks if the generated question has a unique answer. The second unit estimates the difficulty of a question given the information selected by the first unit. It uses a binary logistic regression classifier trained on the Jeopardy! question-answer corpus, with given difficulty judgments. The third unit verbalizes the "raw" facts to make them legible for the users. This is done using a template that mimics Jeopardy!-style questions and makes use of handwritten and automatically created lexical resources to ensure linguistic variation.

A working version of the system was demonstrated at the poster track of the *2015 International World Wide Web Conference*, in Florence, Italy. The actual poster paper, shown at the conference, can be inspected in Appendix A. Further details about the poster can be found in Seyler et al. [48].

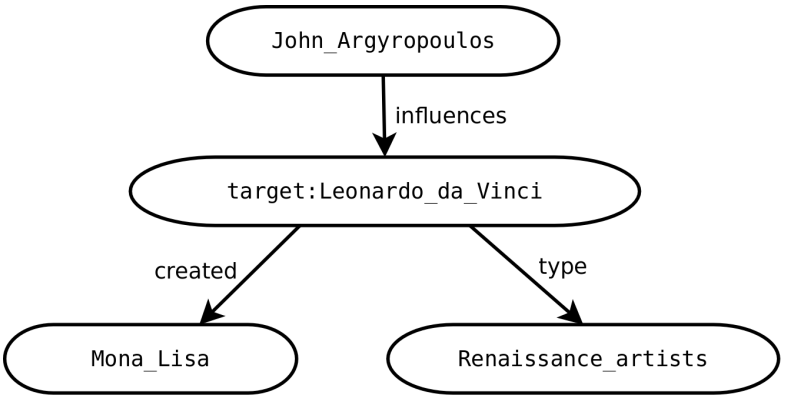

Figure 4.1: Question graph example

4.1 Generation of Question Graphs

The first step of our approach is the generation of a question graph. A *question graph* is a set of facts and a corresponding target entity (the question's answer), which resembles the semantic representation of the question. An example of a question graph is depicted in Figure 4.1, for illustration purposes. An English translation of the graph could be: *This Renaissance artist created Mona Lisa and is influenced by John Argyropoulos.* There, the target entity (`Leonardo_da_Vinci`) is the answer to the question. Because question generation aims at generating quiz questions, we chose the following constraints for each question graph:

- The answer to the question should be a single entity (e.g., Leonardo da Vinci).
- The answer entity should be relevant to the given topic (e.g., Renaissance).
- The question should contain at least 3 but no more than 5 facts to avoid information sparseness or exuberance.
- The entities mentioned in the question's clues should not contain tokens that overlap with the answer entity (e.g., A fact asking about Leonardo da Vinci's birthplace Vinci, Florence should be filtered out).
- The question should specify a meaningful type for the answer entity (e.g., Renaissance artist).
- The question should contain a single unknown variable only, which is the answer to the question. From this follows, that we restrict the generation process to "star"-queries.

To generate a question graph, the algorithm starts by selecting a target entity, which will be the answer to the generated question. In the following step, the system ensures that the entity is semantically related to the given topic. Furthermore, the entity needs to have some degree of popularity. This is important, since the set of entities contains a large amount entities which are too obscure for testing the user's knowledge of a particular topic. In the second step, a subset of facts from the knowledge graph is selected which represent the clues of the question. Additionally, the algorithm ensures that the question graph always contains at least one type and at least one non-type relation which is a requirement for the natural language generation step.

4.1.1 Selection of Target Entities

As mentioned before, one challenge of our system is to generate questions for a given *topic*. This process can be translated to finding a set of entities that corresponds to the provided topic. To address this problem, we make use of the Wikipedia category structure. We use DBpedia as a proxy to gather the information about Wikipedia categories and use it in combination with the YAGO dataset to construct the *Wikipedia category graph* . The graph contains a mapping of all Wikipedia categories to their corresponding YAGO entities. For any given category, we perform a breadth-first search through the category graph visiting all subcategories and collecting their entities up to a certain depth.

4.1.1.1 Wikipedia Category Graph

The Wikipedia category graph contains the Wikipedia category structure and the category membership of every YAGO entity, associated with it. The data structure of the graph is represented as two mappings: One mapping contains the subcategory-to-category pairs (the category structure), while the second mapping contains the categories and their YAGO entities. The Wikipedia category structure can be obtained from one of the DBpedia datasets. The data contain RDF-triples with the relation `<core#broader>` which indicates that subject category is a subclass of the object category. For example, `Category:Italian_Renaissance core#broader Category:Renaissance` states that the category *Italian Renaissance* is a subclass of the category *Renaissance*. In addition, we obtained the mapping between Wiki categories and the DBpedia entities they contain. In the final step we translated DBpedia to YAGO instances, to generate the mapping between categories and their entities. An illustrating example of the resulting graph structure can be found in Figure 4.2. In the example, the elements marked in bold-blue represent the category structure, whereas the elements marked in black, with rounded corners, represent the entity-category mapping.

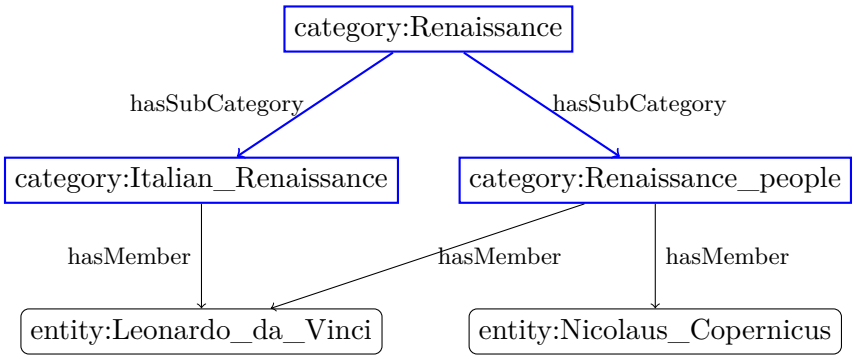

Figure 4.2: Example of Wikipedia category graph

Using the resulting graph, we can obtain all entities for a given category by traversing the graph in a breadth-first manner, until we reach a certain depth. The depth of the category graph needs to be limited, since the Wikipedia category structure is prone to semantic drift. This can be attributed to the fact, that for some categories a more general category is a subcategory to a more specific category. As a concrete example, we can find a path from the category *Computer Science* to any person listed in Wikipedia by traversing the graph as follows:

```
Category:Computer_science
hasSubClass Category:Areas_of_computer_science
hasSubClass Category:Artificial_intelligence
hasSubClass Category:Problem_solving
hasSubClass Category:Abstraction
hasSubClass Category:Identity
hasSubClass Category:National_identities
hasSubClass Category:People_by_nationality
```

For some categories, e.g. *Computer Science*, semantic drift causes the retrieval of all entities in Wikipedia and therefore the graph can only be traversed up to a certain depth before a category's entities become to unrelated to the chosen topic. In our experiments we found that a depth of *three* generally provides the best trade-off between the retrieved entities relatedness to the topic and the coverage of a given topic.

4.1.1.2 Criteria for Choosing the Target Entity

After retrieving all entities for a chosen topic, it is necessary to find a principled way to choose the target entities. We developed two approaches to achieve this: The naïve variant chooses an entity uniformly at random, whereas the more informed variant chooses a target entity according to its popularity. Problems with the naive method arise, since the set of entities for a given category can be very large and a major percentage of this set are obscure. These entities are typically unheard-of and are usually referred to as "long tail" entities. To deal with this common problem, we decided to choose an entity according to its popularity[1]. In our approach the entity's likelihood of being chosen is proportional to its popularity. The probability that entity e is chosen ($P(e)$) can be expressed by the following equation:

$$P(e) = \frac{pop(e)}{\sum_{i=1}^{E} pop(e_i)} \tag{4.1}$$

where $pop(e)$ is the popularity measure and E is the set of all entities. As a result, more popular entities are more likely to be chosen than less popular entities, which makes occurrences of obscure entities less likely.

Another problem that arose was that entities may be popular globally, but not within a certain category. For example, the category *Grammy Award Winners* has *Barack Obama* as its most popular entity. This is true, because he won a Grammy Award for an audio book. Since all other entities in the category are less popular, he becomes the category's most prominent entity, even though a musician might be a better fit. Thus, when calculating the popularity for an entity it is not sufficient to look at its general popularity only. It is required to measure the popularity within the chosen category. Naturally, we call this *category popularity* which is defined in Section 4.3.1.1. After substituting the general popularity with the category popularity, we were able to retrieve entities that are better suited to asses the knowledge of a given topic.

4.1.2 Selection of Facts

When the target entity is selected, we need to ensure that the facts selected for the cues in the question fulfill various properties. Certain facts in the knowledge graph would give away part of the answer, which would have significant impact on question difficulty. Therefore, Section 4.1.2.1 deals with removing facts that have a significant token overlap with the answer. Also, it should be ensured, that each additional clue

[1] A definition of popularity can be found in Section 4.3.1.1.

adds meaningful information to the question. Repetition of facts and redundant types should be avoided. To address this problem we incorporate the type taxonomy (Section 4.1.2.2). Additionally, we selected a set of top-most meaningful classes and we disregard all superclasses in the hierarchy. This is required, since classes at the top of the class taxonomy are too abstract and therefore unsuitable for question generation. Section 4.1.2.3 discusses this issue in more detail.

4.1.2.1 Elimination of Entities with Token Overlaps

When selecting the triples that make up the question's cues, we avoid triples that would give away the answer in any way. This problem occurs when an entity of a newly selected fact has some sort of overlap with the answer entity. For example, this is fairly common in the case of family members, that share a last name. To account for this, each entity's tokens are cross-checked for non-stopword overlaps with the target entity, before choosing a new fact to add to the question graph. As mentioned before, this is required since an entity could reveal part or all of the answer. For example, the question *Which soccer player is married to Victoria Beckham?* reveals the answer entity's (`David_Beckham`) last name . As an example for full revelation of the answer the question *Which rock band created the album Pearl Jam?* contains the all of the answer's tokens (`Pearl_Jam`). In addition, we decided to allow for overlaps in stopwords, since they do not give away information about the answer. However, this approach has its limitations. If an entity is made up entirely of stopwords, our method would not be able to identify this. A common example is the popular rock band, that is simply named *The Who*. Solving this issue is an open problem and is not further discussed in this thesis.

4.1.2.2 Consideration of Class Hierarchy

Triples that contain information about the target entity's classes can contain redundant information. The redundancy occurs when two types about the same entity form a class-subclass relationship. In that case, all superclasses of the most specific type can be ignored, since they do not add useful cues to the question. For instance, consider a question about the soccer player `Ronaldo`. First, the type `soccer_player` is added to the question content. At this point, selecting a type which is a superclass of `soccer_player` will not add any conducive information to the question. In our example, this can be exemplified when adding the type `athlete`. It is already stated that the target entity is a soccer player and through common sense a user can infer that he is an athlete, as well. Therefore, `athlete` can be disregarded. Even though types are selected at random, the algorithm developed as part of our approach guarantees that only the most

Excluded Class	Meaningful Class
wordnet_abstraction_100002137	<wordnet_person_100007846>
wordnet_causal_agent_100007347	<wordnet_organization_108008335>
wordnet_group_100031264	<wordnet_artifact_100021939>
wordnet_life_100006269	<wordnet_event_100029378>
wordnet_living_thing_100004258	<wordnet_location_100027167>
wordnet_object_100002684	
wordnet_organism_100004475	
wordnet_physical_entity_100001930	
wordnet_psychological_feature_100023100	
wordnet_social_group_107950920	
wordnet_thing_104424418	
wordnet_whole_100003553	
yagoGeoEntity	
yagoLegalActor	
yagoLegalActorGeo	
yagoPermanentlyLocatedEntity	
owl:Thing	

Table 4.1: Excluded and top-most meaningful classes

specific type is kept. Before selecting the next type to be added to the question, the algorithm constructs a taxonomy tree that contains all types that are currently present in the question. To retrieve the most specific type, every type that is a superclass of any leaf node is pruned. Consequently, for types: `person`, `athlete`, `soccer_player`; only `soccer_player` is kept as part of the question.

4.1.2.3 Exclusion of Classes and Predicates

Some classes and relations are not suitable for question generation and have to be removed manually. We decided to exclude classes that we consider to be too general, since we found that they are too abstract to result in useful clues in the question. As an extreme example, consider the class `owl:Thing`. Since all entities are things, adding this type information does not result in meaningful question cues. We selected a set of top-most meaningful classes and disregarded any class above in the hierarchy. The right column of Table 4.1 presents the highest-level classes, which we considered to be meaningful. The excluded classes, which are considered to be more general, are depicted in the first column of said table.

In addition to class concepts, we found that certain relations are unfit for the question generation task, as well. Most of these relations are used to represent technical information about an entity and therefore have little value, when used as a question's cue. For instance, the predicate `hasWikipediaArticleLength` provides information about the length of the

```
SELECT ?x WHERE {
        ?x created Mona_Lisa .
        ?x type Renaissance_artists .
        John_Argyropoulos influences ?x
        }
```

Figure 4.3: SPARQL query for Figure 4.1

entity's Wikipedia article. Since it is not reasonable to expect any person to know this exact number, relations of this kind can be ignored. To solve this issue, we disregarded all triples in YAGO that represent meta-data about articles.

4.1.3 Query Generation & Uniqueness Check

After the selection of facts is completed, we have determined the question's answer and context. As described above, that information is represented as a question graph. In the next step, the question graph is transformed into a SPARQL query, which will enable us to query the knowledge graph in later steps. We considered SPARQL to be a natural choice for our purpose, since it is the standard language for querying RDF data. The query for retrieving the target entity as an answer to the query can be constructed as follows: The query is of type `SELECT`, so the outer statement is `SELECT ?x WHERE { ... }`. The variable *?x* is used to retrieve the answer set to the query. The more interesting part is the body of the `WHERE` statement. Here, we insert statements about the triples we selected in earlier steps of the question generation process. Each triple we selected has to be transformed into an intersection statement with the target entity as the variable. Depending on the position of the target entity we select either the subject or the object of a triple as the variable. For instance, in the triple

```
Leonardo_da_Vinci created Mona_Lisa
```

would be turned into

```
?x created Mona_Lisa
```

since the `Leonardo_da_Vinci` is the answer to the question. In the above mentioned triple he is the subject and therefore it has to be replaced with a variable. Similarly, the object is replaced when it resembles the target entity. Finally, the full SPARQL statement for Figure 4.1 is shown in Figure 4.3.

As a final step in the chain of question generation tasks, we need to confirm that the answer to the generated SPARQL query is unique. This step is required, since our goal is

to generate questions that have only a single entity as the answer. This can be achieved by querying the `SELECT` statement, that was constructed in the previous step, to the knowledge graph and measuring the size of the query's answer set. If this query returns more than one result, the selected facts for this target entity are not unique and we have to start afresh. In case the result set only contains the target entity, we assume that a unique set of cues was found and we output the query as a valid question.

4.2 Query Verbalization

At this point of the question generation process we have successfully selected a target entity for the given category and we have selected a set of corresponding facts about the target entity. This set has the properties that it is a unique combination of facts that yields the target entity as its sole result, when queried against the knowledge graph. Furthermore, we have turned the representation of these triples into the SPARQL query format. Since our ultimate goal is to create a system that generates questions in a quiz setting, our system has to generate output that is understandable for human players. In our current state, the question is in a format that is only interpretable by experts that are familiar with the SPARQL language, and therefore needs to be "translated" into natural language. In computer science literature, Reiter and Dale [49] state that computer systems that are able to generate texts in a human language are considered as natural language generation (NLG) systems. As a sub-field of artificial intelligence and computational linguistics, research in NLG is concerned with the verbalization of the underlying non-linguistic representation of information into a human understandable format (natural language). In our setting, we create a query verbalization scheme which allows the system to output natural language. Therefore, our system falls in the category of NLG systems. The following section elaborates on our approach, which uses template-based methods to transform SPARQL queries into natural language.

```
SELECT ?x WHERE {
   ?x actedIn Saving_Private_Ryan .
   ?x graduatedFrom Chabot_College .
   ?x type actors
}
```

Figure 4.4: Example SPARQL query about *Tom Hanks*

4.2.1 Verbalization Approach

For a given SPARQL query, we use a template-based approach to generate the natural language representation. Since we pose our system in a quiz game setting, we defined the template such that generated questions reflect the *Jeopardy! clue* style. In Jeopardy!, questions are called *clues* and are expressed as statements. Similarly, the answer is expressed as a question. For example, in Jeopardy! the question:

> *This fictional private investigator was created by Arthur Conan Doyle.*

has the answer:

> *Who is Sherlock Holmes?*

As stated above, our intermediate SPARQL query contains at least one type and one non-type triple. This enables us to verbalize the query using the following the simple pattern:

$$\textit{This } \texttt{type}_1, \ldots, \textit{ and } \texttt{type}_m\ \texttt{p}_1\ \texttt{o}_1, \ldots, \textit{ and } \texttt{p}_n\ \texttt{o}_n\ . \tag{4.2}$$

Here, each $\texttt{type}_i$ is obtained as the object of the type triples and $\texttt{p}_i\ \texttt{o}_i$ are the predicates and object entities of the remaining triples. For verbalization, we use the surface form of each type and turn it into singular. Also, for object entities we use their canonical surface form, as captured in the knowledge graph (e.g., *David Beckham*).

To verbalize predicates, we constructed a dictionary of paraphrases for every relation (Section 4.2.2). Each relation has at least two paraphrases, one for the case when the target entity is the subject and another for when it is the object. Thus, the `playsFor` predicate can be verbalized as *plays for* or *has player* depending on whether our target entity is `David_Beckham` or `Real_Madrid`, respectively.

4.2.2 Paraphrasing Relations

This pattern-based approach yields acceptable results, considering that its a relatively simple approach. For instance, consider the example query about `Tom_Hanks`, depicted in Figure 4.4. With our approach, the query is verbalized as *This actor acted in Saving Private Ryan and graduated from Chatbot College.* If we exchange the entities in the question, the verbalization hardly changes: *This actor acted in Apollo 13 and graduated from California State University.* This is due to the fact that relations are verbalized in a static way, where there exists only one paraphrase depending on whether the target entity

Relation	Paraphrase
`actedIn`	*acted in* *starred in* *appeared in* *played in*
`graduatedFrom`	*has graduated from* *is a graduate of* *studied at* *is an alumnus of* *holds a degree from*

Table 4.2: Two relations and their paraphrases

is a subject or object in the fact. To cater to variety, we mined paraphrases for each relation from the ClueWeb[2] corpus. These paraphrases where extracted by considering entity pairs, that are already contained in the knowledge graph, and by capturing the text between those entities. For example, the fact `Tom_Hanks actedIn Forrest_Gump` could be found ClueWeb text as: *The actor* `Tom_Hanks` *starred in* `Forrest_Gump` *which grossed a worldwide total of over $600 million at the box office.* Here it should be noted, that in a pre-processing step, mentions of YAGO entities in the ClueWeb dataset had to be annotated. From the example text, we can extract *starred in* as a paraphrase for the relation `actedIn`. From this list of paraphrases we hand select a set that fits into our verbalization scheme. Table 4.2 shows some example paraphrases for the relations `actedIn` and `graduatedFrom`. After applying these paraphrases we can alternatively verbalize the above mentioned SPARQL query as: *This actor starred in Saving Private Ryan and received a degree from Chatbot College.*

4.2.3 Finding Salient Types for Entities

In addition to paraphrasing relations, we present an approach for selecting important types for an entity. Finding the important types for an entity can assist the user when disambiguating the entity mentioned in a question. Consider, for example, the question *This artist created Daughter.* At the current state of the verbalization it is impossible to constitute what *Daughter* means. It could be any artifact: a painting, an album, a song, etc. Let's say in our example we were talking about the *Pearl Jam* song *Daughter*. Stating the type of the entity in the question would be essential to guide the user towards the right entity. Our goal of this task is to be able to express the original query as *This artist created the* ***song*** *Daughter.*

[2]The ClueWeb corpus is introduced in Section 2.4.4

Textual Type	Count	Textual Type	Count
artist	884	painting	108
case	236	person	95
master	225	masterpiece	87
work	219	thinker	86
people	214	score	82
kinda	201	example	78
bloomers	199	figure	78
genius	175	inventor	72
man	118	painter	70
ear	111	some	59

Table 4.3: Textual types and number of occurrences for Leonardo da Vinci

Our approach works as follows. We use ClueWeb as our primary source of data and consider the mentioning of a certain type in context with the entity at hand as evidence that this type and entity are a "good match". More concretely, we mine these textual types from ClueWeb using Hearst patterns [50] and disambiguate the entity, mentioned in the text, and map it to YAGO. In a subsequent step, we map the extracted types to the types in the knowledge graph. For simplicity, we consider only the headword of the extracted type. For instance, consider the sentence *Leonardo da Vinci is widely considered to be one of the greatest and most-talented painters of all time*, to be contained in the ClueWeb corpus. The textual type we extract would be *greatest and most-talented painter*. By looking at the headword (*painter*), we enhance the chances to find an overlap with the classes in our knowledge graph[3]. Now, it is possible to obtain a ranking, as the number of mentions of textual types in the corpus. Here, we implicitly assume, that the number of mentions correlates with the "importance" of a certain type. The top-20 textual types for `Leonardo_da_Vinci` with their occurrence count can be observed in Table 4.3. In the table it can be observed that the textual types are quite noisy. Therefore, as a filtering step, we map the extracted types to the types we already know about the entity in the knowledge graph. To expand the set of types in our knowledge graph, and to increase the chance of finding an overlap with an extracted type, we lookup synonyms in WordNet. For instance, for the WordNet class `discoverer` we find the synonym `inventor`, which maps to the extracted, textual type *inventor*. As a result, Table 4.4 depicts the salient types for `Leonardo_da_Vinci`, extracted by our method. The types seen in the table where mapped to the YAGO classes when considering the top-25 textual types, which where found in the ClueWeb corpus.

[3]Painter would be an exact match to WordNet class *painter*

Salient Type
wordnet_artist_109812338
wordnet_person_100007846
wordnet_scientist_110560637
wordnet_inventor_110214637
wordnet_painter_110391653

Table 4.4: Salient types in knowledge graph for Leonardo da Vinci

4.3 Estimating Question Difficulty

One essential property of the system is how well it measures the difficulty of a question. Standardizing difficulty is a challenging task, since question difficulty depends on multiple factors of the individual who is supposed to answer the question. Strictly speaking, it requires measuring the average, personal difficulty of a group of people, that is of statistically relevant size. For this reason, we choose a gold standard for question difficulty, created by quiz experts, that is representative for a large amount of the general population. We decide that questions of the TV series Jeopardy!, which is a popular American game show, fulfill our desiderata for a gold standard. In the show, general knowledge questions are asked in a broad variety of domains. These questions are associated with monetary values which represent the question's difficulty, as assessed by quiz experts. The show started in the 1980s and has new shows on a daily basis, thus the corpus of questions is sufficiently large. Furthermore, there is an online community (J! Archive[4]) that provides question-answer pairs and other information to almost every show that has been broadcasted.

Even though labeling a question as more difficult than another can be subjective, some questions do appear harder than others. In Jeopardy! it can be observed that the higher the difficulty of a question, the less people are able to answer it correctly. To demonstrate this we used a set of 264,548 Jeopardy! questions and measured the correlation of the number of times the question could not be answered (in Jeopardy! this event is called "Triple Stumper") and the monetary value of the question. The result can be observed in Figure 4.5. To get homogeneous values, the data were normalized by doubling the value of each question before November 26, 2001 and halving the value of the Double Jeopardy Round. This is due to the fact that the values originally ranged from \$100 to \$500 but where doubled in 2001 [51]. Like the name suggests, the values in Double Jeopardy Round are doubled but do not reflect a higher difficulty. In the plot it can be observed that with growing monetary value the number of Triple Stumpers increases, which implies that these questions appear to be more difficult. The plot also shows

[4] http://j-archive.com/

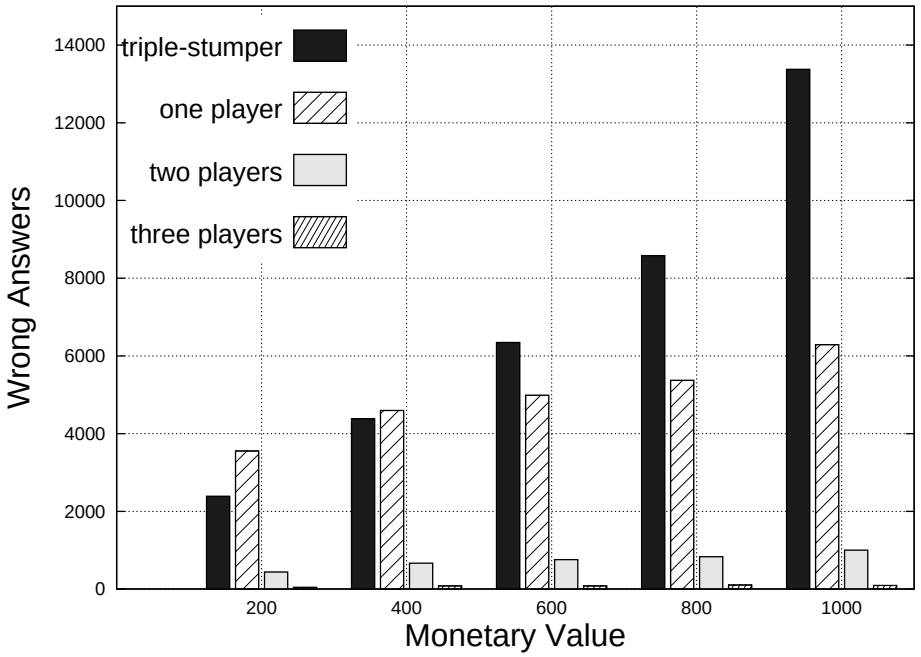

Figure 4.5: Correlation of the number of times a question could not be answered and the monetary value of the question

the number of times one, two, or three players gave the wrong answer. These numbers increase as well, but less drastically.

One component we identified as part of our question generation system, is a difficulty estimator that can decide whether a question belongs to the *easy* or *hard* category. From a high-level point of view our method works as follows: We identify a set of features for every question and integrate them into a mathematical model (the question difficulty classifier). The Jeopardy! questions are then used to train the model since they represent our gold standard of question difficulty. To map the monetary value of Jeopardy! clues to our easy/hard difficulty classes, we consider clues worth \$200 as easy and clues worth \$1000 as hard. Currently, we ignore questions of intermediate difficulties, since their difference in difficulty is subtly nuanced and therefore increases the hardness of the task significantly. The model is then used to assign a numerical value, that reflects the measure of difficulty, to unseen, automatically generated question. Using this difficulty measurement questions can then be ranked into one of the two difficulty categories.

4.3.1 Metrics

Before selecting the mathematical model of our classifier, we identified a set of metrics that can influence question difficulty positively (they make the question easier) or negatively (they make the question harder). These metrics rely on statistics derived from the knowledge graph and Wikipedia. The factors we identified are the following:

- *Popularity*: (global or category-based): Captures the intuition that questions about popular entities tend to be easier in general.
- *Selectivity*: Here the intuition is that more selective triples should give more useful clues to the user.
- *Coherence*: Represents the relatedness between pairs of entities. Based on intuition, more coherent pairs should provide better clues and thus make the question easier.

These metrics form the basis for our features, which we select in Section 4.3.2.3. In the following subsections we explain these metrics in greater detail and give their mathematical definitions.

4.3.1.1 Popularity

Popularity originated from the intuition that the more popular an entity is, the more people know about it and its properties. Thus, the question becomes easier. For example, when looking at the question's answer, popularity can be very decisive. A question about an entity that is unknown to the player makes it impossible for her to answer it. So it follows that the lesser known an entity is in general the harder it becomes (in general) to answer the question.

We define two flavors of popularity: *Global* popularity measures how well known an entity is for the general public, whereas *category-based* popularity measures how well known an entity is within a given category. For instance, *Barack Obama*, as the president of the United States, is internationally known, so in a list ranked by popularity he would rank highly. On the contrary, *Erna Solberg* (the prime minister of Norway) is much lesser known world wide, thus, she would occupy a lower rank than *Barack Obama*. But on a list for the category *Norwegian Politicians* she would rank fairly high. As discussed in Section 4.1.1.2, category popularity plays an important role when the entity has multiple types but is famous for only a subset of these types (e.g, *Barack Obama* is famous for being a politician, not as a musician).

Our approach of measuring popularity is based on the assumption that the more articles talk about a certain entity, the more important, or popular, this entity is. When an article mentions another entity, it is usually denoted by a link to the corresponding Wikipedia page. These links form a graph which can be exploited for measuring the importance of an entity within Wikipedia. With this in mind we can define the popularity of an entity as the number of articles that point to it (in-links). For obtaining a measure in the interval [0,1], we divide the number of in-links by the total amount of entities in Wikipedia, resulting in the following equation:

$$\phi(e) = \frac{I_e}{|E|} \tag{4.3}$$

Where e is an entity, I_e is the number of incoming Wikipedia links for e and $|E|$ is the total number of entities. On the other hand, category-based popularity is calculated by regarding only the links between entities that belong to the given category which form a sub-graph of the Wikipedia link-graph. Category-popularity is calculated as follows:

$$\phi^C(e) = \frac{I_e^C}{|E^C|} \tag{4.4}$$

Where I_e^C is the number of incoming Wikipedia links and $|E^C|$ is the number of all entities considering only entities that belong to category C.

4.3.1.2 Selectivity

Selectivity measures the number of triple patterns that exist for a relation-entity pair. The underlying idea is that more selective triple patterns have less possible answers associated with them and therefore give better clues to the player. Therefore, selectivity is measured as the reciprocal number of answer triples in the knowledge graph and can be expressed as a query for a triple pattern with one variable. For example, the number of results for query `?x <actedIn> <The_Green_Mile>` is much smaller than for the query `?x <livesIn> <Los_Angeles>`, thus the relation that *Somebody acted in The Green Mile* is more restrictive, hence more informative, than the relation *Somebody lives in Los Angeles*. To fit the measure into a [0,1] interval we divide 1 by the size of the answer set of the query. As an effect, the most selective triples with only one answer get a score of 1. The score decreases the less restrictive a triples is. Selectivity can be computed as follows:

$$\omega(rel) = \frac{1}{|q_{rel}|} \tag{4.5}$$

where rel is a relation of the kind `variable predicate object` or `subject predicate variable` and $|q_{rel}|$ is the number of results for query q of relation rel.

4.3.1.3 Coherence

In our setting, coherence captures the semantic relatedness of an entity pair. It can be best compared to measuring co-occurrences of entities in text. For example, it can be assumed that the entities `Barack_Obama` and `White_House` occur together more often than `Barack_Obama` and `Buckingham_Palace`. Therefore, the coherence of the former pair is greater than the coherence of the latter pair. In the context of question difficulty, we assume that higher coherence between entities in a question results in lower difficulty. This is especially true for the coherence of entities in the question and the target entity. When considering the Wikipedia link-graph, we measure coherence as the ratio between the size of the set of articles that point to both entities and the size of the union of the sets of articles that point to either one of the entities. This measure is known as Jaccard similarity coefficient, which is a statistic used for comparing the similarity of sets. Coherence can be derived as:

$$\varphi(e, f) = \frac{I_e \cap I_f}{I_e \cup I_f} \tag{4.6}$$

where e, f are entities and I is the number of incoming Wikipedia links for a given entity.

4.3.2 Question Difficulty Classifier

Even though difficulty is subjective, our goal is to use machine learning techniques to train a model that can classify questions according to their difficulty. Since there exists a large quantity of classifiers, there are multiple factors that need to be considered before choosing a classification model. These choices depend highly on the type of training data and the characteristics of the features. As training data we chose the Jeopardy! dataset, since it contains questions, their answers and an assigned difficulty. In this setting we use labeled data, so it follows that we have to select a classifier that can be used for a supervised learning task. Our features are based on the metrics that were discussed in Section 4.3.1. These features are continuous and exhibit linear behavior. For example, the higher the popularity of an entity, the easier the question should become. Thus, the decision boundary is linear. Another crucial criterion was to choose a classifier with a simple mathematical model, that we can understand and interpret, such that we can apply it to estimate the difficulty of unseen (automatically) generated questions. This was driven by the idea, that if the weights of the model can be learned from the training data, these weights could then be used to classify unseen questions and guide question generation. The final criterion is the format of the output data. Since our goal is to give binary classifications (easy/hard) for questions, the classifier of our choice should also account for this. We selected logistic regression as being the best fit for our task, since

it fulfills all the criteria mentioned above. Logistic regression is a supervised learning method, that can handle linearly separable features and classifies instances into discrete classes.

4.3.2.1 Selecting the Training Data

To learn the difficulty for a question we use training data that contain questions and their difficulty assessment. The assessments should be made by experts in the quiz field or a group of individuals of statistically relevant size. Additionally, the set's size should be adequate. A source of data that was compiled for research purposes is the *Question-Answer Dataset*, created by Smith et al. [52]. The dataset consists of questions-answer pairs with difficulty ratings, which were created as part of a student research project. We decided that the dataset is not a good fit for our purpose, since difficulty ratings where made by no more than two individuals and are therefore not representative of the general public. Another question-answer data source we considered was the *Quiz Bowl Incremental Classification Dataset*, as discussed in [53]. The dataset contains questions and answers of the *Quiz Bowl* game. Quiz Bowl is a quiz game where students of all education levels compete on questions about a wide variety of academic subjects. Even though the set contains the information if a question was answered correctly or not, it does not indicate its difficulty. Thus, it is not suitable for our learning task, as well. Lastly, we considered the archive of the game show Jeopardy!. *J! Archive* is a community effort of game show enthusiasts that index questions, answers and monetary values of almost every show, since it started in 1984. We crawled the information about the shows from the J! Archive web pages and parsed it into entries for each question. A sample instance of a question from the Jeopardy! dataset can be found in Figure 4.6.

```
question-id: 492
show-id: Show #11
date: Monday, 1984-09-24
round-type: Double Jeopardy Round
category: U.S. HISTORY
question: In '63, 200,000 Washington marchers heard him say, "I have a dream"
monetary-value: $200.0
daily-double: false
answer: Martin Luther King, Jr.
players-that-answered: Eric(right)
triple-stumper: false
```

Figure 4.6: A sample instance from the Jeopard! question dataset

Each instance has the following data fields:

- *question-id*: A unique identifier we generate for each question.
- *show-id*: An identifier for a particular Jeopardy! show.
- *date*: The date when the question was first aired.
- *round-type*: Type of the Jeopardy! round: The type can be *regular* or a *double* Jeopardy! round. In a double Jeopardy! round the monetary amount for questions are doubled, but the questions do not appear to be harder.
- *category*: A textual representation of the category.
- *question*: The question text.
- *monetary-value*: The dollar value assigned to a question. The player can earn this amount, when she answers the question correctly.
- *daily-double*: This indicates whether a question is a daily double. Any question can randomly be selected as daily double, which means that only the player who selected the question is allowed to answer it. Before answering, she has to decide how much of her already earned money she wants to wager.
- *answer*: A textual representation of the answer.
- *players-that-answered*: The names of the players that answered the question and an indicator whether their answer was correct or not.
- *triple-stumper*: If set to true, this field indicates that no player was able to give a correct answer. This is referred to as *triple stumper* in Jeopardy! terminology.

We use the extracted information for the training of our classifier. As input to our classifier we make use of the question text, the answer and the monetary value. The monetary value is normalized by doubling the value of each question before November 26, 2001, since the monetary values where doubled in later shows. In addition, the value of the Double Jeopardy Round is halved, since the monetary amount for these questions is doubled, but question difficulty is not affected. To gather questions that have the biggest gap in difficulty, we regard only questions with a dollar amount of $200 as easy questions and $1000 as hard questions.

The metrics discussed in Section 4.3.1 require information about the Wikipedia link graph, which is represented in the YAGO knowledge graph. Therefore, for further processing of the training data, it is required to identify the entities that are mentioned in a question.

For this task we utilized the AIDA [15] system (Section 2.2.4), which is a framework for finding and disambiguating entity mentions in natural language text[5]. We build an instance of the system and process each question as a single document, with *question-id* as the document identifier. The document's content is presented to the system in the form `<Question text>.[<Question answer>]`, which would result in the following input:

```
Shah Jahan built this complex in Agra, India to immortalize Mumtaz,
his favorite wife. [Taj Mahal]
```

For this question the AIDA system extracted the following entities in the question text: `<Shah_Jahan>,<Agra>,<India>,<Mumtaz_Mahal>`. The answer entity to the question is the entity: `<Taj_Mahal>`. In further processing steps, we filter out questions where the answer is not an entity and at least one entity is present in question text. This is required since the questions generated by our system exhibit similar characteristics.

4.3.2.2 Selecting Questions According to Coverage in YAGO

At this point, the training of our classifier on the data set did not yield satisfying results. This was due to multiple reasons: If the entities that were spotted in the question provide insufficient coverage of the actual entities in the question, the actual content of the question is not reflected correctly. Therefore, the classification task becomes distorted. For example, consider the question:

> *During WWII the Declaration of Independence was moved to this Kentucky military base for safety.*

In this question, the system found one entity only (`<Kentucky>`), while two more entities where missing (`<World_War_II>`, `<United_States_Declaration_of_Independence>`). Having only this information available, it is not possible to adequately reconstruct the question in the knowledge graph and our metrics become inapplicable. Another problem we encountered occurs when one or more entities are disambiguated incorrectly. This leads to an unintentional falsification of the classification results. For instance, if a question has a highly popular entity as an answer, it would be an indication of an easy question. However, if the answer is disambiguated with an incorrect, low-popularity entity, the question spuriously appears hard.

[5] For a detailed description of the AIDA system please refer to Section 2.2.4.

A related case we encountered was the following question:

> *This British physician and novelist based his Holmes character on one of his university professors.*

In this question the entities `<United_States>` and `<Sherlock_Holmes>` where spotted. It can be noticed that the token *British* was falsely identified as `<United_States>`, when the correct entity would have been `<United_Kingdom>`. Since we have no influence on the spotting and disambiguation task, we have to manually discard these questions from our training data set.

Since this task has to be carried out manually, we decided to limit our training set to *500* questions. We manually verified that in each question all spotted entities are disambiguated correctly and the coverage of entities is sufficient. Additionally, the question's semantics can be conveyed when the question is reconstructed in our knowledge graph. Before indiscriminately evaluating questions from the (large) J! Archive question set, we decided to apply another filtering step. Before considering questions for manual evaluation, we filtered out questions where the spotted entities are not connected in the YAGO knowledge graph. This accounts for relations between entities other than the `<linksTo>` relation, which indicates a Wikipedia Link and does not provide information about the relation's semantics. With this step we ensure that an underlying connection between the spotted entities is prevailing.

After the filtering step we generate a list of questions at random and start the manual annotation. When annotating Jeopardy! questions as being adequately replicable in YAGO, we evaluate the following criteria: Given the spotted, correctly disambiguated entities, would it be possible to reproduce the question in YAGO to an extent that the answer to question would still be unique? To exemplify this consider the question

> *Golf course groundskeeper Bill Murray battles golfers in this laugh riot.*

where the answer is the movie *Caddyshack*. The fact that Bill Murray acted in Caddyshack may be captured in the knowledge graph, but the resulting question *What movie did Bill Murray act in?* is very unlikely to be unique. As a counter-example, consider the following question:

> *Tommy Lee Jones & Anne Heche no doubt had a blast making this film that erupted on the screen in 1997.*

where the answer is the entity `<Volcano_(1997_film)>`. Here, the combination that Tommy Lee Jones and Anne Heche acted in a movie in 1997 is much more likely to be

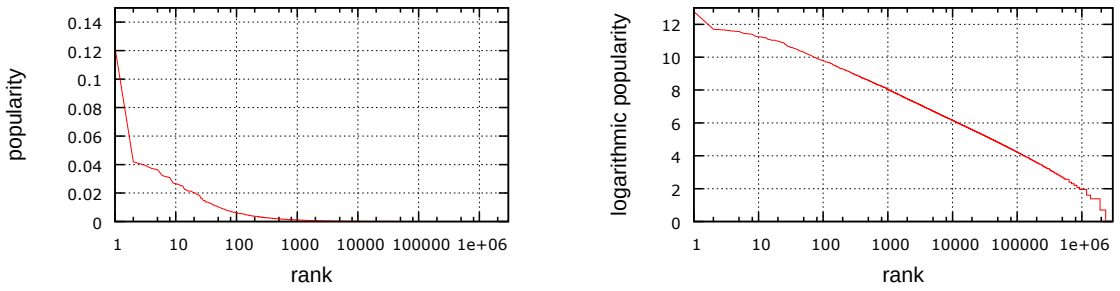

Figure 4.7: Popularity distribution among entities, logarithmic scale

unique. The resulting question *What 1997 movie did Tommy Lee Jones and Anne Heche act in?* is therefore likely to have just one answer and it is covered in YAGO at the same time. Therefore, this question would be evaluated positively and added to the training data set.

4.3.2.3 Features

The features that were selected for the classifying task are based on the metrics presented in Section 4.3.1, different attributes of the target entity and the type and number of the entities in the question. We selected features that measure the extrema (*minimum* / *maximum*), the arithmetic mean and the sum of the metrics. A list of features we identified and their description can be found in Table 4.5.

Furthermore, we found that the distribution of the values for popularity follows a *long tail* distribution. Meaning that there are few, highly popular entities (the head), compared to many entities with low popularity (the tail). The distribution of popularity is depicted in the left plot of Figure 4.7. The x-axis enumerates entities, sorted by popularity in logarithmic scale. The y-axis depicts the popularity measure. Because the margin between the most popular entities and entities at lower ranks is extremely high, we decided to use logarithmic damping on the original popularity measure. This has the effect, that the difference in popularity between highly ranked and lower ranked entities decreases, while achieving ranking equivalence. The resulting distribution can be seen in the right plot of Figure 4.7.

4.3.3 Incorporating Difficulty Estimating in Question Generation

In the context of generating a question with a specified difficulty, we propose a more principled approach than randomly selecting facts until we find a question with matching difficulty. We reason that we can incorporate difficulty estimates in our generation algorithm, to guide our search towards an easy or hard question. From a high-level point

Feature	Description
popularity of target (ϕ_{target})	popularity of the target entity
minimum popularity (ϕ_{min})	minimum popularity of all entities
maximum popularity (ϕ_{max})	maximum popularity of all entities
sum popularity (ϕ_{Σ})	sum over popularity of all entities
mean popularity (ϕ_{μ})	mean popularity of question and answer entities
mean popularity question (ϕ^{q}_{μ})	mean popularity of the entities in the question
person min. popularity (ϕ^{P}_{min})	minimum popularity of all entities of type person
person max. popularity (ϕ^{P}_{max})	max popularity of all entities of type person
person sum popularity (ϕ^{P}_{Σ})	sum over popularity of all entities of type person
person mean popularity (ϕ^{P}_{μ})	mean popularity of all entities of type person
organization min. pop. (ϕ^{O}_{min})	minimum popularity of all entities of type organization
organization max. pop. (ϕ^{O}_{max})	max popularity of all entities of type organization
organization sum pop. (ϕ^{O}_{Σ})	sum over popularity of all entities of type organization
organization mean pop. (ϕ^{O}_{μ})	mean popularity of all entities of type organization
location min. pop. (ϕ^{L}_{min})	minimum popularity of all entities of type location
location max. pop. (ϕ^{L}_{max})	max popularity of all entities of type location
location sum pop. (ϕ^{L}_{Σ})	sum over popularity of all entities of type location
location mean pop. (ϕ^{L}_{μ})	mean popularity of all entities of type location
other min. popularity (ϕ^{Oth}_{min})	minimum popularity of all entities of neither type
other max. popularity (ϕ^{Oth}_{max})	max popularity of all entities of neither type
other sum popularity (ϕ^{Oth}_{Σ})	sum over popularity of all entities of neither type
other mean popularity (ϕ^{Oth}_{μ})	mean popularity of all entities of neither type
maximum coherence (φ_{min})	maximum coherence of all entity pairs
sum coherence (φ_{Σ})	sum over coherence of all entity pairs
mean coherence (φ_{μ})	average coherence of all entity pairs
mean coherence (φ^{QTA}_{μ})	average coherence of entity pairs that involve answer
is person	binary indicator: answer is of type person
is organization	binary indicator: answer is of type organization
is location	binary indicator: answer is of type location
is other	binary indicator: answer is of neither type
number of entities	the number of entities in the question and answer

Table 4.5: Features and their description

of view, the approach still iteratively builds the question graphs by selecting facts, but with the following difference: after a new fact is selected, the upper and lower bounds for difficulty are calculated to decide whether it is still possible to reach the desired difficulty, given the already selected facts.

4.3.3.1 Intuition of the Approach

As described in Section 4.3.2, the model of our classifier for question difficulty is logistic regression. This model's parameters is a linear combination of weights, that minimize the classification error on the training data (Section 2.3.3). These weights and their

observations, in combination with the logistic function, output a class probability in the range $[0, 1]$[6]. When creating a question, it is now possible to estimate the bounds of the probability range, depending on the facts that are already selected in the question. As new facts are added, these bounds get tighter and therefore we can determine whether the current set of facts can be extended to generate a question with the desired difficulty. When we take a detailed look at the properties of the features, it can be noted that if the target entity and the number of entities in the question graph are fixed, the values for each feature increase or decrease monotonically when adding new entities. Features that account for the minimum can only decrease when adding new facts. Similarly, the maximum, average and sum can only increase when adding new facts (given that total the number of entities is fixed). From this follows, that the upper and lower bounds will gradually become closer together, enabling us to approximate the minimum and maximum reachable difficulty at any point in the question generation process.

Moreover, we distinguish between two kinds of feature states: *converged* and *converging*. If a feature is converged, its value is determined and is not going to change when new facts are added to the question graph. The value of converging features is still alterable, but only in an monotonically increasing or decreasing matter. While adding new facts, the set of converged features grows, since more features will change their state from converging to converged. For example, consider a question graph that is in the generation process. The maximum popularity, as defined in Table 4.5, may be $\phi_{max}(e) = \epsilon$. At this point, the feature is still considered as converging, since adding a fact that contains an entity with $\phi(e) > \epsilon$, would increase the feature's value. However, if a fact is added with an entity of popularity $\phi(e) = 1$, the maximum function reaches its highest possible value. Therefore, the feature's state changes from converging to converged.

A further distinction we make for converging features is whether they are *dependent* or *independent*. Independent features are only conditioned on a single factor, thus giving us more freedom in the manipulation of their values. An example of an independent feature is ϕ_{target}. Its value reflects the target entity's popularity and can therefore be directly influenced by choosing an entity with the desired popularity. In contrast, the value of dependent features is conditioned on various other factors. Thus, manipulating the value of a dependent feature is more difficult. For instance, the mean popularity (ϕ_{μ}) is dependent on the target entity's popularity, the number of entities in the question and the popularities of the entities in the question.

[6] Since we consider a binary classification problem, $P(easy)$ is equal to $1 - P(hard)$ and therefore it is sufficient to look at one class probability only.

4.3.3.2 Obtaining Bounds on Question Difficulty

In Section 2.3.3 we discussed the logistic function (Equation 2.4), which is the basis for the logistic regression classifier. To find the lower and upper bounds, given a question q, we need to minimize and maximize the linear function of the feature weights t. Let $D(q)$ be the difficulty function for question q then we can obtain the minimum and maximum bounds as follows:

$$\min D(q) = \frac{1}{1 + e^{-t_{min}}} \tag{4.7}$$

$$\max D(q) = \frac{1}{1 + e^{-t_{max}}} \tag{4.8}$$

Let f_i, $i \in \{1, ..., n\}$ be the features in . Let $\beta_i \in [-\infty, \infty]$ be the feature weights and let $x_i \in [0, 1]$ be the normalized observations. Now, each feature can be represented as a tuple $f_i = (\beta_i, x_i)$. To determine the bounds of $D(q)$, each feature has a lower bound and upper bound associated with it. These bounds limit the interval of the observations as $x_i \in [l_i, u_i]$. Let $\mathcal{F} = \{f_1, ..., f_n\}$ be the set of all features. Now, converging features in $\mathcal{F}$ have the property that the endpoints of the observation's interval differ ($l_i < u_i$). From this follows: If a feature is converged, their observations' difference of the endpoints is 0 ($l_i = u_i$). Therefore, the bounds can be obtained by summing over the upper and lower bounds of all features in $\mathcal{F}$. This is reflected in the following equation:

$$t_{min} = \sum_{f_i \in \mathcal{F}} \beta_i \, lower(x_i) \tag{4.9}$$

$$t_{max} = \sum_{f_i \in \mathcal{F}} \beta_i \, upper(x_i) \tag{4.10}$$

To find the bounds for each feature we have to distinguish four cases: The feature is (1) positive and increasing ($\beta_i > 0 \wedge x_i \to 1$), (2) positive and decreasing ($\beta_i > 0 \wedge x_i \to 0$), (3) negative and increasing ($\beta_i < 0 \wedge x_i \to 0$) or (4) negative and decreasing ($\beta_i < 0 \wedge x_i \to 1$). Depending on the case the minimal and maximal bounds can differ, since β_i are fixed and the bounds depend on x_i only. Table 4.6 lists the upper ($upper(x_i)$) and lower ($lower(x_i)$) bounds. t_{min} and t_{min} enable us to determine the minimum and maximum for $D(q)$, before adding new entities to the question. Since each feature's value is monotonically increasing or decreasing, it follows that the bounds for $D(q)$ shrink as we add new entities. Determining these bounds is essential when identifying the possibility to reach the desired difficulty, with the already selected entities. Based upon this information the algorithm can decide whether to continue with the current selection of facts or backtrack.

	$\beta_i > 0 \wedge x_i \to 1$	$\beta_i > 0 \wedge x_i \to 0$	$\beta_i < 0 \wedge x_i \to 0$	$\beta_i < 0 \wedge x_i \to 1$
$lower(x_i)$	x_i	0	x_i	1
$upper(x_i)$	1	x_i	0	x_i

Table 4.6: The four different kinds of features

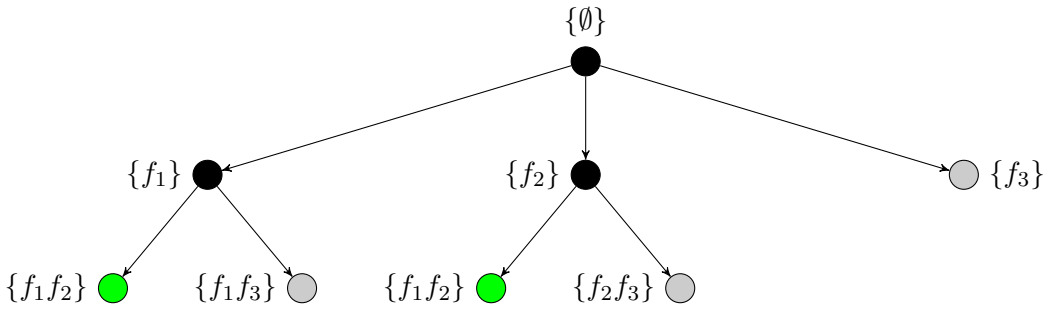

Figure 4.8: Example search tree for an entity with a total of three facts

4.3.3.3 Backtracking Algorithm Design

Using the scheme described in the previous section, we can cast the problem of finding a new question for a given difficulty into a backtracking problem. First, we give a formal definition of the search space corresponding to our problem: Consider the set of all facts for an entity as $\mathcal{F}_e = \{f_1, f_2, ..., f_n\}$. Then our problem space can be seen as a n-ary search tree, where each node has the currently selected facts associated to it. The algorithm starts at the root, adds facts to its set of selected entities $\mathcal{S}$ until it finds a combination of facts, that fulfill the input criteria. The algorithm is guided by the estimated bounds. In the case that the bounds indicate that no addition of any entity can achieve the desired difficulty, the algorithm backtracks. In case the entire search space is explored, the algorithm terminates unsuccessfully, since no satisfying solution is possible for the given target entity. Figure 4.8 depicts the search tree for an entity with a total of three facts. The set $\mathcal{S}$ is shown next to each node. States depicted with a black node indicate that the current state does not fulfill the exit criteria and therefore the search is continued. Gray nodes indicate that the bounds signal that no solution can be found for the current selection of facts and the algorithm has to backtrack. The procedure terminates as soon as a question with two facts and the target difficulty is found. Successful termination is denoted with a green node.

To cast question generation into a backtracking problem, we identify the following elements: P is the search tree, similar to the example seen in Figure 4.8. g is the current state of the question graph, depicted as nodes in the figure. f is a fact of entity e that can be added to g. The backtracking algorithm traverses the search tree in depth-first order,

```
procedure backtrack(g)
   if reject(P,g) then return
   if accept(P,g) then output(P,g)
   f = first(P,g)
   g += f
   while f != null do
       backtrack(g)
       f = next(P,g)
```

Figure 4.9: Backtracking algorithm in pseudo code

from the root to the leaf nodes. For each node g, which represents a partial question graph, the bounds of the difficulty are checked and the algorithm decides whether g can be completed to reach the desired difficulty level. In case it is not possible, the entire sub-tree is ignored, or pruned, in further processing steps. Otherwise, the algorithm decides whether g itself satisfies the input constraints and outputs it, in case it is a valid solution. If g is not valid, the algorithm recursively enumerates all possible sub-trees of g and continues. Therefore, the resulting search tree is only a sub-set of the entire search space, since sub-trees that can not result in a solution are pruned.

Backtracking can be implemented by defining six procedures that formalize the problem to be solved. In the context of the question generation task, these functions are defined as follows:

- $root(P)$: Return the target entity at the root of the search tree.
- $reject(P, g)$: Return true, only if the desired question difficulty is unreachable from the current state.
- $accept(P, g)$: Return true, if g has the specified number of facts and the desired difficulty.
- $first(P, g)$: Retrieve the first fact for partial solution g.
- $next(P, g)$: Retrieve the next alternative fact for partial solution g.
- $output(P, g)$: Output g as the solution for the question generation task.

After defining these functions, solving the problem can be reduced to one single call $backtrack(root(P))$. The following procedure shown in Figure 4.9 recursively implements the backtracking algorithm in pseudo code.

Chapter 5

Prototype Implementation

This chapter presents the implementation of the components described in Chapter 4. The first section elaborates on a high-level view of the system's architecture. The subsequent sections are structured top-down, following the system's component layers. The first layer is described in Section 5.2, where we present two web applications. One application was part a poster shown at the *2015 World Wide Web Conference* and demonstrates the question generation algorithm and the verbalization component. The second web application was used for the experimental evaluation of the difficulty classifier. Details about the implementation of the classifier can be found in Section 5.3, along with the details about the implementation of our question generation approach. The last section discusses details about the tripartite data storage layer, which contains the representation of the knowledge graph in a triple store.

5.1 System Architecture

Figure 5.1 depicts the system's architecture from a high level point of view. We decided to name the system "Q2G", which is an abbreviation for *Quiz Question Generator*. Following the architecture from top to bottom, users can make use of a web browser to access the web application layer (Section 5.2) which is made up of two components. The Q2G Web component is a web interface built for demonstration of the question generation and verbalization schemes. We created the second web application as part of an online experiment, which enables users to participate in a study we conducted as part of the experimental evaluation. The major part of the system is the core component, which is presented in Section 5.3. It contains the back-end, which implements the concepts presented in this thesis (Chapter 4). Also, it provides a web service which acts as an interface for the web applications to access data related to the question generation process.

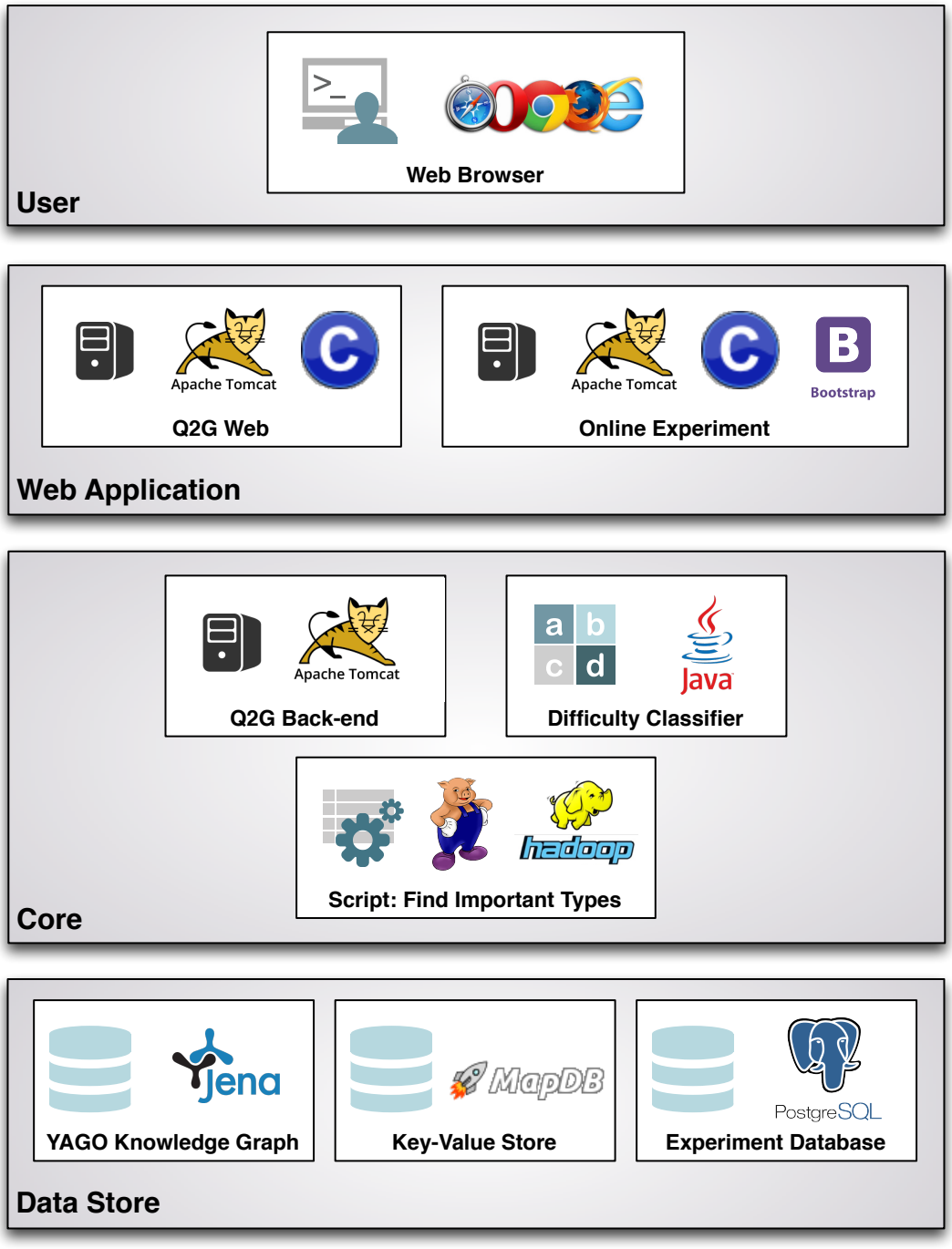

Figure 5.1: System architecture

Furthermore, it contains a custom implementation of a logistic regression classifier, which is applied to estimate question difficulty. The back-end facilitates communication with three databases, which make up the data store layer of the system (Section 5.4). This layer comprises the YAGO knowledge graph, which is represented in a triple store. Additionally, it contains a high-performance key-value store that stores pre-computed values and the SQL database, that was facilitated for our experiments.

5.2 Web Application

In the course of this thesis we built two web applications. The first application was part of the poster, which we presented at the 2015 World Wide Web conference. With the interface, a user can generate question graphs for a given YAGO entity and retrieve statistics about the graph and its facts. Also, it was possible to retrieve the verbalization for the generated question. The second web application was build for the purpose of evaluating the difficulty classifier as part of a user study. It provides a complete user interface that handles user management, input logging, and presentation of the data.

5.2.1 Q2G Web

Q2G Web was created as a web interface, which is responsible for the presentation of the generated questions to the user, and handling her input. It was implemented using the Apache Click[1] framework, which is built on the Java Servlet API. Using the framework enabled us to easily deploy the web application on an Apache Tomcat[2] web server. The communication between front-end, back-end, and database is implemented following the *Model-View-Controller* (MVC) design pattern, from Gamma et al. [54]. For data exchange, the front-end interfaces with the back-end, which receives data from the database and stores them as data access objects in the Java application. These objects are then serialized using a XML serializer and sent to the front-end, which deserializes the objects and handles the presentation to the user. The user input is also cast into objects, which are in turn transported to the back-end using XML. We decided to split presentation and back-end to keep the presentation application as light as possible. Our decision was based on the fact that the application had to run on a public web server, which hosts multiple applications simultaneously.

In the Q2G Web application, the user can input any YAGO entity and retrieve an automatically generated quiz question about that entity. A screenshot of the interface of

[1] https://click.apache.org/
[2] http://tomcat.apache.org/

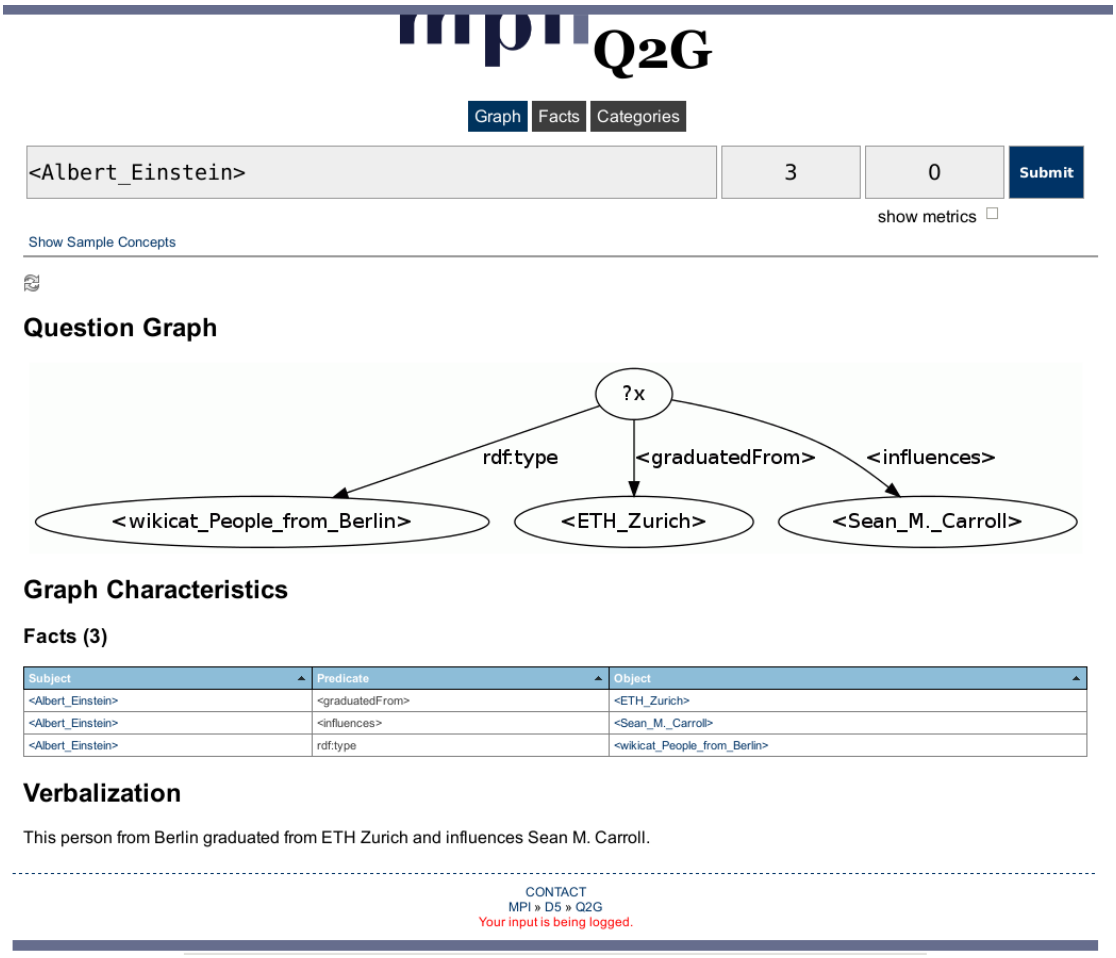

Figure 5.2: Q2G Web application

Subject	Predicate	Object	pSubject	pObject	Selectivity	Coherence
<Albert_Einstein>	<isCitizenOf>	<Switzerland>	0.0006344	0.0111637	0.0051282	0.0024080
<Albert_Einstein>	rdf:type	<wikicat_Jewish_American_scientists>	0.0006344	0.0000000	0.0027397	0.0000000
<Mileva_Marić>	<isMarriedTo>	<Albert_Einstein>	0.0000066	0.0006344	1.0000000	0.0070270

Figure 5.3: Metrics of a question about Albert Einstein

the system is shown in Figure 5.2. There, it is possible to specify the number of facts the question should contain. Additionally, the user can specify the number of variables in the question graph. This enables system to retrieve questions that are more than star queries. For example, the question could ask about the football player, whose spouse is a Spice Girl. Below the interface elements for the user's input, the question is shown in graph representation. This graph is automatically drawn using the SPARQL query of the question, that was generated by the system. Below the graph, one can see all facts from the YAGO knowledge base that are contained in the question. Using the checkbox "show metrics" it is possible to retrieve statistical information about these facts. In Figure 5.3 one can inspect the metrics for the facts of a question about Albert Einstein (*popularity*

subject, popularity object, selectivity of relation, coherence of entity pair). Please refer to Section 4.3.1 for a detailed explanation of the metrics. On the bottom of the web page, the question query is shown in its verbalized form.

5.2.2 Question Difficulty Evaluation Experiment

As part of our evaluation of the difficulty classifier, we conducted a user study with human participants (Section 6.1.3). In this context, we created a web application with user interface, to enable participants to access our study. Similarly to Q2G Web, the online experiment application uses Apache Click to implement the application logic and is therefore hosted on an Apache Tomcat web server, as well. However, we used the Bootstrap[3] front-end framework as building blocks for the user interface, which contains HTML- and CSS-based design templates and interface objects (e.g., buttons, boxes, etc). For this application it was not required to split data access/preparation and presentation, since no load-heavy operations needed to be executed. Internally, we also follow the MVC design pattern. The questions are retrieved in graph and SPARQL representation from a PostgreSQL[4] database (Section 5.4) and then cast into Java objects. The logic exclusively operates on these Java objects and persists the data when necessary. Error handling is performed by the view component. The logic of the application infers the rankings from a drag-and-drop list and stores the corresponding entries in an relational database. The database schema in UML notation can be found in Figure 5.4. Each question is represented as a question graph in the database. A question graph has an unique id and stores the question in SPARQL and DOT-graph representation, alongside with the name of the answer entity. Every human evaluator has a user name, which is linked to a unique identifier. Furthermore, the ID of the last entity, whose questions she worked on, is stored so that if she logs back in to the system, she continue where she stopped before. Each evaluated question has its rank and absolute difficulty level associated with it, as well as a time stamp. The difficulty assessments made by our classifier are kept in a separate table (`Question_Graph_Classification`). In addition to the human judgments, the classifier's assessments have the probability of being an easy question associated.

In the user interface, the participant is presented with a welcome page, where she can sign in using her user name. Each user's progress is stored in the database alongside her user name, so that she can stop at any point and return later to wherever she left off. After signing in, she is guided to the first of two main screens for the evaluation (Figure 5.5). There, she is presented with a ranked list of questions in either SPARQL or graph

[3] http://getbootstrap.com/
[4] http://www.postgresql.org/

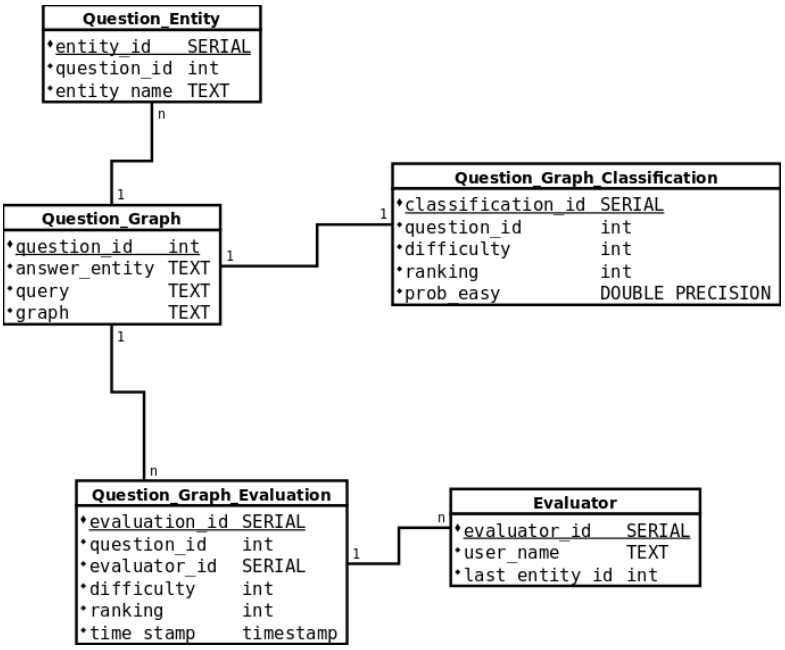

Figure 5.4: Database URL diagram for online experiment

view[5]. Using drag-and-drop she can move each question in the ranked list to the desired position. After using the "SUBMIT" button, she is guided to the next screen, Figure 5.6, where she is asked to give absolute difficulty judgments for the questions she ranked in the previous screen. The questions appear in the same order as she organized them before. After selecting the "SUBMIT" button, the system assures that all questions are rated and that the absolute difficulty assessments are aligned with the relative difficulty judgments. Then, she is again presented with the ranking screen, but with questions about the next entity. This procedure continues until she finished all questions and the system shows the closing "thank you" screen.

5.3 Core

As stated above, the core component implements the concepts that we present in this paper. More concretely, it implements the question graph generation algorithm (Section 4.1), the verbalization scheme (Section 4.2) and the difficulty estimation component (Section 4.3). Since the back-end executes the "expensive" computations, it provides a web service for the front-end and handles access to the data stores.

In the query generation process, the facts are retrieved from the YAGO knowledge graph which is stored using the Apache Jena[6] framework (Section 5.4). Once the minimum

[5] Using a button she can switch between the views.
[6] https://jena.apache.org/

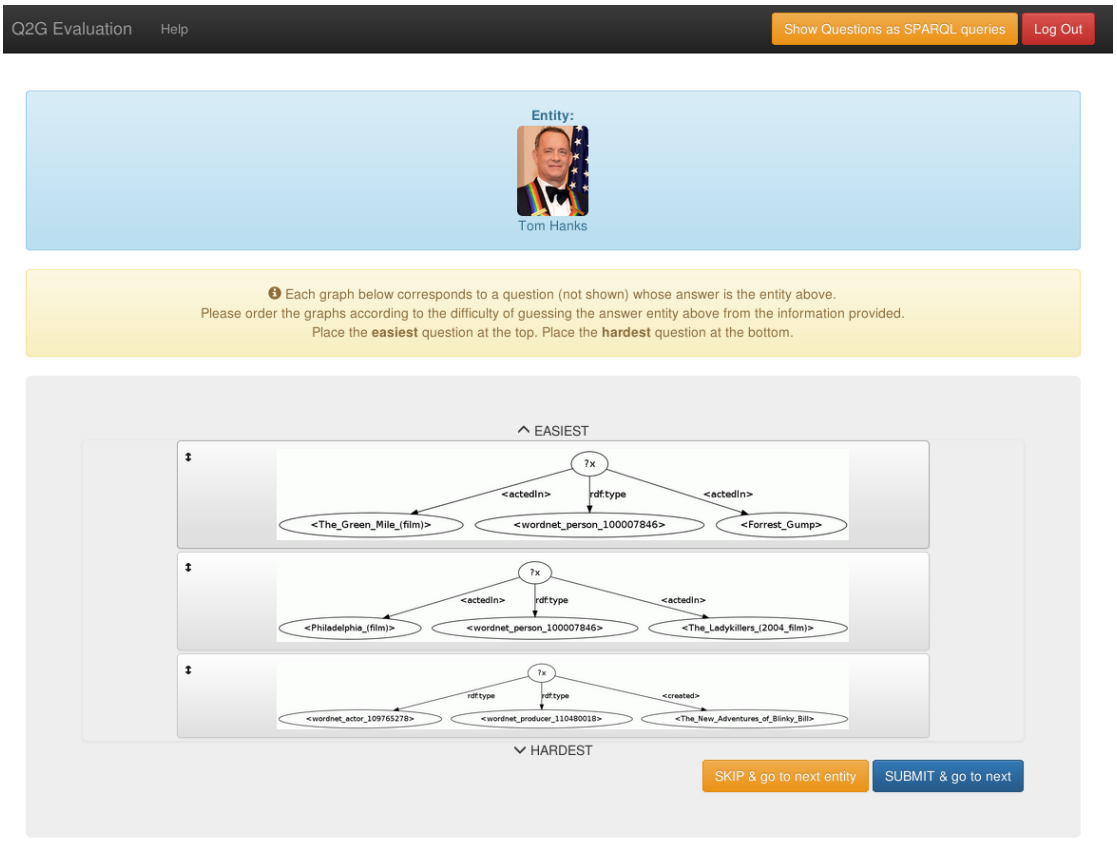

Figure 5.5: Difficulty ranking of a question about Tom Hanks for the user study

number of facts is selected, these facts are turned into a SPARQL query and then queried to the knowledge graph using the SPARQL interface of the Jena API. Depending on the size of the answer set, it is then decided if the answer to the question is unique or not. The query is visualized using Graphviz[7] and shown to the user via the Q2G Web interface. For being able to use Graphviz, we had to write a parser that turns SPARQL queries into the DOT language[8]. The metrics (popularity, selectivity, coherence) are either calculated on-the-fly or looked up from a disk-backed, key-value store. Since the possible number of combinations of relation-entity pairs for selectivity, and entity-entity pairs for coherence is too large to precompute, we decided to calculate these metrics at query execution time. To calculate selectivity, we replace the target entity of a fact with a variable and retrieve all facts that match the pattern from the knowledge graph. We then measure selectivity as the reciprocal number of the retrieved answer triples. For coherence, we query the knowledge graph to retrieve the Wikipedia links and calculate the Jaccard coefficient of the sets of Wikipedia articles pointing to both entities. However, popularity for each entity is precomputed as the number of Wikipedia articles that point

[7] http://graphviz.org/
[8] http://www.graphviz.org/content/dot-language

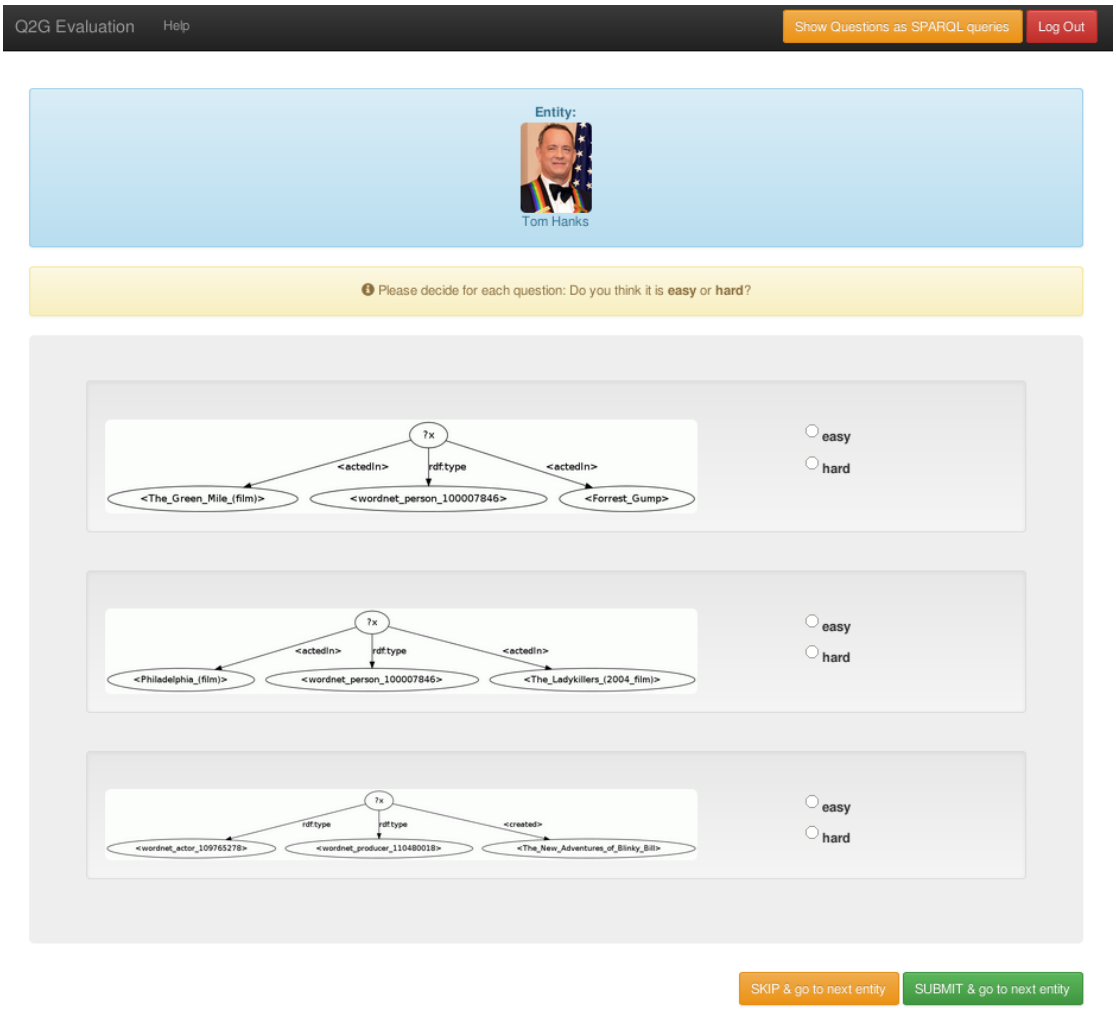

Figure 5.6: Absolute difficulty assessment of a question about Tom Hanks for the user study

to the entity's page. These values are stored as *entity-popularity* pairs in a key-value store, as well.

For the verbalization component, we use an incremental implementation that verbalizes triples first and then assembles the question according to our pattern-based scheme, presented in Section 4.2.1. For the paraphrasing of predicates, we maintain a file that contains a mapping of each relation to its paraphrase, depending on whether the target entity is the subject or object in the question. The algorithm chooses randomly between the standard paraphrasing and the additional paraphrases, that were introduced in Section 4.2.2. A Pig script was used to implement our approach of finding salient types for an entity, which is discussed in Section 4.2.3. Because of the size of the textual types dataset, the script was run "offline" on a Hadoop cluster. The resulting salient types were put in a key-value store, so that they only need to be looked-up when the query is verbalized.

The difficulty classifier was implemented in two iterations. In the first iteration we used the WEKA machine learning framework to learn the features for our logistic regression classifier. Section 2.3.6 describes details about WEKA and Section 2.3.3 elaborates on the details of logistic regression. We used the WEKA tool, since it gave us enough freedom to inspect the data and experiment with various classifiers and features. Once we selected the classifier type and the corresponding features, we used WEKA to train our regression model and learn the optimal feature weights. Since WEKA is tailored to work on a single data set, we found that it performs poorly on single instance classification. Therefore, we decided to implement the logistic regression classifier in Java, from scratch. The classifier receives the feature weights, learned using WEKA, and a question graph as input and classifies the question accordingly. Using our customized Java classifier, we were able to reduce the processing time for a single instance by a factor of 10.

5.4 Data Store

The system has three main data stores. The central component for data storage is the YAGO knowledge graph, which is implemented as a Jena TDB triple store. The second component is a simple key-value store, which is implemented using the disk-backed MapDB[9] library. The key-value store contains the popularity tuples for all entities in YAGO, as well as the salient types which where identified by our Pig script component. In addition, the Wikipedia Catgory graph (Section 4.1.1.1) is also stored in two maps. One captures the category to subcategory mapping, whereas the second map captures the entity-category membership. The third data storage component is the experiment database, which is a relational database that uses PostgreSQL. The experiment database contains the graphs and queries that are used in our experiment, as well as the question difficulty rankings and absolute difficulty judgments that where assessed as part of the user study. Furthermore, it contains the classification probabilities that were obtained from our difficulty classifier. Since the database contains both the outputs of the user study and the corresponding classifier output, the PostgreSQL database was also utilized for all calculations that were part of the evaluation. We decided to split the storage of data into three components, due to the unique internal data representation models of each database scheme. Since data represented as triples can be imagined as a graph, it is more natural to store them in a graph database. On the other hand, information needed for our experiment could be more clearly represented and organized in a relational database, since it allows for multiple, named tables with attributes. Furthermore, one can leverage the SQL language to analyze the data more conveniently, which was essential for

[9] http://www.mapdb.org/

computing the statistics of our experiments. The motivation to use a separate key-value store for tuples was solely based on performance.

From a data-centric point of view, the knowledge graph serves as the data back-end of our system. It contains all the semantic knowledge about entities and their relations from the YAGO knowledge base, and additionally allows to infer statistics about the Wikipedia link graph. YAGO represents these facts using RDF triples, which are preferably represented in a triple store. We decided to use the Java framework Apache Jena for the storage of the knowledge graph. We preferred Jena, since it features the TDB component that allows for the storage and query of RDF data and can be used as a high performance RDF back-end on a single computer. Furthermore, it features a rich Java API, which makes it conveniently accessible from within the Java code, since the API enables us to pose queries in SPARQL language. Another reason for choosing Jena was its ability to compactly store data, which resulted in the database using a total of 9.3 GB in disk space. The size of the database was equal to the raw data, thus the system minimizes the database overhead. Additionally, the framework for accessing the data store was hosted entirely on a workstation computer in a Java virtual machine with 4 GB main memory and Intel Core i3-2120 CPU, clocked at 3.30 GHz.

For populating the database we used the *turtle* files available for download at http://yago-knowledge.org/. Using the `tdbloader2` command line tool, which is part of the Jena suite, we imported the data into the Jena database. For the purpose of saving disk space, we use only portions of YAGO, which are relevant to our system:

- *yagoFacts*: Contains all facts that hold between entities, excluding literals.
- *yagoTypes*: Contains the coherent types extracted from Wikipedia.
- *yagoTransitiveType*: Contains the transitive closure of all rdf:type/rdfs:subClassOf facts.
- *yagoTaxonomy*: Contains all `rdfs:subClassOf` facts derived from Wikipedia and WordNet.
- *yagoWikipediaInfo*: Contains information about in/out-links between Wikipedia articles.

Since the type taxonomy only includes the immediate class-to-sub-class relations, we had to automatically generate the transitive closure of the taxonomy. This was required for the question generation step, where we look for the most specific type of an entity (Section 4.1.2.2). The resulting database contained 273,781,489 facts from YAGO.

Chapter 6

Experimental Evaluation

In this chapter we present various experiments to evaluate the effectiveness of our methods. In the first section, we experimentally evaluate our approach for predicting question difficulty. As part of the experiment, we inspect the agreement of humans labeling the difficulty of Jeopardy! questions (Section 6.1.1). Additionally, we compare their labels to the difficulty assessments, made by the Jeopardy! clue writers. In Section 6.1.2, we present an empirical study that investigates the performance of the difficulty classifier on the Jeopardy! dataset. Section 6.1.3 discusses our experimental setup and our results of a user study that measured the agreement of question evaluators, with our difficulty classifier. Finally, we briefly discuss a query processing method that was customized to our question generation algorithm.

6.1 Question Difficulty

In this section we present the setup and evaluate the results of two users studies that deal with question difficulty. In the first study we measure the inter-rater agreement of humans labeling Jeopardy! questions as hard or easy. We further compare their assessments to a gold standard, which is the difficulty rating made by the Jeopardy! writers. In the second user study, we evaluate how much our classifier agrees with human question difficulty evaluators on automatically generated questions. There, we measure the agreement for relative and absolute difficulty judgments.

	$eval_2$	$eval_3$	majority
$eval_1$	0.192	0.325	0.500
$eval_2$		0.443	0.661
$eval_3$			0.810

Table 6.1: Agreement between human evaluators (all measurements are Fleiss' Kappa)

6.1.1 Human Evaluation of Jeopardy! Question Difficulty

In our first user study, we inspected the agreement between humans when assigning binary difficulty labels to Jeopardy! questions. Furthermore, we compare their labels to those given by the Jeopardy! clue writers, which we consider as a gold standard. Our setting works as follows. We randomly choose 200 questions that were either labeled as easy ($200) or hard ($1000) by the Jeopardy! writers. More concretely, we ensure that the same amount of easy and hard questions is selected. In the subsequent step, three evaluators ($eval_1$, $eval_2$, $eval_3$) annotated the question difficulty for all questions. After obtaining their difficulty labels, we compared their overall agreement, pairwise agreement and agreement, in terms of Fleiss' Kappa [55], with the majority vote. Here, the majority vote is simply the label that was assigned by the majority of the evaluators. Overall, the evaluators reach an agreement of $\kappa = 0.328$, which is considered fair according to Landis and Koch [56]. We attribute the fair overall agreement of the evaluators to the subjectivity of the task. Table 6.1 shows the pairwise agreement, as well as the agreement of each evaluator with the majority vote. From the table it can be seen that pairwise agreement of the annotators ranges from fair to moderate. However, each annotator achieves moderate or substantial agreement with the majority vote, which leads us to regard the majority vote as an appropriate reflection of human performance. When considering the majority vote of the three evaluators, they were able to correctly classify 62.5% of all questions, compared to the Jeopardy! gold standard.

6.1.2 Validation of Question Classifier

In this subsection we evaluate the performance of our question difficulty classifier, introduced in Section 4.3.2, on the Jeopardy! training data. The training data consisted of a set of 500 Jeopardy! questions that have been annotated as being covered in YAGO (Section 4.3.2.2). Using ten-fold cross validation (Section 2.3.5), our classifier was able to correctly classify 66% of the questions. To achieve this result, we performed a feature ablation study, were we compared different combinations of features and selected the combination with the highest performance.

Row	popularity	log. popularity	coherence	types	**performance**
1	yes	yes	yes	yes	**66.4%**
2	no	yes	no	yes	65.8%
3	yes	yes	no	yes	65.8%
4	no	yes	yes	yes	65.6%
5	no	yes	yes	no	64.2%
6	no	yes	no	no	63.8%
7	yes	yes	yes	no	62.6%
8	yes	no	no	no	62.4%
9	yes	no	no	yes	62.2%
10	yes	yes	no	no	62.2%
11	yes	no	yes	no	61.8%
12	yes	no	yes	yes	61.8%
13	no	no	no	yes	60.0%
14	no	no	yes	yes	57.8%
15	no	no	yes	no	52.4%
16	no	no	no	no	50.0%

Table 6.2: Results of ablation study of the features introduced in Section 4.3.2.3. The performance is based on the cross-validation of the question difficulty classifier.

We grouped features into the following four categories:

- *Popularity*: Features that are related to entity popularity (in Table 4.5 these are denoted with ϕ).
- *Logarithmic popularity*: Features that are related to entity popularity (ϕ) on logarithmic scale.
- *Coherence*: Features related to entity coherence (φ).
- *Types*: Features related to types (all $\phi^T, T \in \{P, O, L, Oth\}$ and *is person*, *is organization*, *is location*, *is other*).

In Table 6.2 we give all feature combinations ordered by their corresponding performance, which reflects the percent of instances that were classified correctly. It can be observed, that the best performance is reached when all features are enabled and removing features leads to worse results. Thus, we reason that all features are relevant and contribute positively to the classification performance. Moreover, we noticed that popularity-related features play an essential role in the overall performance. We came to this conclusion, since the feature combinations that omit popularity are situated at the lowest ranks of Table 6.2. We further noted that choosing log-popularity over popularity results in a major performance improvement. In the table this can be seen in row 12 and 4. In row 12, coherence- and type-based features are enabled in combination with “normal”

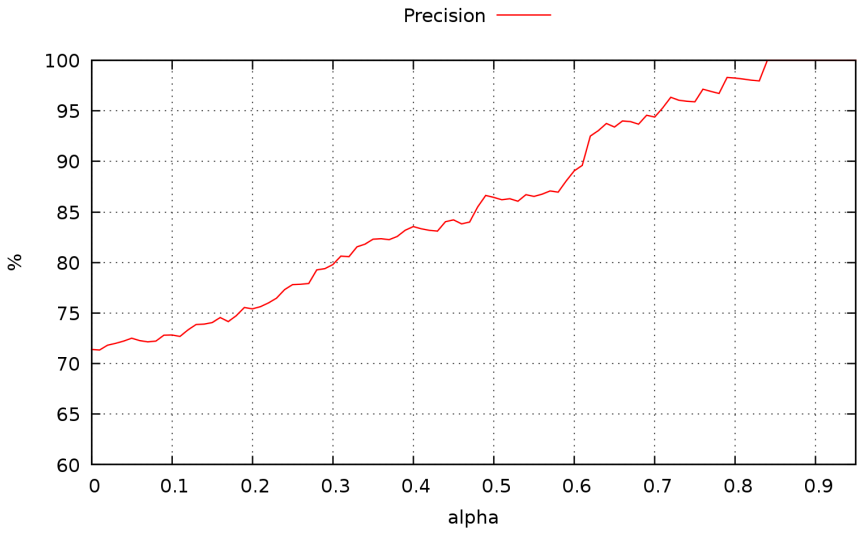

Figure 6.1: Classifier precision vs. size of remaining dataset

popularity. Here the 61.8% of questions are classified correctly. When exchanging popularity with logarithmic popularity (row 4), performance increases to 65.6%. When using a combination of both (line 1) performance increases to 66.4%.

Furthermore, we compared the percentages of correctly classified questions by our classifier (66%) with the percentage of correctly annotated questions with the majority vote of human evaluators (62.5%)[1] and find that the classifier slightly outperforms the human competition by 3.5%.

In addition to measuring the percentage of correctly classified instances, we inspected the precision of our classifier in correspondence to the confidence of the predictions. Since we use logistic regression, the output of the classifier is the probability of an instance being easy $[P(easy)]$ or hard $[P(hard)]$. Since $P(easy) = 1 - P(hard)$, we define confidence as $\alpha = |P(easy) - P(hard)|$. We then inspect the percentage of correctly classified instances by considering predictions above a certain α-value only. The plot in Figure 6.1 depicts the precision (percentage of correctly classified instances) on the y-axis, compared to the confidence (α-value) on the x-axis. As expected, the precision increases with the confidence of the prediction. In our setting, the system provides enough flexibility to generate multiple questions for a target entity and pick one with the highest confidence. It could also be imagined to pick the questions among the ones that pass a certain confidence threshold.

[1] Details can be found in Section 6.1.1.

6.1.3 User Study

To compare the difficulty assessments of our classifier to the ones made by humans, we conducted a user study, to experimentally prove the effectiveness of our method. The study was executed using a web-based interface, which was published to the Databases and Information Systems group of the Max Planck Institute. The reason we restricted the study to colleagues only is that we needed computer science experts, who are able to read SPARQL queries and/or RDF graphs. The following subsections, will elaborate on the setup of the experiments and present our results.

6.1.3.1 Experimental Setup

For the user study we manually selected a set of $n = 50$ entities that have reasonable coverage in YAGO. Coverage is considered reasonable when at least 5 non-type facts exist, to ensure that it is possible to generate questions with a certain variety. Participants could rate the same set of n entities and questions, but they were allowed to skip entities that they were not familiar with. For each entity, we presented three automatically generated questions to the participants and asked them to (1) order them according to relative difficulty (Figure 5.5) and (2) give absolute difficulty judgments (Figure 5.6). Questions were shown to participants in graph representation and as a SPARQL query to avoid any bias introduced by an automatic or human verbalization. The users were able to switch between the graph and query views. Since questions were shown in these representations, it was necessary to ask computer science experts for participation in our experiment.

To measure relative difficulty, we proceed as follows: Let E be the set of n selected entities $\{e_1, e_2, ..., e_n\} \in E$. For each e_n we generate questions q_{i1}, q_{i2}, q_{i3} for $i = \{1, .., n\}$. Now, we can obtain a relative difficulty ranking triple r as $q_{ia} < q_{ib} < q_{ic}$, for any permutation of $a, b, c \in \{1, 2, 3\}$. For each r we can now infer three binary ranking pairs ($q_{ia} < q_{ib}$, $q_{ib} < q_{ic}$, $q_{ia} < q_{ic}$). To measure the correlation between the rankings of participants and our classifier, we use Kendall's τ rank correlation coefficient, which is defined as:

$$\tau = \frac{n_c - n_d}{\frac{1}{2}n(n-1)} \tag{6.1}$$

where n_c is the number of concordant pairs, n_d is the number of discordant pairs and n is the total number of pair combinations. In the equation, the denominator accounts for all possible pair combinations, therefore the value of τ ranges between -1 and 1. $\tau = 1$ in the case both rankings perfectly agree. On the other hand, $\tau = -1$ if the disagreement

of the rankings is perfect, meaning that one ranking is the reverse of the other. If both rankings do not agree τ is expected to be close to zero.

Let R^C be the set of ranking pairs of our classifier and R_k^H the ranking pairs of participant $k, k = \{1, 2, ..., N\}$. A pair is concordant if both the classifier and the participant agree on the ranking of a question pair (e.g., $q_{ia} < q_{ib} \in R^C$ and $q_{ia} < q_{ib} \in R_k^H$). A pair is discordant if the classifier and participant have opposed difficulty rankings (e.g., $q_{ia} < q_{ib} \in R^C$ but $q_{ia} > q_{ib} \in R_k^H$). Now, we can obtain the average correlation using Kendall's τ as:

$$\frac{1}{N}\sum_{k=1}^{N}\tau(R^C, R_k^H) \tag{6.2}$$

When obtaining absolute difficulty judgments, we want to show that there is sufficient agreement between our classifier and the human ratings. Our basis are binary judgments for questions q_{ij}, where i enumerates all entities and j enumerates the questions for each entity. To obtain a numerical difficulty rating we define:

$$D(q_{ij}) \begin{cases} 0 & q_{ij} = easy \\ 1 & q_{ij} = hard \end{cases} \tag{6.3}$$

We use Cohen's kappa, which is a statistic that reflects the inter-rater agreement of two participants which rate a fixed number of items into mutual exclusive categories:

$$\kappa = \frac{p_0 - p_e}{1 - p_e} \tag{6.4}$$

Here, p_0 is the relative observed agreement between two raters and p_e is the expected chance of agreement. p_e can be seen as the hypothetical probability that two raters agree, given the observed data. The measure has the following characteristics: If two raters are in complete agreement, then $\kappa^{cohen} = 1$. The lesser the raters agree, the more κ^{cohen} tends towards 0. Since Cohen's kappa measures the agreement of two raters only, we calculate the agreement of every participant with our classifier and average over the result. For participant $k, k = \{1, 2, ..., N\}$, this is reflected in the following equation:

$$\frac{1}{N}\sum_{k=1}^{N}\kappa^{cohen}(R^C, R_k^H) \tag{6.5}$$

	τ	$\overline{\tau}$	κ	$\overline{\kappa}$	$\|Q\|$
$eval_1$	0.538	0.227	0.544	0.229	39
$eval_2$	0.538	0.454	0.333	0.281	78
$eval_3$	0.519	0.303	0.540	0.315	54
$eval_4$	0.480	0.638	0.348	0.462	123
$eval_5$	0.880	1.426	0.524	0.850	150
$eval_6$	0.625	0.973	0.340	0.529	144
$eval_7$	0.591	0.594	0.510	0.513	93
$eval_8$	0.741	0.865	0.500	0.584	108
$eval_9$	0.627	1.018	0.333	0.540	150
$eval_{10}$	0.167	0.151	0.241	0.219	84
$eval_{11}$	0.455	0.324	0.515	0.367	66
$eval_{12}$	0.619	0.562	0.351	0.318	84
$eval_{13}$	0.533	0.173	0.467	0.151	30
average	**0.563**	**0.593**	**0.427**	**0.412**	**92.538**

Table 6.3: Agreement between human evaluators ($eval_n$) and the difficulty classifier. τ values indicate the agreement in terms of relative difficulty and κ values indicate agreement on absolute difficulty.

6.1.3.2 Evaluation of Results

In this section we discuss our findings of the user study we conducted to evaluate the performance of our difficulty classifier. As stated above, we evaluate the classifier in terms of (i) relative and (ii) absolute difficulty judgments. We present the results of our user study in Table 6.3. A total of 13 evaluators took part in the study, as captured in the table. The columns labeled as τ and κ correspond to the measures defined in Equations 6.1 and 6.4, respectively. To recall, τ rages in $[-1, 1]$ and κ ranges in $[0, 1]$. Furthermore, the columns labeled as $\overline{\tau}$ and $\overline{\kappa}$ correspond to the weighted measures of τ and κ. There, each user's contribution to the final average depends on the number of questions she evaluated. Therefore, we give higher emphasis to the values of those users who finished the study and avoid overly representing users that evaluated only a small numbers of questions. In the non-weighted case, each user is treated "equally", which has the effect that users who label the questions of a single entity only have as much influence on the result as a user that finishes all 150 questions. The last column ($|Q|$) corresponds to the number of questions that each participant has evaluated.

Following the table it can be reasoned, that the rankings produced by classifier moderately agree with the human annotators with $\tau = 0.563$. The τ-values of most evaluators range between 0.625 and 0.519. $eval_5$ and $eval_{10}$ seem to be outliers in the data set, since there values correspond to 0.880 and 0.167, receptively. If we disregard the outliers, the τ values vary 0.286 from the minimum to the maximum τ. Additionally, when the τ-values for users are weighted by study participation, the weighted average rises to $\overline{\tau} = 0.593$.

The average agreement of absolute difficulty assessments, as captured by Cohen's Kappa, has a value of $\kappa = 0.427$ and is considered to be moderate according to Landis and Koch [56]. Here, the pairwise agreement between most annotators and the classifier ranges around the 0.5 mark. The difference between maximum and minimum κ values is 0.303, which is close to the value for τ. Following this, we reason that there is about the same variance in κ and τ. In general, we attribute the reason for the relatively high variance of both values to the subjectivity of the task. When weighing the κ-values, the average performance falls slightly to $\overline{\kappa} = 0.412$, which is still considered as moderate agreement.

6.2 Anecdotal Results

This section presents anecdotal results of the output of our system. In Table 6.4 we present example questions that were generated by our system for the category `Renaissance_artists` and difficulties easy and hard. The entity that corresponds to the given category is `<Leonardo_da_Vinci>`. For each question we present the SPARQL query and verbalization produced by our system. Further we show the class probability that was provided by the question difficulty classifier. To recall, if the probability is greater than 0.5, the question is considered as easy, whereas a probability smaller than 0.5 is an indication for a hard question. Therefore, the first question was classified as easy with probability 0.8704. The verbalization of the question shows how the salient types, which we identified in Section 4.2.3, can be used to express that *Mona Lisa* is a painting. The second question is classified as hard. In the verbalization one can see how the triple `<John_Argyropoulos> <influences> ?x`, where the target entity is an object, is verbalized as *is influenced by*. The verbalization of the same relation but with the target entity as subject can be inspected in the third question. Here, the relation of the triple `?x <influences> <Victor_Bregeda>` is verbalized as *influences Victor Bregeda*. Additional anecdotal results are presented in Appendix B for the interested reader.

The average time it takes to generate a question for an entity with three triples is close to 600 ms. If the number of triples is raised to 5, the average generation time drops to about 300 ms. The reason for this decrease is that questions with five triples tend to have unique answers more often than questions with three triples only. Therefore, less questions have to be discarded before a question with a unique answer is found.

Category	`Renaissance_artists`
Difficulty	*easy*
Target Entity	`<Leonardo_da_Vinci>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_painter_110391653> .`
	`?x <created> <Vitruvian_Man> .`
	`?x <created> <Mona_Lisa>`
	`}`

Verbalization	*This painter created Vitruvian Man and the painting Mona Lisa.*
$P(easy)$	0.8704
Category	`Renaissance_artists`
Difficulty	*hard*
Target Entity	`<Leonardo_da_Vinci>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_inventor_110214637> .`
	`?x rdf:type <wordnet_artist_109812338> .`
	`<John_Argyropoulos> <influences> ?x`
	`}`

Verbalization	*This inventor and artist is influenced by John Argyropoulos.*
$P(easy)$	0.4779
Category	`Renaissance_artists`
Difficulty	*hard*
Target Entity	`<Leonardo_da_Vinci>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_artist_109812338> .`
	`?x rdf:type <wordnet_scientist_110560637> .`
	`?x <influences> <Victor_Bregeda>`
	`}`

Verbalization	*This artist and scientist influences Victor Bregeda.*
$P(easy)$	0.2383

Table 6.4: Example questions for entity Leonardo da Vinci generated by the system

```
QUERY:
SELECT (COUNT(*) AS ?count) {
   SELECT ?x WHERE {
                ?x rdf:type <wordnet_person_100007846> .
                ?x <actedIn> <Toy_Story_3> .
                ?x <actedIn> <Forrest_Gump> .
   }
}

EXECUTION PLAN:
Reorder: (?x <type> <wordnet_person_100007846>)
         (?x <actedIn> <Toy_Story_3>)
         (?x <actedIn> <Forrest_Gump>)
>> Input
    0  1382348 : ?x <type> <wordnet_person_100007846>
    1       10 : ?x <actedIn> <Toy_Story_3>
    2       10 : ?x <actedIn> <Forrest_Gump>
<< Output
    ?x <actedIn> <Toy_Story_3>

>> Input
    0        1 : TERM <type> <wordnet_person_100007846>
    1          : null
    2        1 : TERM <actedIn> <Forrest_Gump>
<< Output
    TERM <type> <wordnet_person_100007846>

>> Input
    0          : null
    1          : null
    2        1 : TERM <actedIn> <Forrest_Gump>
<< Output
    TERM <actedIn> <Forrest_Gump>
```

Figure 6.2: Example of Jena's optimization of a query execution plan

6.3 Custom Query Execution

As part of an optimization effort, which was aimed at increasing the speed of query execution when querying for the uniqueness of a question, we inspected logs of query execution plans of the underlying Apache Jena database. The Jena framework comes with built-in optimizer for query execution plans, which uses a statistics file that contains counts for triples with one variable (`?x <type> <wordnet_person_100007846>`) and the answer set's size $(1,382,348)$. For a given query Jena reorders the statements, such that more selective statements are executed first. To illustrate this, consider the query execution plan that can be found in Figure 6.2, which depicts the optimization process for a query with three triples. At every stage, one fact is selected from the input-set (`>> Input`). The selected fact is marked by "`<< Output`". In our example the optimizer knows that the size of the result set for `<wordnet_person_100007846>` is extremely large. Therefore, one of the more selective statements is chosen first (`?x <actedIn> <Toy_Story 3>`). The rest of the statements is then evaluated given the already chosen statement. Thus, it does not make a difference in execution cost, whether `<type> <wordnet_person_100007846>` or `<actedIn> <Forrest_Gump>` is chosen next.

Step	Selected triples	Result size	Action
1	<actedIn><Toy_Story_3>	10	continue
2	<actedIn><Toy_Story_3> <actedIn><Forrest_Gump>	1	terminate
3	<actedIn><Toy_Story_3> <actedIn><Forrest_Gump> <type><person>	1	not considered

Table 6.5: Example for custom query execution

6.3.1 Intuition of the Approach

Our question generation scheme iteratively adds facts to a question graph. As part of the procedure, the knowledge graph is queried multiple times to check for uniqueness of the answer, given the selected triples, until a suitable combination of triples is found (Section 4.1.3). Instead of running a full query at each step with just one additional triple, our approach stores the result set of intermediate queries and joins it with the result set of the subsequent query. Furthermore, it exploits the fact that when an answer to a query is unique, using only a subset of the already selected triples, it terminates without considering additional triples. This is possible since adding more triples to an already unique query will not change the uniqueness. As a further improvement, the algorithm calculates the joins of triples that do not indicate types first, since these queries are mostly less expensive. This is due to the fact that their result sets are smaller on average. For demonstrative purposes, consider the example that can be found in Table 6.5. Here, the algorithm can terminate after two steps, since the result set of `<actedIn> <Toy_Story_3>` joined with the result set of `<actedIn> <Forrest_Gump>` is already one, thus the answer is unique. The third step does not need to be considered, which spares us the expensive query `?x <type> <person>`.

6.3.2 Evaluation of the Results

The results for our empirical evaluation can be inspected in Table 6.6. We ran four tests with different inputs and measured the average time to generate a question. The tests where executed for 100 questions were each question was generated for a different entity picked at random (Test number 1 and 3). In contrast, Tests 2 and 4 were executed on 100 questions about a single entity only. We also varied the number of minimum triples in the question, which we hoped would give our algorithm an advantage, since it stores the joined sets of the already executed queries. Unfortunately, there was no improvement for Tests 1 through 3. This was due to the fact that for joining the sets, expensive type queries had to be made. For example, consider the join of the result

Test No.	Test Case	min triples	naive (ms)	custom (ms)
1	random entities	3	**602**	1875
2	single entity	3	**131**	234
3	random entities	5	**313**	744
4	single entity	5	111	**64**

Table 6.6: Time measurements for query execution methods

sets `<actedIn> <Forrest_Gump>` and `<type> <wordnet_person>`. The second set is an highly expensive query, since its result set is of size larger than 1 million entities. After changing the implementation that non-type facts were executed first the execution time for test 4 was halved. This may be attributed to the fact that unique answers about a single entity can be found after adding only few non-type facts and therefore omitting expensive type queries. Based on these results, we did not further pursue this direction of research.

6.4 Evaluation Summary

In our first user study we find that labeling questions with absolute difficulty is not an easy task for humans, since the task has a subjective component. Nevertheless, as part of our study on question difficulty 4.3 we find that it is possible to capture this notion. The results of our experiments show that it is possible to standardize question difficulty to some extent by building a difficulty classifier that was able to correctly classify 66.4% of the questions. In a second user study we show that on questions generated by the system, the same classifier achieves moderate agreement with human difficulty judgments in terms of relative and absolute difficulty estimates. This is the first system that tackles this specific task, to the best of our knowledge and even though our estimates only moderately agree with human judgments, we consider our insights into question difficulty as valuable. We believe that our findings show the potential of future research in this area, where our system could serve as a baseline.

Chapter 7

Conclusion & Future Work

In this thesis we presented a novel approach for generating questions using semantic information from a knowledge graph. From a high-level point of view, our system takes a category and difficulty level as input and outputs a question in natural language. To achieve this, we obtained a category-entity mapping from Wikipedia, by building a category graph. From the set of entities we retrieve from the category graph, we select a target entity which is used as the answer to the question. Then, our question generation algorithm retrieves all facts from the knowledge graph and decides which triples are selected to form the question's clues. The question is then posed as a SPARQL query, which is queried to the knowledge graph to check for the uniqueness of the question's answer.

In addition, we developed a verbalization scheme, that turns the intermediate SPARQL query into a natural language question. We make use of a custom pattern-based technique, which mimics Jeopardy!-clue articulation of the question. Furthermore, we cater to question variety by finding meaningful paraphrases for relations and elaborate a novel method to find important types for a given entity.

For being able to estimate the difficulty of a question, we trained a logistic regression classifier on Jeopardy! training data, which could correctly classify 66% of training instances using ten-fold cross-validation. As features we identified the metrics of popularity of entities, selectivity of a relation and coherence of an entity pair, which are measured using statistical methods in conjunction with the knowledge graph. We then outlined an approach to integrate the difficulty estimate into the question generation process to guide the search for a question with the desired difficulty level.

To evaluate the effectiveness of our method, we conducted an extensive user study, which measured the agreement of human judges with our difficulty classifier. First,

we evaluated the classifier's performance in terms of relative question difficulty. For the ranking of a total of 150 question pairs, our classifier reached moderate agreement with human evaluators, and an average Kendall's τ of 0.563. Second, we measured the agreement in terms of absolute question difficulty estimation. After 150 questions were evaluated as being easy or hard by the participants of our study, the evaluators reached moderate agreement with the difficulty classifier, which was measured using Cohen's Kappa ($\kappa = 0.427$).

Future work could be conducted in the domain of extending the question difficulty classifier's features, to be able to handle questions that are not entirely composed of entities and classes. For the purpose of making questions more compelling, it would be interesting to find features to estimate difficulty on literals. Since estimating the exact number of inhabitants of, for instance, Berlin, is a very difficult task, it would be advantageous to conduct research on finding appropriate ranges for literals, as well.

In the context of question features, it would be beneficial to investigate the effect of different data sources to measure popularity of an entity. Usable resources could be the number of mentions in a large, annotated text corpora, the entity's Wikipedia article length, the number of search queries issued about the entity, etc.

Moreover, our method could be extended to pose questions with not only one entity as an answer, but for questions that ask for a set of entities. Another stage of research could be conducted on the impact of question difficulty for questions that are not made up of "star queries". This would enable us to formulate questions that ask for the *football player, whose spouse is a former Spice Girl*, for example.

Further research could address the granularity of difficulty classes. In contrast to the binary classifications, one could categorize questions into multiple hardness categories. A prominent example is Jeopardy!, where clues are presented in five difficulty levels. A further problem that could be inspected is measuring the amount that the question's language (verbalization) contributes to the difficulty. This would be highly related to measuring the reading difficulty of text. Work in that problem domain was discussed in Section 3.2.

As mentioned before, question difficulty is subjective. Therefore, it would be beneficial to find a way of personalizing the estimation scheme. It could be imagined to build a customized difficulty estimator that takes the domain knowledge of the player into account. Requiring the player beforehand to answer a questionnaire with domain specific questions of various difficulties to estimate her level of expertise, could be an approach to address this problem.

Additional improvements could be made in the verbalization scheme. To generate more natural sounding questions, research could be invested in finding an alternative to our template-based approach. If the resulting questions would be similar enough to those written by humans, these questions could even be used to train a question answering system.

A further application where our system could be the basis for future work is in the domain of finding fraudulent users in crowdsourcing communities. There our approach could generate multiple-choice answers, by generating semantically close distractors. The distractors could be generated by removing one or multiple triples from the question's query. A distractor, related to the answer could then be selected from the result set of this relaxed query. Distractors that are not related to the answer could be found as entities that are selected at random and have no triples in common with the correct answer. Having this automatically generated gold standard of question-to-answer mappings, it would be possible to statistically discriminate between users that just randomly click through the data and users who click on answers that are either correct or closely related to the correct answer. Here the assumption is, that users who more question correct on average, are more likely to seriously work on the present crowdsourcing task.

List of Figures

List of Tables

Appendix A

Poster shown at WWW 2015

The poster depicted in Figure A.1, was presented to attendants of the 2015 World Wide Web Conference, as part of the poster session.

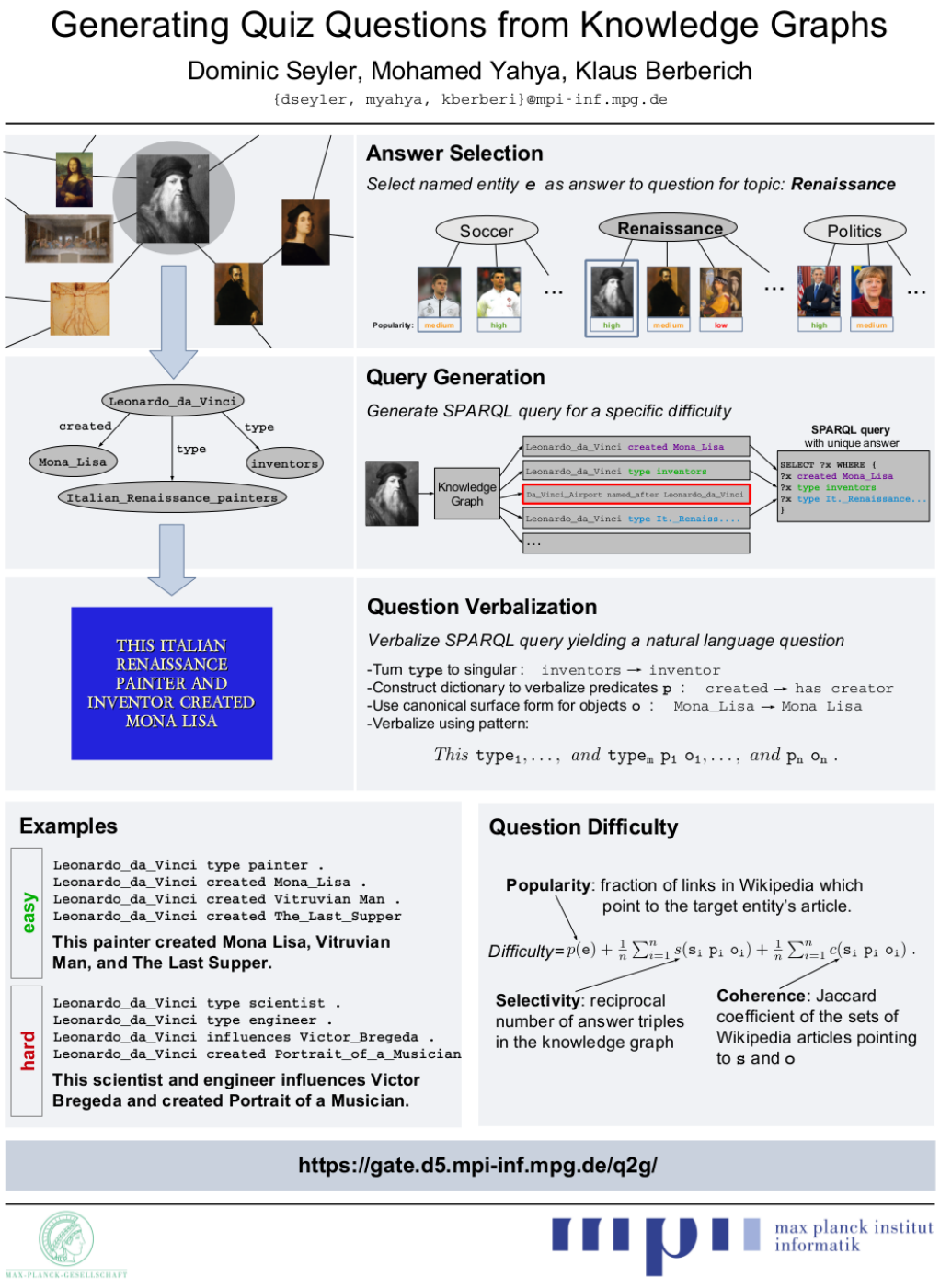

Figure A.1: Poster shown at the 2015 International World Wide Web Conference in Florence, Italy

Appendix B

Anecdotal Results

Category	`Theoretical_physicists`
Difficulty	easy
Target Entity	`<Albert_Einstein>`

Query	`SELECT ?x WHERE {`
	`?x <wasBornIn> <Ulm> .`
	`?x rdf:type <wordnet_physicist_110428004> .`
	`?x <worksAt> <ETH_Zurich>`
	`}`

Verbalization	*This physicist was born in Ulm and works at ETH Zurich.*
$P(easy)$	0.7394

Category	`Theoretical_physicists`
Difficulty	hard
Target Entity	`<Albert_Einstein>`

Query	`SELECT ?x WHERE {`
	`?x <hasAcademicAdvisor> <Alfred_Kleiner> .`
	`?x <isMarriedTo> <Mileva_Marić> .`
	`?x rdf:type <wordnet_physicist_110428004>`
	`}`

Verbalization	*This physicist is a student of Alfred Kleiner and is married to Mileva Marić.*
$P(easy)$	0.4951

Category	`Theoretical_physicists`
Difficulty	hard
Target Entity	`<Albert_Einstein>`

Query	`SELECT ?x WHERE {`
	`?x <influences> <Émile_Meyerson> .`
	`?x <worksAt> <University_of_Zurich> .`
	`?x rdf:type <wordnet_physicist_110428004>`
	`}`

Verbalization	*This physicist has influence on Émile Meyerson and works at University of Zurich.*
$P(easy)$	0.4310

Category	`American_billionaires`
Difficulty	easy
Target Entity	`<Bill_Gates>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_billionaire_110529684> .`
	`?x <created> <Microsoft> .`
	`?x <livesIn> <Medina,_Washington>`
	`}`

Verbalization	*This billionaire created the company Microsoft and lives in Medina, Washington.*
$P(easy)$	0.8345

Category	`American_billionaires`
Difficulty	hard
Target Entity	`<Bill_Gates>`

Query	`SELECT ?x WHERE {`
	`?x <created> <Cascade_Investment> .`
	`?x rdf:type <wordnet_billionaire_110529684> .`
	`?x <wasBornIn> <Seattle>`
	`}`

Verbalization	*This billionaire was born in Seattle and created the company Cascade Investment.*
$P(easy)$	0.4491

Category	`American_billionaires`
Difficulty	hard
Target Entity	`<Bill_Gates>`

Query	`SELECT ?x WHERE {`
	`?x <owns> <GAMCO_Investors> .`
	`?x rdf:type <wordnet_philanthropist_110421956> .`
	`?x <graduatedFrom> <Harvard_University>`
	`}`

Verbalization	*This philanthropist owns GAMCO Investors and is an alumnus of Harvard University.*
$P(easy)$	0.0862

Category	`American_folk_singers`
Difficulty	easy
Target Entity	`<Bob_Dylan>`

Query	`SELECT ?x WHERE {`
	`?x <created> <Knocked_Out_Loaded> .`
	`?x rdf:type <wordnet_singer_110599806> .`
	`?x <created> <Mr._Tambourine_Man>`
	`}`

Verbalization	*This singer created the album Knocked Out Loaded and the song Mr. Tambourine Man.*
$P(easy)$	0.9039

Category	`American_folk_singers`
Difficulty	easy
Target Entity	`<Bob_Dylan>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_musician_110340312> .`
	`?x <created> <Ain't_Talkin'> .`
	`?x <wasBornIn> <Duluth,_Minnesota>`
	`}`

Verbalization	*This musician was born in Duluth, Minnesota and created Ain't Talkin'.*
$P(easy)$	0.7920

Category	`American_folk_singers`
Difficulty	easy
Target Entity	`<Bob_Dylan>`

Query	`SELECT ?x WHERE {`
	`?x <created> <Tonight_I'll_Be_Staying_Here_with_You> .`
	`?x <influences> <Jason_Myles_Goss> .`
	`?x rdf:type <wordnet_songwriter_110624540>`
	`}`

Verbalization	*This songwriter has influence on Jason Myles Goss and created Tonight I'll Be Staying Here with You.*
$P(easy)$	0.6766

Category	`Pioneers_of_music_genres`
Difficulty	easy
Target Entity	`<Elvis_Presley>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_performer_110415638> .`
	`?x <created> <Jailhouse_Rock_(song)> .`
	`?x <created> <Viva_Las_Vegas_(song)>`
	`}`

Verbalization	*This performer created Jailhouse Rock and Viva Las Vegas.*
$P(easy)$	0.9241

Category	`Pioneers_of_music_genres`
Difficulty	easy
Target Entity	`<Elvis_Presley>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_musician_110340312> .`
	`?x <created> <Love_Me_Tender_(song)> .`
	`?x <created> <Blue_Christmas_(song)>`
	`}`

Verbalization	*This musician created the song Love Me Tender and Blue Christmas.*
$P(easy)$	0.7887

Category	`Pioneers_of_music_genres`
Difficulty	easy
Target Entity	`<Elvis_Presley>`

Query	`SELECT ?x WHERE {`
	`?x <influences> <John_ac_Alun> .`
	`?x <actedIn> <The_Trouble_with_Girls_(film)> .`
	`?x rdf:type <wordnet_artist_109812338>`
	`}`

Verbalization	*This artist acted in The Trouble with Girls and has influence on John ac Alun.*
$P(easy)$	0.6750

Category	`Internet_companies_of_the_United_States`
Difficulty	easy
Target Entity	`<Google>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_company_108058098> .`
	`?x <owns> <Freebase> .`
	`?x <created> <Android_(operating_system)>`
	`}`

Verbalization	*This company owns Freebase and created Android.*
$P(easy)$	0.5620

Category	`Internet_companies_of_the_United_States`
Difficulty	hard
Target Entity	`<Google>`

Query	`SELECT ?x WHERE {`
	`<Preston_McAfee> <worksAt> ?x .`
	`?x rdf:type <wordnet_company_108058098> .`
	`?x <owns> <Boston_Dynamics>`
	`}`

Verbalization	*This company owns Boston Dynamics and has employee Preston McAfee.*
$P(easy)$	0.0922

Category	`Internet_companies_of_the_United_States`
Difficulty	hard
Target Entity	`<Google>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_company_108058098> .`
	`?x <owns> <Neotonic_Software> .`
	`<Udi_Manber> <worksAt> ?x`
	`}`

Verbalization	*This company owns Neotonic Software and has employee Udi Manber.*
$P(easy)$	0.0623

Category	`19th-century_American_writers`
Difficulty	easy

Target Entity	`<Mark_Twain>`

Query	`SELECT ?x WHERE {`
	`?x <created> <The_Adventures_of_Tom_Sawyer> .`
	`?x rdf:type <wordnet_writer_110794014> .`
	`?x <created> <A_Conn._Yankee_in_King_Arthur's_Court>`
	`}`

Verbalization	*This writer created the novel The Adventures of Tom Sawyer and A Connecticut Yankee in King Arthur's Court.*
$P(easy)$	0.8734

Category	`19th-century_American_writers`
Difficulty	easy
Target Entity	`<Mark_Twain>`

Query	`SELECT ?x WHERE {`
	`?x <influences> <Ken_Kesey> .`
	`?x rdf:type <wordnet_writer_110794014> .`
	`?x <created> <Huckleberry_Finn>`
	`}`

Verbalization	*This writer influences Ken Kesey and created the fictional character Huckleberry Finn.*
$P(easy)$	0.8553

Category	`19th-century_American_writers`
Difficulty	hard
Target Entity	`<Mark_Twain>`

Query	`SELECT ?x WHERE {`
	`?x <influences> <Des_Dillon> .`
	`?x rdf:type <wordnet_writer_110794014> .`
	`?x <hasChild> <Jean_Clemens>`
	`}`

Verbalization	*This writer is the parent of Jean Clemens and has influence on Des Dillon.*
$P(easy)$	0.4342

Category	`Film_directors`
Difficulty	easy
Target Entity	`<Steven_Spielberg>`

Query	`SELECT ?x WHERE {`
	`?x rdf:type <wordnet_producer_110480018> .`
	`?x <directed> <Jaws_(film)> .`
	`?x <created> <E.T._the_Extra-Terrestrial>`
	`}`

Verbalization	*This producer created the movie E.T. the Extra-Terrestrial and directed Jaws.*
$P(easy)$	0.9317
Category	`Film_directors`
Difficulty	easy
Target Entity	`<Steven_Spielberg>`

Query	`SELECT ?x WHERE {`
	`?x <directed> <Catch_Me_If_You_Can> .`
	`?x <directed> <Jurassic_Park_(film)> .`
	`?x rdf:type <wordnet_manufacturer_110292316>`
	`}`

Verbalization	*This manufacturer directed Catch Me If You Can and Jurassic Park.*
$P(easy)$	0.9210
Category	`Film_directors`
Difficulty	easy
Target Entity	`<Steven_Spielberg>`

Query	`SELECT ?x WHERE {`
	`?x <directed> <The_Terminal> .`
	`?x rdf:type <wordnet_film_director_110088200> .`
	`?x <directed> <Minority_Report_(film)>`
	`}`

Verbalization	*This film director is the director of The Terminal and Minority Report.*
$P(easy)$	0.8944

Table B.1: Example questions generated by the system

Bibliography

[1] David A. Ferrucci, Eric W. Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya Kalyanpur, Adam Lally, J. William Murdock, Eric Nyberg, John M. Prager, Nico Schlaefer, and Christopher A. Welty. Building Watson: An Overview of the DeepQA Project. *AI Magazine*, 31(3):59–79, 2010.

[2] W3c Semantic Web Activity Homepage. URL http://www.w3.org/2001/sw/.

[3] Tim Berners-Lee, James Hendler, Ora Lassila, and others. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[4] Larry Masinter, Tim Berners-Lee, and Roy T. Fielding. Uniform Resource Identifier (URI): Generic Syntax. URL https://tools.ietf.org/html/rfc3986.

[5] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. URL http://www.w3.org/TR/rdf-sparql-query/.

[6] What is knowledge base? URL http://searchcrm.techtarget.com/definition/knowledge-base.

[7] George A. Miller. Wordnet: A Lexical Database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[8] Niket Tandon, Gerard de Melo, Fabian M. Suchanek, and Gerhard Weikum. WebChild: harvesting and organizing commonsense knowledge from the web. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*, pages 523–532. ACM, 2014.

[9] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. Yago: A Core of Semantic Knowledge. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 697–706, New York, NY, USA, 2007. ACM.

[10] Kurt D. Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of*

Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008, pages 1247–1250. ACM, 2008.

[11] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. DBpedia: A Nucleus for a Web of Open Data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007.*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer, 2007.

[12] Linked Data - Connect Distributed Data across the Web. URL http://linkeddata.org/.

[13] Wikipedia, July 2015. URL https://en.wikipedia.org/w/index.php?title=Wikipedia&oldid=673543003. Page Version ID: 673543003.

[14] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, Edwin Lewis-Kelham, Gerard de Melo, and Gerhard Weikum. YAGO2: exploring and querying world knowledge in time, space, context, and many languages. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011, Hyderabad, India, March 28 - April 1, 2011 (Companion Volume)*, pages 229–232. ACM, 2011.

[15] Johannes Hoffart, Mohamed Amir Yosef, Ilaria Bordino, Hagen Fürstenau, Manfred Pinkal, Marc Spaniol, Bilyana Taneva, Stefan Thater, and Gerhard Weikum. Robust Disambiguation of Named Entities in Text. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing, EMNLP 2011, 27-31 July 2011, Edinburgh, UK*, pages 782–792. ACL, 2011.

[16] Peter Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, 2012. ISBN 978-1-139-57979-7.

[17] Peter Russell, Stuart; Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003. ISBN 978-0137903955.

[18] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN 0-387-31073-8.

[19] Jürgen Groß. The Linear Regression Model. In *Linear Regression*, number 175 in Lecture Notes in Statistics, pages 33–86. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-40178-0 978-3-642-55864-1.

[20] Logistic Regression, July 2015. URL https://en.wikipedia.org/w/index.php?title=Logistic_regression&oldid=671306573. Page Version ID: 671306573.

[21] David W. Hosmer Jr, Stanley Lemeshow, and Rodney X. Sturdivant. *Applied Logistic Regression, 3rd Edition*. John Wiley & Sons, 3 edition, 2013. ISBN 978-1-118-54835-6.

[22] P. McCullagh and John A. Nelder. *Generalized Linear Models, Second Edition*. CRC Press, 1989. ISBN 978-0-412-31760-6.

[23] Saskia Le Cessie and Johannes C Van Houwelingen. Ridge estimators in logistic regression. *Applied Statistics*, pages 191–201, 1992.

[24] Standardization vs. normalization | Data Mining Research. URL http://www.dataminingblog.com/standardization-vs-normalization/.

[25] Ian H. Witten, Eibe Frank, and Mark Andrew Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Elsevier/Morgan Kaufmann Publishers, 2011. ISBN 978-0-12-374856-0.

[26] Chris Snijders, Uwe Matzat, and Ulf-Dietrich Reips. Big data: Big gaps of knowledge in the field of internet science. *International Journal of Internet Science*, 7(1):1–5, 2012. URL http://iscience.deusto.es/wp-content/uploads/2012/08/ijis7_1_editorial.pdf.

[27] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *science*, 332(6025):60–65, 2011. URL http://www.sciencemag.org/content/332/6025/60.full.

[28] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51:107–113, 2008.

[29] IBM - What is the Hadoop Distributed File System (HDFS) - United States, 2015. URL http://www-01.ibm.com/software/data/infosphere/hadoop/hdfs/.

[30] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1099–1110. ACM, 2008.

[31] The clueweb12 dataset. URL http://www.lemurproject.org/clueweb12.php/.

[32] Vasile Rus, Brendan Wyse, Paul Piwek, Mihai C. Lintean, Svetlana Stoyanchev, and Cristian Moldovan. The First Question Generation Shared Task Evaluation Challenge. In *INLG 2010 - Proceedings of the Sixth International Natural Language Generation Conference, July 7-9, 2010, Trim, Co. Meath, Ireland*. The Association for Computer Linguistics, 2010.

[33] Keisuke Sakaguchi, Yuki Arase, and Mamoru Komachi. Discriminative Approach to Fill-in-the-Blank Quiz Generation for Language Learners. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, ACL 2013, 4-9 August 2013, Sofia, Bulgaria, Volume 2: Short Papers*, pages 238–242. The Association for Computer Linguistics, 2013.

[34] Annamaneni Narendra, Manish Agarwal, and Rakshit shah. Automatic Cloze-Questions Generation. In *Recent Advances in Natural Language Processing, RANLP 2013, 9-11 September, 2013, Hissar, Bulgaria*, pages 511–515. RANLP 2011 Organising Committee / ACL, 2013.

[35] Igor Labutov, Sumit Basu, and Lucy Vanderwende. Deep Questions without Deep Understanding. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers*, pages 889–898. The Association for Computer Linguistics, 2015.

[36] Maha M. Al-Yahya. OntoQue: A Question Generation Engine for Educational Assesment Based on Domain Ontologies. In *ICALT 2011, 11th IEEE International Conference on Advanced Learning Technologies, Athens, Georgia, USA, 6-8 July 2011*, pages 393–395. IEEE Computer Society, 2011.

[37] Jing Liu, Quan Wang, Chin-Yew Lin, and Hsiao-Wuen Hon. Question Difficulty Estimation in Community Question Answering Services. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, EMNLP 2013, 18-21 October 2013, Grand Hyatt Seattle, Seattle, Washington, USA*, pages 85–90. ACL, 2013.

[38] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill™ : A Bayesian Skill Rating System. In *Advances in Neural Information Processing Systems 19*, pages 569–576. MIT Press, 2007.

[39] Jiang Yang, Lada A. Adamic, and Mark S. Ackerman. Competing to Share Expertise: The Taskcn Knowledge Sharing Community. In *ICWSM*, 2008.

[40] Kevyn Collins-Thompson and Jamie Callan. Predicting reading difficulty with statistical language models. *Journal of the American Society for Information Science and Technology*, 56(13):1448–1462, November 2005.

[41] Michael Heilman, Kevyn Collins-Thompson, and Maxine Eskenazi. An Analysis of Statistical Models and Features for Reading Difficulty Prediction. In *Proceedings of the Third Workshop on Innovative Use of NLP for Building Educational Applications*,

EANL '08, pages 71–79, Stroudsburg, PA, USA, 2008. Association for Computational Linguistics.

[42] Axel-Cyrille Ngonga Ngomo, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber. Sorry, I Don'T Speak SPARQL: Translating SPARQL Queries into Natural Language. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 977–988. International World Wide Web Conferences Steering Committee, 2013.

[43] George Doddington. Automatic Evaluation of Machine Translation Quality Using N-gram Co-occurrence Statistics. In *Proceedings of the Second International Conference on Human Language Technology Research*, HLT '02, pages 138–145, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

[44] Basil Ell, Denny Vrandečić, and Elena Simperl. Spartiqulation: Verbalizing sparql queries. In *The Semantic Web: ESWC 2012 Satellite Events*, pages 117–131. Springer, 2012.

[45] Georgia Koutrika, Alkis Simitsis, and Yannis Ioannidis. Conversational Databases: Explaining Structured Queries to Users. 2009.

[46] Adam Lally, John M. Prager, Michael C. McCord, Branimir Boguraev, Siddharth Patwardhan, James Fan, Paul Fodor, and Jennifer Chu-Carroll. Question analysis: How Watson reads a clue. *IBM Journal of Research and Development*, 56(3):2, 2012.

[47] Aditya Kalyanpur, Siddharth Patwardhan, Branimir Boguraev, Adam Lally, and Jennifer Chu-Carroll. Fact-based question decomposition in DeepQA. *IBM Journal of Research and Development*, 56(3):13, 2012.

[48] Dominic Seyler, Mohamed Yahya, and Klaus Berberich. Generating Quiz Questions from Knowledge Graphs. In *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 113–114. ACM, 2015.

[49] E. Reiter and R. Dale. *Building Natural Language Generation Systems*. Studies in Natural Language Processing. Cambridge University Press, 2006. ISBN 9780521024518.

[50] Marti A. Hearst. Automatic acquisition of hyponyms from large text corpora. In *14th International Conference on Computational Linguistics, COLING 1992, Nantes, France, August 23-28, 1992*, pages 539–545, 1992.

[51] Jeopardy!, August 2015. URL https://en.wikipedia.org/w/index.php?title=Jeopardy!&oldid=674554775. Page Version ID: 674554775.

[52] Noah A. Smith, Michael Heilman, and Rebecca Hwa. Question generation as a competitive undergraduate course project. In *Proceedings of the NSF Workshop on the Question Generation Shared Task and Evaluation Challenge*, 2008.

[53] Jordan L. Boyd-Graber, Brianna Satinoff, He He, and Hal Daumé III. Besting the Quiz Master: Crowdsourcing Incremental Classification Games. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL 2012, July 12-14, 2012, Jeju Island, Korea*, pages 1290–1301. ACL, 2012.

[54] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer, 1993.

[55] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[56] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *Biometrics, Vol. 33*, pages 159–174, 1977.