Bachelor Thesis

Query Auto-Completion using an Abstract Language Model

Natalie Prange

October 28th, 2016

University of Freiburg
Faculty of Engineering
Department of Computer Science

Submission Date
28. 10. 2016

Reviewer
Prof. Dr. Hannah Bast

Supervisor
Prof. Dr. Hannah Bast

Declaration

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date

Signature

Contents

Abstract

In recent years, query auto-completion has become a common feature of search engines. While typing a query into a search engine's input field, the user receives suggestions as to how his query could be completed or extended. Most approaches towards this issue rely on query logs in order to provide meaningful completion suggestions. Sufficiently large and recent query logs, however, might not be available for researchers or search engines with a small user base. Moreover, purely relying on query logs means that queries which have not been seen before will not be suggested. In this bachelor thesis, we propose a novel approach using an abstract language model in which specific entities have been replaced by abstract categories. When at some point during query completion the language model suggests such a category as the most likely word to follow, it is replaced by an entity assigned to the same category. The possible entities are ranked using an entity prominence score and word vector similarity. The top k completions are offered to the user as completion suggestions. We analyzed our approach on three datasets. For each dataset, we compared different versions of our query auto-completion algorithm to a baseline given by a basic version of our algorithm. The results illustrate the role of different components of our algorithm and indicate a high quality of our completion suggestions.

Zusammenfassung

In den letzten Jahren hat Query Auto-Completion bei Suchmaschinen weite Verbreitung gefunden. Gibt ein Benutzer eine Suchanfrage in das Eingabefeld einer Suchmaschine ein, werden ihm sofort Vorschläge geliefert, wie er seine Anfrage vervollständigen oder erweitern könnte. Eine herkömmliche Herangehensweise ist die Verwendung von Query Logs um sinnvolle Vervollständigungen liefern zu können. Für Forscher oder Suchmaschinen mit einem kleinen Nutzerkreis kann es jedoch problematisch sein, Zugriff auf aktuelle und ausreichend große Query Logs zu bekommen. Das bloße Verwenden von Query Logs hat außerdem den Nachteil, dass Suchanfragen, die zum ersten Mal auftreten, meist nicht korrekt vervollständigt werden können. Die in dieser Bachelorarbeit vorgestellte Methode, benutzt ein abstrahiertes Sprachmodell, in dem spezifische Entitäten durch abstrakte Typen ersetzt wurden. Wenn das Sprachmodell einen solchen Typ als nächstes Wort vorschlägt, wird dieser mit einer Entität des entsprechenden Typs aufgefüllt. Die so gewählten Entitäten werden daraufhin mithilfe eines Prominenz-Scores und ihrer Wortvektor-Ähnlichkeit zu zuvor eingegebenen Entitäten sortiert. Die am höchsten bewerteten Entitäten werden dem Benutzer als mögliche Vervollständigungen der Suchanfrage vorgeschlagen. Wir evaluieren unsere Methode auf verschiedenen Datensätzen. Dabei vergleichen wir verschiedene Versionen unseres Algorithmus mit einer Baseline, die durch eine einfache Implementierung unseres Algorithmus gegeben ist. Die Ergebnisse veranschaulichen die Bedeutung einzelner Komponenten unseres Algorithmus und lassen auf eine hohe Qualität unserer Vervollständigungen schließen.

1. Introduction

All major web search engines nowadays provide query auto-completion (QAC), a feature which tries to predict the user's query and offers its predictions to the user as completion suggestions. The main purpose of this feature is to reduce the user's effort when inputting a query. This is done by minimizing the amount of required keystrokes, preventing spelling mistakes and assisting in formulating the query. To fulfill its objective, QAC must try to suggest the desired completion after a minimal amount of keystrokes at the highest position of the offered completion suggestions. A more formal description of the problem is given in section 1.1. In section 1.2 we explain the motivation behind this work. Section 1.3 gives a short overview over our approach.

1.1. Problem Description

A QAC algorithm receives an input string q_p typed by a user. This is the prefix of the query the user has in mind. Let Q_s be the set of possible query completion suggestions. Typically, the QAC algorithm then ranks the completion suggestions $q \in Q_s$ before offering the top k completions to the user. A completion is a match if it matches the query the user intended to type. It depends on the algorithm which completions are being included in Q_s. Some approaches - and most major web search engines - are robust against spelling mistakes. That is to say, that for example the prefix *"Barak O"* would still lead to the completion suggestion *"Barack Obama"*. Other approaches require q_p to be a prefix of q. Some algorithms also support mid-string completion which means that for example "*Albert Einstein*" would be a valid completion for the user input "*Einst*". While spelling correction is not a feature of the QAC algorithm described in this work, mid-string completion is.

1.2. Motivation

Most research done in this field focuses on QAC on the basis of query logs. These datasets, such as the publicly available AOL query log, are collections of anonymized user queries. If the query logs are large enough, they allow reasonable conclusions about the popularity of queries and can therefore be used to predict future queries of a user.

Sufficiently large query logs, however, are not always readily available for researchers or search engines with a small user base. Additionally, publicly available query logs tend to be outdated and therefore might not represent the popularity of a query correctly. Another restriction of QAC with query logs is that users often submit queries which have never been asked before and therefore cannot be predicted by just suggesting popular queries from a query log. In 2012, allegedly 16 to 20% of Google queries were asked for the first time[1].

The possibility of providing query completion suggestions independent of query logs is therefore an interesting topic to explore and subject of this work.

1.3. Our Approach

The objective of this bachelor thesis is to build a system for query auto-completion based on an abstract language model. Our focus is on whole questions instead of typical search engine queries, which are often just fragments of the question a user has in mind and lack proper sentence structure.

In our approach, we use a language model to predict which word a user will type next, given the words he has typed so far. A language model is a probability distribution learned over a set of word sequences. It can be used to assign a probability to a sequence of words or, as is the case here, to predict which word is most likely to follow a given word sequence. For example, given the word *how* a language model would typically assign a high probability to the words *many* or *much*. A more detailed description of what a language model does, is given in section 3.1. A problem that is frequently encountered when working with language models is data sparsity. This problem manifests itself in two ways:

1. A word sequence has not been observed in the training data. The language model therefore can not make any reasonable predictions for words following this sequence.
2. A word occurs infrequently in the training data and has never been observed following a certain word sequence. This word will not be predicted by the language model given the particular sequence.

As an example for case 1, consider the question prefix *"When did Jerry Maguire"*. Most likely, the language model can provide only few predictions for words following such a specific context. For case 2 consider the word sequences *"who played"* or *"why did"*. In both situations, the language model will most likely provide several suggestions for the next word, however, specific words such as *Nemo* or *Angelina Jolie* might not have been observed following this context and would therefore not be predicted.

[1]http://readwrite.com/2012/02/29/interview_changing_engines_mid-flight_qa_with_goog/#awesm=%7EoiNkM4tAX3xhbP

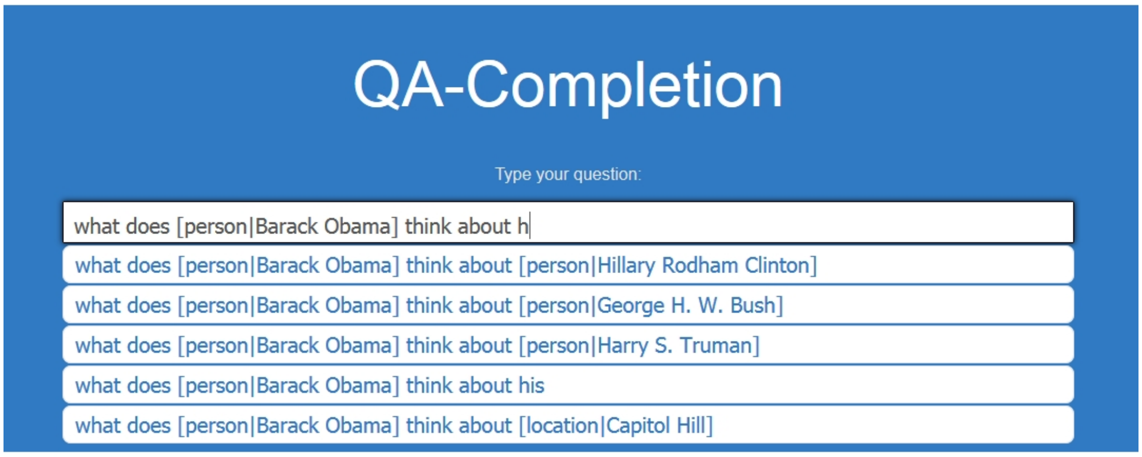
We tackle these issues by using an abstract language model. In this language model, specific entities have been replaced by abstract types. The question *"Who played Dory in Finding Nemo ?"* for example would translate to *"Who played [fictional character] in [film] ?"*

In order to learn such a language model, we use a set of questions in which Freebase entities have been identified. Freebase was an online database containing semantic data. It was shut down in May 2016, but data dumps are still available. For this work, the Freebase Easy [2] data dump was used. Freebase stores a variety of information for each of its entities. Yet, relevant for this work are besides the name of an entity only its unique machine identifier (MID) and its types. The entity *White House* for instance has the MID *m/081sq* and the types *Location, Building, Tourist attraction, Topic* among others. It is important to note, that although Freebase names are not necessarily unique, Freebase Easy names are. This is achieved by adding disambiguating suffixes to Freebase names if two or more entities share the same name. Freebase Easy names - which are used in this work - can therefore differ from the original Freebase names.

In the training questions for our language model we replaced identified Freebase entities with one of their types. Here, the type which is considered the most general but still meaningful categorization was chosen. An abstract language model, in which all entities have been replaced by the type *Topic* for instance would obviously be too general. On the other hand, a model in which *Wolfgang Amadeus Mozart* is assigned the type *Animal Owner* or *Albert Einstein* the type *Diet Follower* would hardly be meaningful. In the following, we will refer to these selected types as an entity's *category* to avoid ambiguity.

Our QAC algorithm uses the abstract language model to predict the words most likely to be typed, given a question prefix. If the predicted word is a category, the QAC algorithm selects the best matching entity out of all entities assigned to that category. This selection is done by using either a Freebase Easy prominence score or word vector similarity. Which of the two criteria is used depends on whether there exists enough context in the question typed so far to put word vectors to reasonable use. We built a demo web application for our approach. See Figure 1.1 for a screenshot of our demo.

We did an exhaustive empirical analysis of our approach. For this analysis, we evaluated the quality of the completion suggestions produced by our system on three different datasets. We tested several versions of our algorithm and compared the results to a baseline provided by a basic version of our algorithm. The results show a high quality of our completion suggestions. Even if the user has only entered one letter of the next word, our algorithm can often provide the desired word or entity on a high rank among its completion suggestions.



what does [person|Barack Obama] think about h

what does [person|Barack Obama] think about [person|Hillary Rodham Clinton]

what does [person|Barack Obama] think about [person|George H. W. Bush]

what does [person|Barack Obama] think about [person|Harry S. Truman]

what does [person|Barack Obama] think about his

what does [person|Barack Obama] think about [location|Capitol Hill]

Figure 1.1.: Our demo web application in action.

2. Related Work

In this section, we introduce some of the work that has been done in the field of QAC in recent years. Most approaches to QAC make use of query logs. Some of these approaches are summarized in section 2.1. An outline of the relatively little research that has been done on QAC without using query logs is given in section 2.2.

2.1. Query Auto-Completion using Query Logs

Most approaches to QAC are based on the exploitation of information provided by query logs. A common approach is to suggest queries on the basis of how often they have been observed in a query log. Context- or time-related information are then added to improve on the QAC results.

Whiting and Jose [12], for example, describe an approach in which they address the problem of providing query suggestions for both consistently and recently popular queries. To tackle this issue, they propose a variety of different algorithms. These include a sliding window approach where they only consider the popularity of queries over the last N days, a similar approach in which they vary the length N of the sliding window depending on how common the entered prefix is, and an approach in which they use short-range popularity trends in order to predict the current popularity of a query.

Bar-Yossef and Kraus [1] use recent queries of a user to produce more relevant completion suggestions. Their proposed algorithm suggests those queries from a query-log as completions for a user input, which are most similar to the user's previously typed queries. In order to measure similarity, they expand each query by adding related terms. Queries are then represented as term vectors and their similarity is computed as the cosine similarity.

2.2. Query Auto-Completion without Query Logs

Relatively few researchers so far have explored the possibilities of query-log-independent QAC. Among them are Bast and Weber [3, 4] who developed a search engine that features instant auto-completion. Completion suggestions are offered for the last incomplete word (as opposed to offering completion suggestions for the entire query the user might have in mind). The completion suggestions are terms occurring in the documents that their search algorithm yields, when searching for the preceding complete words of the query.

Bhatia et al. [5] propose a method that relies on an n-gram model built from the documents of the corpus that is being searched. N-grams extracted from these documents are used to complete a query prefix typed by a user. By using an n-gram model, this approach comes closest to what we are proposing in this thesis. A key difference is, however, that we use an abstract language model in which we insert specific entities. Moreover, the objective of the auto-completion is a different one: both approaches mentioned last aim mainly at finding certain information rather than predicting the user's input, as is the case in this work.

3. Components of the System

In this section, we discuss the basic functionality of the main components that make up our system. Along with that, we give a short description as to how these components were applied in our system and which libraries were used to integrate them into our implementation.

3.1. The Language Model

One of the main components of the system is an n-gram language model. As mentioned earlier, a language model is a probability distribution over a set of word sequences. A language model allows us to estimate the probability of a sequence of words

$$P(W) = P(w_1, w_2, ..., w_m)$$

or to estimate the probability of an upcoming word, given its predecessors

$$P(w_m | w_1, w_2, ..., w_{m-1})$$

The latter is how the language model is being used in this work. There exist various language models which use different methods to estimate these probabilities. One of the most commonly used models is the n-gram model. This model makes the simplifying assumption that the probability of a word in a word sequence does not depend on all prior words, but only on its preceding $n-1$ words. In an n-gram model the probability of a word w_m following the sequence $(w_1, ..., w_{m-1})$ would therefore be approximated as

$$P(w_m | w_1, ..., w_{m-1}) \approx P(w_m | w_{m-n}, ..., w_{m-1})$$

In our system, we use an n-gram model to predict which word the user is about to enter next. The implementation of our approach is entirely Python-based, hence, we use a Python library to include an n-gram model into our system. A popular Python library when it comes to processing natural language is the open source library *NLTK* (Natural Language Toolkit) [6]. Although the n-gram model implementation is not part of the official NLTK package at the time of writing this thesis, the code can be obtained via GitHub[1].

[1]https://github.com/nltk/nltk/tree/model/nltk/model (22.07.2016)

NLTK builds an n-gram model by first dividing training sentences into n-grams, that is, word sequences of size n. Sentences are padded from both ends with $n - 1$ padding symbols. For example, the 3-grams of for the sentence *"Who is Kappa ?"* would be

($, $, who), ($, who, is), (who, is, kappa), (is, kappa, ?), (kappa, ?, $), (?, $, $).

The padding at the sentence start and end is necessary in order to make predictions about the first word(s) in a sentence and about when a sentence is complete, respectively. The first $n - 1$ words of the n-grams are in the following called the n-gram context and form the keys for an NLTK *ConditionalFreqDist* object. This dictionary-like object maps each n-gram context to a frequency distribution of the words succeeding it. Using this data structure, a list of the words most likely to succeed a given n-gram context can be obtained in constant time. The probability for each word is then computed as the relative frequency of this word, given its n-gram context. This means, given the n-gram context c, each word w is assigned the probability

$$P(w|c) = \frac{count(c, w)}{count(c)}$$

where $count(\cdot)$ denotes the number of appearances of a given word sequence in the training data. This probability can be easily computed using the frequency distribution of the given n-gram context.

3.2. The Word Vectors

Word2vec [8] is a system which uses a two-layer neural network to learn vector representations of words. Word2vec takes a large text corpus as input and outputs a set of vectors, where each vector corresponds to one specific word. The more common context two words share, the more similar their vectors are in terms of cosine similarity. The learned word vectors can be used to compute semantic similarity between two words. A Word2vec model can therefore solve a variety of word association tasks. Consider for example a task such as "*man* is to *king* as *woman* is to x". Word2vec solves this kind of task by simply computing

$$vector(king) - vector(man) + vector(woman)$$

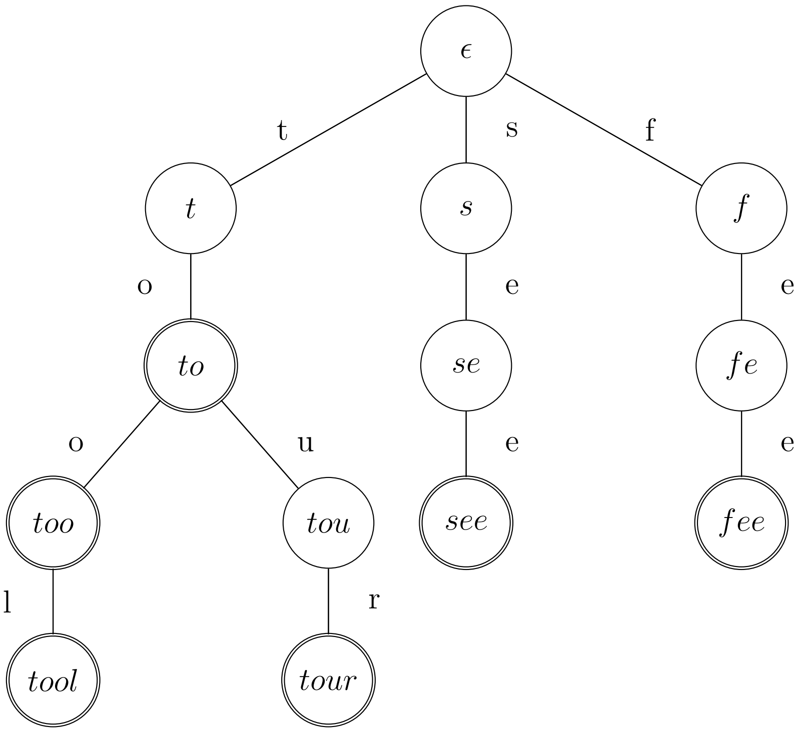
In a properly trained Word2vec model, the vector with the highest cosine similarity to the resulting vector will be $vector(queen)$.

In our system, we use a Word2vec model to decide which entity should be inserted for an abstract category predicted by our language model. In order to do so, we compute the similarity between entities already occurring in the question prefix and the entities of the predicted category. We use the Python library *Gensim* [9] to build the Word2vec model and to compute the cosine similarities.

3.3. The Tries

Efficiency is an important topic in QAC. It is essential that the user receives completion suggestions for his input instantaneously. Otherwise, one of the main objectives of QAC - reducing time and effort for inputting a query - can not be fulfilled. Typically, there are thousands of possibilities of how a user input can be completed. The QAC algorithm has to decide which of these possibilities are appropriate completions, rank them and return them - all within a fraction of a second. In our approach, the possible completions are limited to the predictions of the n-gram model. As soon as the user enters the prefix of the next word, the number of possible completions is further reduced to the number of predicted words or entities starting with the given prefix. However, if the n-gram model predictions include for example the category *Location*, there are more than 38,000 possible entities that have to be checked for prefix equivalence. For the category *Person* the number of possible entities is more than 64,000. Often, the n-gram model predictions include more than one category. In order to perform efficient QAC, we therefore need a data structure that allows fast retrieval of all its elements that start with a given prefix.

A trie, also known as prefix tree or radix tree, is such a data structure. Each edge in a trie is assigned a single letter. The accumulation of letters on each path from the root to a leaf node is a word from which the trie was built. That way, each path in the trie represents a prefix. Nodes which share a common parent node, also share a common prefix. The following is a trie built from the words *to, too, tool, tour, see* and *fee*:



Complete words, that is, words that were used to build the trie, are marked by a double circle. Note, that a complete word does not necessarily have to be a leave node (see the words *to* and *too* in the example trie). If every node has access to a list of its successive complete words, all words that start with a given prefix can be retrieved in time $O(M)$, where M is the maximum length of any inserted word.

In this work, we use tries for two different purposes. The first purpose is the efficient retrieval of entities of a given category whose names start with a given prefix. For this, we use a dictionary which maps each category to a trie that contains all entities assigned to this category. See section 4.6 for a more detailed description. The second purpose is the retrieval of normal words (that is, non-entities) that start with a given prefix. Here, we use a single trie containing all normal words of the n-gram language model vocabulary. More on this in section 4.8. We use the trie implementation of the Python package datrie[2] to integrate the tries into our algorithm.

[2]https://pypi.python.org/pypi/datrie

4. Query Auto-Completion using an Abstract Language Model

In this chapter, we illustrate our approach in detail. First, we provide an intuition for what our system does by outlining the user experience of our system in action. We then discuss the various steps that are necessary in order to achieve the described user experience. We start by explaining the necessary preprocessing, namely building the abstract language model and the Word2vec model. This is followed by a comprehensive description of the QAC algorithm itself. Figure 4.1 gives an overview of the basic pipeline of our algorithm. Each element of the pipeline will be described in the following sections.

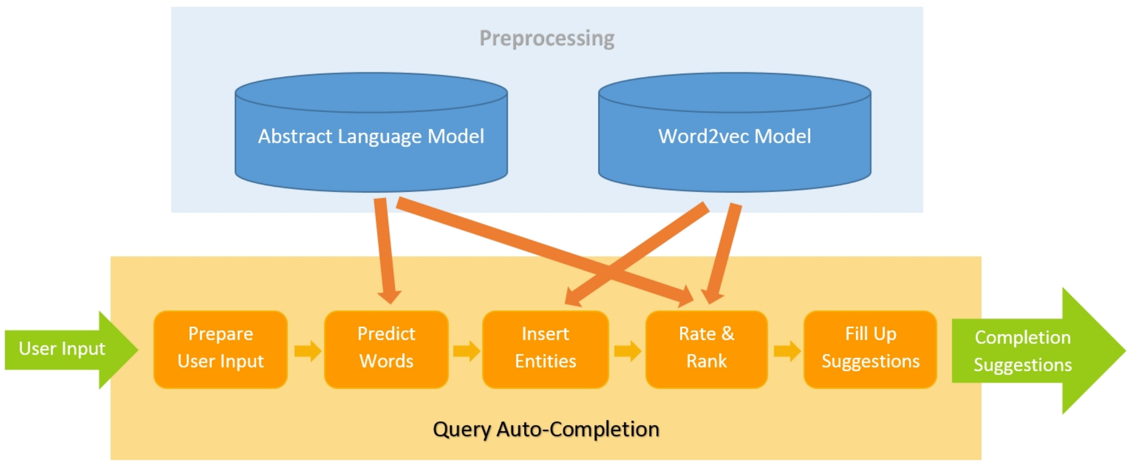
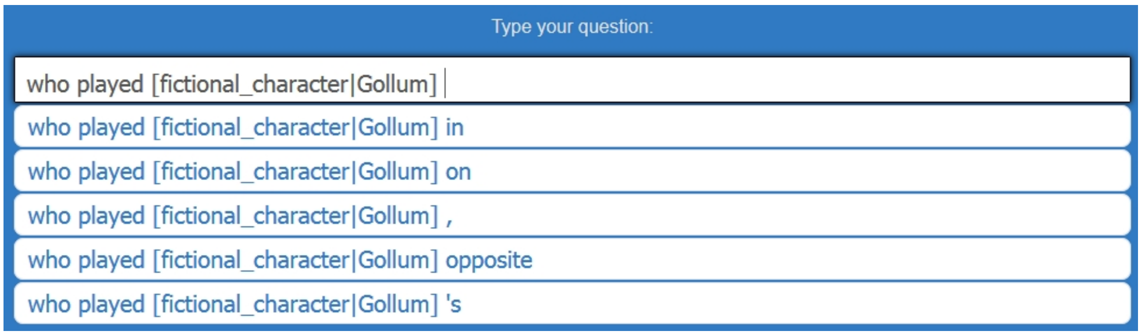


Figure 4.1.: The basic pipeline of our algorithm.

4.1. User Experience

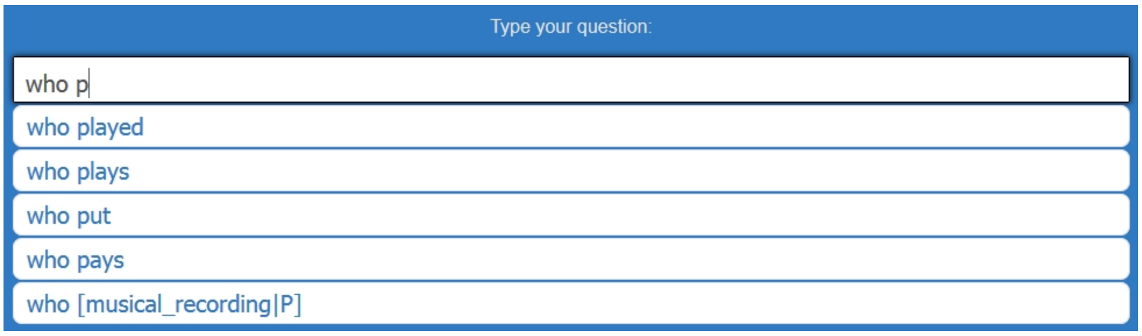
We built a web application for demonstrative purposes. After each key-stroke, the user immediately receives a predefined number of completion suggestions. The completion suggestions can either be suggestions for the next word or entity name, or suggestions for the current word or entity name. The former is the case if the last entered character is a space and the last fully typed word(s) are not expected to be part of an entity name. An example for this is shown in Figure 4.2. The latter ap-



who played [fictional_character|Gollum]
who played [fictional_character|Gollum] in
who played [fictional_character|Gollum] on
who played [fictional_character|Gollum] ,
who played [fictional_character|Gollum] opposite
who played [fictional_character|Gollum] 's

Figure 4.2.: Completion suggestions for the next word.

plies if the character entered last is not a space and the word is therefore expected to be incomplete (Figure 4.3). It also applies if the word(s) which have been typed last



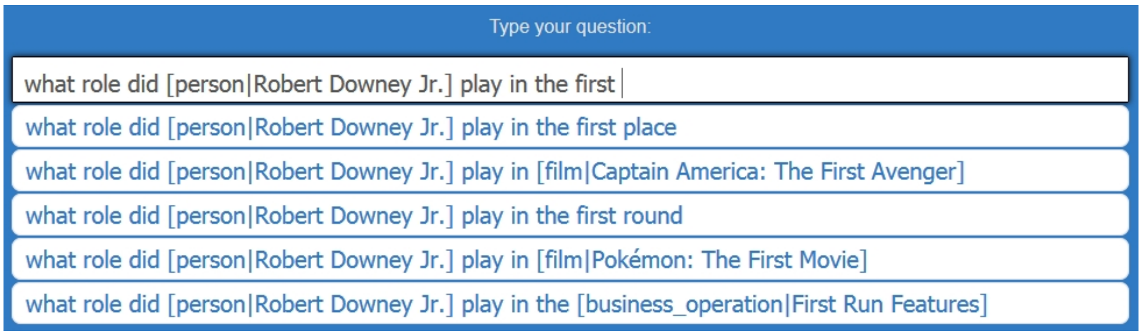
who p
who played
who plays
who put
who pays
who [musical_recording|P]

Figure 4.3.: Completion suggestions for the current word or entity name.

are likely to be part of an entity name (Figure 4.4). As can be seen in the examples, entities are suggested in the format *[<category>|<name>]*. Consequently, entities are only recognized as such by the algorithm if they appear in that same format in the question prefix. If for example the user enters the question prefix *"Why did Angela Merkel"* and does not select the offered completion suggestion *"Why did [person|Angela Merkel]"* but instead keeps on typing, *Angela Merkel* will not be treated as an entity in the resulting question prefix.

4.2. Building the Abstract Language Model

In order to build an abstract language model, a training set is needed that consists of questions in which Freebase Easy Entities have been identified and marked. In the training set we use, identified entities appear in the form *[<MID>|<original_word>]*. To create an abstract language model, each entity has to be assigned to a single category.


Type your question:

what role did [person|Robert Downey Jr.] play in the first

what role did [person|Robert Downey Jr.] play in the first place
what role did [person|Robert Downey Jr.] play in [film|Captain America: The First Avenger]
what role did [person|Robert Downey Jr.] play in the first round
what role did [person|Robert Downey Jr.] play in [film|Pokémon: The First Movie]
what role did [person|Robert Downey Jr.] play in the [business_operation|First Run Features]

Figure 4.4.: Completion suggestions for the next word and current entity name.

Choosing Categories. The first step in choosing a category for each entity, was to connect each Freebase Easy entity name with a list of types assigned to the entity. The files facts.txt and freebase-links.txt from the Freebase Easy data dump were used for this purpose. The freebase-links file provides a mapping from Freebase Easy entity names to Freebase links. These links contain the MID of the corresponding entity. Therefore this file can be used to extract a mapping from Freebase Easy names to MIDs. The facts file provides, among other information, a mapping from Freebase Easy names to their types via an *is-a* relation. Since Freebase Easy names are unique, these two mappings are sufficient to connect each MID with a list of corresponding types.

The second, and more challenging task, was to choose one out of all the types connected with an entity. The objective was to select a type which is as general as possible while still providing meaningful, distinguishing information about the specific entity. For example, a collection of types of the Freebase entity *Albert Einstein* would be *Astronomer, Diet follower, Literature Subject, Person* and *Topic*. Clearly, there are some types which are better suited as category than others. *Astronomer* is a meaningful, but not very general categorization. *Literature Subject* and *Topic* on the other hand are quite general, but don't provide meaningful information about the entity. And finally, *Diet follower* is neither meaningful nor general. In this case, *Person* would be the category of choice, as it is the most general category which still provides meaningful information.

The initial approach to this issue was to make a choice solely dependent on type frequencies. First, for each type, the number of total occurrences in entity type lists was counted. Then, for each entity, the type with the most occurrences was chosen out of the entity's type list. One further adjustment was made by excluding the type *Topic* from the type lists, as nearly all entities have this type. This approach is a fairly good guarantor of generality, however, meaningfulness is often not achieved. Using this way of categorization, the entity *Facebook*, for instance, is assigned the category *Award Nominee* instead of e.g. *Website*, and *JavaScript* becomes a *Literature Subject* instead of a *Programming Language*.

The second approach was to use a hand-picked, sorted list of preferred types that would be chosen if they existed in an entity's type list. If more than one of the preferred types occurred in a type list, the type with the higher preference would be chosen. Only if none of the hand-picked types existed in a type list, the initial approach of using type frequencies was used. Additionally, we used a hand-picked, sorted list of undesired types. Types in that list would not be chosen, even if they had a higher frequency than other types, unless they were the only type in an entity's type list. The two lists of preferred and undesired types can be found in the appendix.

Learning the Language Model. In order to learn the abstract language model, the identified entities in the training questions are replaced by the category assigned to them. Some MIDs in the training questions can not be assigned to a Freebase Easy name. In these cases, the entities are replaced by an *unknown*-tag. Nonetheless, the questions are not excluded from the dataset since they can still provide valuable information about general sentence structure. The resulting questions are prepared for the learning process by dividing them into a list of words and removing special characters such as brackets and quotation marks. Additionally, all words are cast to lower case. An n-gram model is then learned using *NLTK* as described in section 3.1. The best results have been achieved using a 4-gram model.

4.3. Building the Word2vec Model

The Word2vec model was built using *Gensim* as discussed in section 3.2. The training had to be done on a text corpus in which Freebase Easy entities have been identified and marked and which is large enough to have a sufficient amount of occurrences of most prominent entities. We used a recent Wikipedia dump with identified entities for this purpose.

This concludes the necessary preprocessing for our system. In the following sections, we will take a closer look at the algorithm that generates the completion suggestions.

4.4. Preparing the User Input

The first step of the QAC algorithm is the tokenization of the question prefix entered by the user. Here, the question prefix is split into single words and entities. At the same time, special characters are filtered out and words are transformed to lower case. Entities are being excluded from this process, since otherwise, entities that contain special characters can not be identified correctly. The resulting list of tokens is padded from the left with $n-1$ padding symbols.

4.5. Predicting Words

The next step is the prediction of the n-gram model. Predictions are made for the next or current word, given an n-gram context and a (possibly empty) prefix of that word. All identified entities in the n-gram context, that is, all tokens of the form *[<category>/<name>]*, are replaced by their category, as only the categories and not the entity names appear in the n-gram model. The prediction itself would be straight forward if we did not have to identify entities. The n-gram context would consist of the last $n-1$ fully typed words and the prefix would be the last word not followed by a space character. The n-gram model would then predict the words that are most likely to succeed the n-gram context and that start with the given prefix. In our case, however, a single word in the n-gram model, e.g. *[film]* can correspond to multiple words typed by the user, e.g. *"the lord of the rings"*. In order to identify such multi-word entities correctly, it is not sufficient to just consider the last $n-1$ complete words as n-gram context for the predictions and the last word not followed by a space character as the prefix. Instead, it is required to make predictions for each previous $n-1$-word context, where the concatenation of the following words is possibly an entity prefix. We assume this holds true for each prefix which does not contain a recognized entity and whose n-gram context does not contain an entity as the last word. This is because in any question an entity does rarely ever directly precede another one. Normal word predictions can be excluded from the predictions for previous n-gram contexts, as normal words per definition never consist of multiple words. All steps described in the following sections are performed separately for each n-gram context and corresponding prefix. The predicted words are sorted by their n-gram probability. This is, as discussed in section 3.1

$$P(w|c) = \frac{count(c, w)}{count(c)}$$

Where w is the predicted word and c is its n-gram context. Unknown words such as the *unknown*-tags for unknown categories are omitted from this and any further processing.

4.6. Inserting Entities

In the next step, we insert entities for all categories predicted by the language model. In order to qualify as a possible insertion entity, an entity has to be assigned to the predicted category and match the given prefix. For efficient retrieval of matching entities, we use a dictionary which maps each category to a trie. This trie holds all entities assigned to the specific category that have a Freebase Easy prominence score of at least 1,000. The Freebase Easy prominence score is a combination of entity occurrences in the ClueWeb'12 Corpus [1] and Freebase relation in- and out-degree

[1] http://lemurproject.org/clueweb12/

counts. To get a feeling for the dimensions: the highest score - held by the entity *United States of America* - is 81,032,073.1196. Entities with a score of less than 1,000 are expected to be rarely typed by a user. Excluding them therefore saves computation time and does not significantly impair the quality of the completion suggestions.

Since we want to provide mid-string completion for entities (*"Einst"* should lead to the completion suggestion *"Albert Einstein"*), we insert all rotations of an entity name into the corresponding trie. For the above example we would add *Albert Einstein* and *Einstein Albert* into the trie for the category *Person*.

4.7. Rating and Ranking

The inserted entities and the predicted normal words are then rated. The rating can be done in two different ways, depending on the question typed by the user so far. In the first scenario, the question prefix does not contain any entity. Consider for example the prefix *"Where did"* or *"Why are so many"*. Such sentences rarely provide any information about the specific entity the user has in mind other than its category. The rating is therefore done using the Freebase Easy prominence score. The score of each entity that matches a predicted category is first normalized to take on a value between 0 and 1. This is done by applying the formula

$$s_{norm} = \frac{s - s_{min}}{s_{max} - s_{min}}$$

where s is the original Freebase Easy score, $s_{min} = 1,000$, the lowest score of any suggested entity, and $s_{max} = 81,032,073.1196$, the highest entity score. The final score is computed in the following way:

$$s_{final} = p_{ngram} * (s_{norm})^{0.3}$$

where p_{ngram} is the probability computed by the n-gram model. Normal words are assigned a score $s = 100,000$ and then treated like entities. This way of computing the final score has been found to, in many cases, lead to better results than a logarithmic approach. The specific values for the exponent and the normal word score have been determined by experimentation.

In the second scenario, the question prefix contains at least one entity. This one entity can already provide important information about the second entity to be typed. Consider for example the prefix *"Who played [fictional_character/Gollum] in "* and the n-gram model prediction *[film]*. Anyone who is familiar with the Lord of the Rings trilogy will know that the movie *The Lord of the Rings: The Return of the King* is much more likely to be the required entity than Quentin Tarantino's *Inglourious Basterds*. To rate possible next entities, some formal way of measuring similarity between the already typed entities and the possible next entity is needed. The Word2vec model provides such a similarity estimate. For each predicted category, the algorithm does the following: For every entity of the category it computes the Word2vec similarity between the entity name and the names of the entities in the question prefix. The final score is then computed in the same way as in the first scenario. The only difference is that s_{norm} is replaced by the Word2vec similarity:

$$s_{final} = p_{ngram} * (s_{simil})^{0.3}$$

Normal words are assigned a similarity $s_{simil} = 0.7$.

Entities and words are then ranked by their score.

4.8. Filling Up the Completion Suggestions

If there are less than k matching entities and words, where k is the number of completions that should be offered to the user, the completion suggestions are filled up. For this fill-up, we use normal words and entities that match the given prefix but which (whose categories) have not been predicted by the language model. Here, too, tries are used. This time, however, not only for the entities but also for normal words. Otherwise the whole n-gram vocabulary would have to be inspected instead of just the words which match the given prefix. The scoring for the fill-up entities is done using the normalized Freebase Easy score. For normal words, we use a normalized frequency count. The normalization is similar to that of Freebase Easy scores:

$$c_{norm} = \left(\frac{c - c_{min}}{c_{max} - c_{min}}\right)^{0.5}$$

where c denotes the number of occurrences of the word in the training data, c_{min} is the count of the least common word in the training data, and c_{max} the count of the word with the most occurrences. These fill-up words and entities are sorted by their score and appended to the completion suggestions.

Entities whose names have been typed completely, such that the current prefix is the entity name, are always added to the completion suggestions offered to the user. This is to ensure that entities are reliably recognized by the algorithm. Recall, that an entity in the question prefix is only recognized by the algorithm, if it appears in the format *[<category>/<name>]*. It can be difficult for a user to enter an entity in this format by hand as he might not know which category an entity was assigned to. Hence, an entity which is not suggested by the algorithm, even after its entire name was typed, will not be recognized as an entity in the question prefix. This has typically a big influence on the quality of any following completion suggestions. The importance of proper entity recognition is best illustrated by an example. Consider the question prefix *"When did Bill Clinton become "*. In this case, the n-gram context for a 4-gram language model would be (*bill*, *clinton*, *become*). This context, however, will most likely never appear in the training data, where the question prefix would translate to *"When did [person] become "*. The language model would therefore be unable to provide useful predictions for the next word. The same holds true for all $n - 1$ words succeeding the unidentified entity. Appending an entity to the completion suggestions once its name has been typed completely, plays an important role in reducing the number of unrecognized entities.

Once the completion suggestions have been selected, ranked and filled-up the top k completions are offered to the user.

5. Evaluation

We conducted a thorough evaluation of our system. For this evaluation, we trained a language model on over one million abstract questions and learned a Word2vec model on a recent Wikipedia dump. The training process and the resulting models are described in section 5.1. We tested our system on three different datasets, one containing 100,000 questions and the other two containing 10,000 questions each. A detailed description of the test sets is given in section 5.2. We compare four different versions of our algorithm which are discussed in section 5.3. The evaluation metrics used to compare our results are described in section 5.4. The results are discussed in section 5.5.

5.1. Training Sets

The same abstract language model was used for every evaluation. The training set for building the language model consists of 1,407,979 questions extracted from the ClueWebb12 Facc1 corpus [7]. This corpus comprises 456,498,584 English web pages in which Freebase entities have been identified and marked. The precision of the entity recognition is estimated in [7] to be around 80-85% and the recall around 70-85%. Each question in the training set starts with one of the question words *what, where, when, who, why, how* and ends with a question mark. Moreover, each question contains at least one entity and not more than 50 words. In the training questions, identified Freebase entities were replaced by their category. Learning the language model takes on average 3.4 minutes. The learned model has a space consumption of 2.4GB.

We used the same Word2vec model for every evaluation. The model was trained on a recent Wikipedia dump. This Wikipedia dump contains 140,879,947 sentences of articles from the English Wikipedia in which Freebase Easy entities have been identified and marked. We assigned a size of 200 to the layers of the neural network that was used to train the the Word2vec model. During training, only words which occurred at least 20 times in the training data were considered. Entity names were treated as one single word. The training took 10.3 hours using 5 workers on a machine with 8 processing units available. The resulting Word2vec model consists of 2,240,478 200-dimensional word vectors. Loading the learned Word2vec model into RAM takes on average 15.1 seconds where it uses 4.9GB of memory.

Building the tries takes 19 seconds on average. Their space consumption is with less than 50MB negligible. Additionally, some indices have to be build. All in all, the initialization phase of our system takes on average 4.1 minutes, if the necessary files exist. After this initialization phase, the system is fully operable. The system then uses 7.4GB of RAM.

5.2. Test Sets

The test questions were taken from the ClueWebb12 Facc1 dataset described in section 5.1 and match the same conditions as the training questions for the abstract language model. In the test questions however, identified entities were replaced by their name. Questions containing an MID which could not be assigned to any entity were not included in the test sets. The test questions were transformed to lower case and special characters were removed. Entity names were excluded from this process. We created three test sets to conduct our experimentation on. They differ in terms of the number of entities appearing in each question. Test set 1 comprises 100,000 questions with varying numbers of entities. The test sets 2 and 3 both contain 10,000 questions each. Test set 2 consists of questions, containing exactly one entity. Test set 3 consists of questions, containing at least two entities. This differentiation has been made mainly to examine the influence of word vectors on completion suggestions, as the Word2vec model is only used if a question contains more than one entity.

5.3. Tested Versions of the Algorithm

We tested three different versions of our algorithm against a baseline. The goal of this comparison was to identify the importance of some of the features of our algorithm. The baseline is given by a basic version of our algorithm. This basic version differs from the algorithm explained in chapter 4 in two aspects: 1) it does not add completely typed entities to the completion suggestions, and 2) it does not fill up the completion suggestions with normal words and entities that have not been suggested by the n-gram model. The second version of our algorithm that is tested against this baseline fulfills 1) but does fill up the completion suggestions. The third version differs from the final algorithm in that it does not use the Word2vec model in order to rate entities. Instead, it only uses the Freebase Easy prominence score for that purpose. The fourth and final tested version is the algorithm as described in chapter 4.

For every evaluation, we used a 4-gram language model and provided 5 completion suggestions to choose from.

5.4. Evaluation Metrics

We use two different methods to evaluate our system. With the first method we measure how much user interaction is needed in order to get the desired output. We assume that the user selects the correct completion as soon as it is offered to him as a completion suggestion. Both, typing a letter and selecting a matching completion suggestio,n equally account for the number of user interactions. The resulting amount of user interaction is given as percentage over the length of the question. For this method, we also count entities as completion suggestion matches, which have not been recognized as such, but whose name is properly completed by normal words. If for example the words *brad* and *pitt* are both suggested before the entity *[person/Brad Pitt]* is, they are considered a match. We do, however, keep track of the number of unidentified entities and present the percentage of unidentified entities over all entities in the test set in the results.

The second method takes into account at which rank a matching completion suggestion is presented to the user. Mean Reciprocal Rank (MRR) is a measure commonly used to evaluate retrieval which aims at a single relevant item. In query auto-completion, MRR finds particularly wide-spread application as for example in [1, 10, 11, 12]. The idea behind MRR is that a higher ranked item is more helpful to the user than a lower ranked item, but the importance of the order decreases with lower ranks. This means, that the difference between an item being ranked at for example the 1st or 2nd position is bigger than the difference of it being ranked at the 4th or 5th position. Given an ordered list of completion suggestions S that are offered to a user and the completion q_c the user has in mind, the reciprocal rank RR is computed as follows:

$$RR(q_c, S) = \frac{1}{rank(q_c, S)}$$

where $rank(q_c, S)$ denotes the rank that q_c occupies in S. If q_c does not appear in S at all, the reciprocal rank is defined as $RR(q_c, S) = 0$. We compute the reciprocal rank for each word of a question after having typed its first letter. The MRR is then computed as the mean over the reciprocal rank of every word of each question in the test set. Note that a small increase in MRR can be due to a large increase in completions with an $RR > 0$. To a user, however, even showing a matching completion suggestion at a low rank will be helpful as opposed to not offering any matching completion suggestion at all. Therefore, even small changes in the MRR should be considered.

5.5. Results

Table 5.1 shows the evaluation results for test set 1, which contains 100,000 questions with varying numbers of entities. As expected, the baseline algorithm performs worse than all other versions of the algorithm in terms of MRR, user interaction and the number of unidentified entities. Only the completion time is reduced by 43% compared to the second and fourth version of the algorithm. Since the impact on MRR and user interaction made by adding complete entities is rather small, the baseline results show that filling up entities has a big influence on the quality of the completion suggestions. This suggests, that data sparsity is still to some extent an issue in our approach.

Interestingly, while the MRR and the user interaction results of the baseline do not differ much on the two other datasets, the number of unidentified entities does. This is shown in Table 5.2 and Table 5.3. For questions containing only one entity, the number of unidentified entities is reduced by 34.1% compared to questions containing at least two entities. A possible explanation for this phenomenon is that the already high number of unidentified entities leads to even more unidentified entities in the same question. This is because the language model can rarely provide reliable predictions with an unidentified entity in the n-gram context.

Note, that in general the results between test set 1 and test set 2 vary less than those between 1 and 3. This is because 80% of all questions in test set 1 contain only one entity. The data in the first test set is therefore more similar to test set 2 than it is to test set 3.

All questions				
Algorithm Version	MRR	User Interaction	Unid. Entities	Time
Baseline	0.376	0.64	38.9%	0.027 secs
w/o complete entities	0.469	0.49	11.1%	0.047 secs
w/o Word2vec model	0.449	0.49	6.3%	0.040 secs
Complete algorithm	0.457	0.49	6.3%	0.047 secs

Table 5.1.: The evaluation results for test set 1, containing questions with varying numbers of entities per question. Shown are the MRR, the average user interaction required per question, the percentage of unidentified entities and the average completion time for four different versions of the algorithm.

When comparing version 2 of our algorithm (no appending of complete entities) with the complete algorithm, one can see that the MRR and the user interaction do not vary much. In fact, the version 2 algorithm even performs slightly better regarding the MRR. A big difference however, is notable in the number of unidentified entities.

This number is considerably lower when adding complete entities to the completion suggestions. A slight reduction of the MRR in turn for a significantly lower number of unidentified entities seems justifiable.

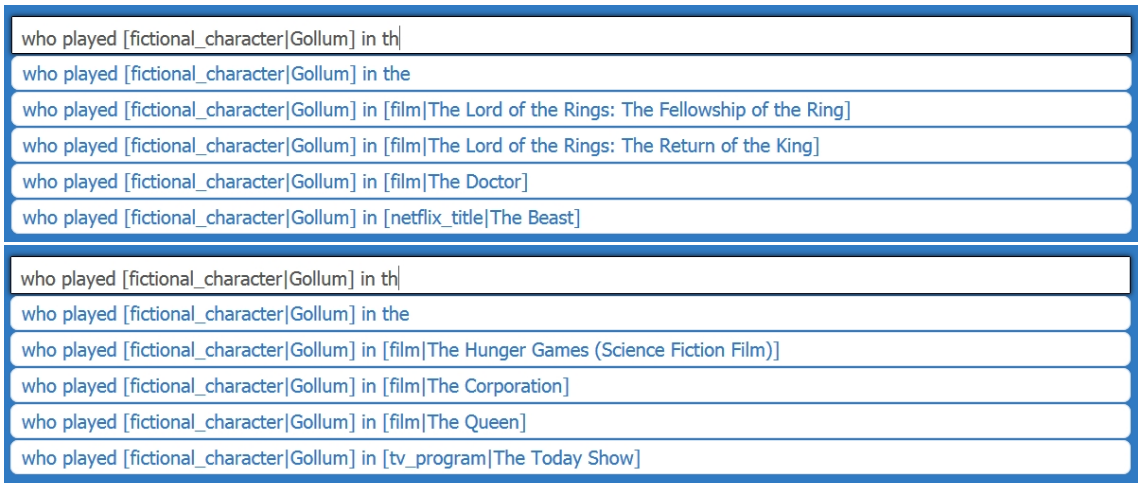
Questions containing one entity				
Algorithm Version	MRR	User Interaction	Unid. Entities	Time
Baseline	0.373	0.64	33.2%	0.028 secs
No complete entities	0.469	0.49	10.2%	0.048 secs
w/o Word2vec model	0.449	0.50	6.2%	0.041 secs
Complete algorithm	0.457	0.50	6.2%	0.047 secs

Table 5.2.: The evaluation results for test set 2, containing only questions with one entity.

Questions containing two or more entities				
Algorithm Version	MRR	User Interaction	Unid. Entities	Time
Baseline	0.385	0.66	50.4%	0.025 secs
No complete entities	0.465	0.49	15.7%	0.046 secs
w/o Word2vec model	0.444	0.47	6.7%	0.037 secs
Complete algorithm	0.452	0.48	6.8%	0.046 secs

Table 5.3.: The evaluation results for test set 3 which contains only questions with two or more entities.

A somewhat surprising result is, how little difference the MRR and the user interaction show between using Word2vec similarity and purely relying on the Freebase Easy prominence score for the rating of entities. Intuitively, one tends to expect considerably better results using Word2vec similarity. Especially when looking at the completion suggestions of both versions for the same question prefix as shown in Figure 5.1. We closely examined the completion suggestions of both versions for the same question prefix. We identified two scenarios in which using Word2vec similarity can be inferior to using the prominence score. In scenario 1, there is no strong correlation between the entities contained in the question prefix and the desired entity. Consider for example a question such as *"When will Linkin Park finally come to the Philippines ?"*. In this case, the Word2vec similarity between the two entities *Linkin Park* and *Philippines* is rather low. It is therefore likely, that the approach using the prominence score will yield the matching completion suggestion *Philippines* before the Word2vec approach does. In scenario 2, the desired entity is quite general and there are more specific entities which have a higher Word2vec similarity to the entities in the question prefix. An example for this scenario is the question *"What did Angela Merkel do for Germany ?"*. Using Word2vec similarity, entities such as *West Germany* or *Federal Constitutional Court of Germany* are suggested long before the desired entity *Germany* is. Overall, however, using Word2vec similarity does still lead to better results than using only the Freebase Easy prominence score.



```
who played [fictional_character|Gollum] in th
who played [fictional_character|Gollum] in the
who played [fictional_character|Gollum] in [film|The Lord of the Rings: The Fellowship of the Ring]
who played [fictional_character|Gollum] in [film|The Lord of the Rings: The Return of the King]
who played [fictional_character|Gollum] in [film|The Doctor]
who played [fictional_character|Gollum] in [netflix_title|The Beast]

who played [fictional_character|Gollum] in th
who played [fictional_character|Gollum] in the
who played [fictional_character|Gollum] in [film|The Hunger Games (Science Fiction Film)]
who played [fictional_character|Gollum] in [film|The Corporation]
who played [fictional_character|Gollum] in [film|The Queen]
who played [fictional_character|Gollum] in [tv_program|The Today Show]
```

Figure 5.1.: Top: Completion suggestions using Word2vec similarity. **Bottom:** Completion suggestions on the basis of the Freebase Easy prominence score.

We changed several more aspects of our algorithm, all of which had little influence on the MRR and the required user interaction. One of these aspects concerned how the completion suggestions are being filled up with entities and normal words. In the current implementation, the fill-up only depends on the entity score and the normal word count. We tested an approach in which we used predictions of an $(n-i)$-gram model $(\forall i \in 1,..,n-1)$ if the $(n-i+1)$-gram model did not yield enough results to fill the number of requested completion suggestions. This did not significantly improve the MRR or the user interaction. It did, however, result in a considerably longer completion time and was thus not considered further.

We also tested different versions of computing the Word2vec similarity. In the most promising version, we computed the Word2vec similarity between a predicted entity and all entities contained in the question prefix plus all non-stop-words. Neither the MRR, nor the user interaction or number of unidentified entities varied significantly from the results of the complete algorithm. Only the completion time was slightly higher for this version of computing the Word2vec similarity.

6. Conclusion

We introduced a novel approach to query auto-completion in which we use an abstract language model to create completion suggestions for the current or next word, given a question prefix. By making use of a language model, our QAC algorithm is independent of query logs. We tackle the issue of data sparsity by replacing specific entities in our language model with abstract categories. During auto-completion, entities are inserted for these categories on the basis of their Word2vec similarity with previously entered entities or their Freebase Easy prominence score. We evaluated our approach on three different datasets and compared the results of different versions of our algorithm. We thereby illustrated the importance of certain components of our algorithm, such as filling up the completion suggestions with normal words and entities that have not been predicted by the language model. The results show, that the completion suggestions produced by our algorithm are of a high quality, even if the prefix for the next word does only contain one letter.

6.1. Future Work

In our approach, the user has to type the exact entity name or a rotation of the entity name to ensure that the entity is identified correctly. However, the user might not know what the correct name of an entity is or might not be aware of the fact, that a specific entity name is required. The user might therefore enter a synonym or an abbreviation of the actual entity name, as for example *Irish* for *Republic of Ireland* or *SPD* for *Social Democratic Party of Germany.* Integrating a proper entity recognition algorithm into our system, could therefore improve the user experience.

Similarly, making our QAC algorithm more robust against spelling mistakes would most likely contribute to a better user experience. This extension would result in a much higher number of possible completions that have to be processed. A more efficient implementation might be necessary in order to keep the completion process fast.

In our evaluation we showed, that using Word2vec similarity to rate entities, can in some cases lead to results which are inferior to those obtained by only using the Freebase Easy prominences score for this task. Using a combination of both the Word2vec similarity and the Freebase Easy score of an entity could improve the quality of the completion suggestions in these cases. First tests in that direction yielded promising results. A thorough evaluation is needed to analyze whether this approach can improve the quality of the completion suggestions in general.

So far, our system only provides completions for the current or next word. Suggesting whole questions is expected not to be feasible, since questions, in contrast to typical queries, tend to be rather lengthly. Moreover, the first few words usually provide relatively little information about how the question should be completed. Completion suggestions for the entire question would rarely be a match after the first few typed letters and even words. Therefore, the user would have to enter a relatively long input string in order to get a matching completion suggestion and would most likely have to type even more than if he was provided with single-word completions. Additionally, entities which the user is typing early on in the question prefix would not be suggested, unless the completion suggestions already contained the matching completion for the whole question. These entities would therefore not occur in the proper entity format and would not be recognized as entities by our current algorithm. This, in turn, would impair the quality of any further completion suggestions. However, providing completion suggestions with a variable number of words could possibly improve the performance of our system. Analyzing this is left for future research.

Acknowledgment

I would like to thank Prof. Dr. Hannah Bast for enabling me to work on this interesting topic. A big thank-you also, for offering valuable advice and suggestions. It is a pleasure to work at this chair.

A special thanks goes out to Timo and Keshav for proofreading my thesis and offering advice.

Finally, I want to thank my family for continuously supporting me. Your encouragement throughout the course of my studies helped me a lot and means a great deal to me.

A. Lists for Choosing Categories

A.1. List of Preferred Types

Person
Film
Fictional Character
Ethnicity
Religion
Celestial Object
Unit
Event
Animal
Plant
Disease or medical condition
Holiday
Recurring event
Programming Language
Human Language
Religious Text
Media Genre
Sports Team
Political ideology
Film series
Software
Website
Profession
Business Operation
Organisation
Brand
Structure
Building complex
Location
File Format
Consumer Product
Industry

A.2. List of Undesired Types

Freebase Data Task
Thing (m/0cgyc3f)
Type/domain equivalent topicInfluence Node
Namesake
Ranked Item
Issuer
Author
Quotation Subject
Film subject
Radio subject
TV subject
Art Subject
Award Nominee
Award-Nominated Work
Award Winner
Award-Winning Work
Adapted Work
Published Work
Literature Subject

Bibliography

[1] Bar-Yossef, Z., and Kraus, N. Context-sensitive query auto-completion. In *Proceedings of the 20th international conference on World wide web* (2011), ACM, pp. 107–116.

[2] Bast, H., Bäurle, F., Buchhold, B., and Haussmann, E. Easy access to the freebase dataset. In *Proceedings of the 23rd International Conference on World Wide Web* (2014), ACM, pp. 95–98.

[3] Bast, H., and Weber, I. Type less, find more: fast autocompletion search with a succinct index. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval* (2006), ACM, pp. 364–371.

[4] Bast, H., and Weber, I. The completesearch engine: Interactive, efficient, and towards ir & db integration. In *Third Biennial Conference on Innovative Data Systems* (2007), pp. 88–95.

[5] Bhatia, S., Majumdar, D., and Mitra, P. Query suggestions in the absence of query logs. In *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval* (2011), ACM, pp. 795–804.

[6] Bird, S. Nltk: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions* (2006), Association for Computational Linguistics, pp. 69–72.

[7] Gabrilovich, E., Ringgaard, M., and Subramanya, A. Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0). *Note: http://lemurproject.org/clueweb09/FACC1* (2013).

[8] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[9] Řehůřek, R., and Sojka, P. Software framework for topic modelling with large corpora. In *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (2010), Citeseer, pp. 45–50.

[10] Shokouhi, M. Learning to personalize query auto-completion. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval* (2013), ACM, pp. 103–112.

[11] Shokouhi, M., and Radinsky, K. Time-sensitive query auto-completion. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval* (2012), ACM, pp. 601–610.

[12] Whiting, S., and Jose, J. M. Recent and robust query auto-completion. In *Proceedings of the 23rd international conference on World wide web* (2014), ACM, pp. 971–982.