Master's Thesis

Question Auto-Completion using a Typed LSTM Language Model

Natalie Prange

Examiner: Prof. Dr. Hannah Bast
Advisers: Prof. Dr. Hannah Bast, Niklas Schnelle

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair of Algorithms and Data Structures

December 13th, 2019

Examiner

Prof. Dr. Hannah Bast

Second Examiner

Dr. Fang Wei-Kleiner

Advisers

Prof. Dr. Hannah Bast, Niklas Schnelle

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 also hereby declare that my thesis has not been prepared for another examination or assignment, either in its entirety or excerpts thereof.

Place, Date	Signature

Abstract

Query auto-completion (QAC) is the task of predicting a user's query as they type. A QAC system presents completion predictions to the user to ease the process of entering a query. Most approaches to QAC heavily rely on query logs - large collections of queries asked by users in the past. These logs are often used to estimate the probability of a query based on how frequently it occurs in the logs. Major limitations of this approach are the inability to predict queries that have not been asked before as well as the fact that search engines with a small user base or recently deployed search engines usually do not have access to large enough and up-to-date query logs. We therefore explore the possibilities of QAC independent of query logs. For this work, we focus on questions as opposed to queries which are often just fragments of the question a user has in mind. We deploy a typed Long Short Term Memory (LSTM) language model trained on more than 11 million questions. In our language model, we replace concrete entities by abstract types in order to overcome the problem of data sparsity. When predicting a query completion, we insert entities for the types that the language model predicts by using an entity prominence score, co-occurrence between entities and word vector similarity. We evaluate our system over a hand-crafted ground truth of question prefixes and reasonable completions. Additionally, we evaluate our system over three question test sets. We compare several versions of our system. Apart from illustrating the importance of various system components, the results show that using co-occurrence to insert entities is superior to inserting entities purely based on an entity prominence score or word vector similarity.

Zusammenfassung

Query Auto-Completion (QAC) ist die Vorhersage von Suchanfragen eines Nutzers während dieser seine Suchanfrage eingibt. Ein QAC System präsentiert dem Nutzer mögliche Vervollständigungen für seine Suchanfrage, um ihm die Eingabe zu erleichtern. QAC basiert meist auf dem Einsatz von Query Logs. Diese sind umfangreiche Sammlungen von Suchanfragen, die in der Vergangenheit von Nutzern gestellt wurden. Query Logs werden häufig benutzt um die Wahrscheinlichkeit einer bestimmten Suchanfrage auf Basis ihrer Häufigkeit im Query Log zu berechnen. Ein Nachteil dieser Herangehensweise ist, dass Suchanfragen, die in der Vergangenheit nie gestellt wurden, nicht vorhergesagt werden können. Des weiteren haben Suchmaschinen mit einer kleinen Nutzerbasis oder neu veröffentlichte Suchmaschinen in der Regel keinen Zugriff auf ausreichend große und aktuelle Query Logs. Wir untersuchen daher die Möglichkeiten, die ein QAC System bietet, welches unabhängig von Query Logs operiert. Der Fokus dieser Arbeit liegt auf Fragen anstelle von Queries, welche oft nur ein Bruchstück der Frage darstellen, die ein Nutzer im Sinn hat. Dafür benutzen wir ein Long Short Term Memory (LSTM) Sprachmodell, welches wir auf einem Fragendatensatz von über 11 Millionen Fragen trainieren. In diesen Fragen wurden konkrete Entitäten durch abstrakte Typen ersetzt um das bei Sprachmodellierung typische Problem der zu geringen Trainingsdaten zu umgehen. Wird dann eine Vervollständigung einer Suchanfrage vorhergesagt, so werden für abstrakte Typen spezifische Entitäten anhand eines Entitätsprominenz-Scores, Kookkurrenz, sowie Wortvektor-Ähnlichkeit eingefügt. Wir evaluieren unser System anhand einer manuell erstellten Ground Truth bestehend aus Suchanfragenpräfixen und möglichen Vervollständigungen für diese. Des weiteren evaluieren wir unser System mit Hilfe dreier Fragendatensätze. Wir vergleichen verschiedene Versionen unseres QAC Systems. Dabei demonstrieren wir die Relevanz der einzelnen Komponenten des Systems. Insbesondere zeigen wir die Überlegenheit von Kookkurrenz beim Einfügen von Entitäten im Vergleich zum Einfügen von Entitäten auf Basis eines reinen Entitätsprominenz-Scores oder Wortvektor-Ähnlichkeit.

Contents

1. Introduction

Nowadays, practically all major search engines such as Google, Yahoo or DuckDuckGo offer query auto-completion (QAC). QAC is the task of predicting a user's query as they type. While the user enters a query, several possible completions for the query prefix are offered to them. The user can then select a completion that matches the query they had in mind. The goal is to ease the process of entering a query. This is done in at least three different ways: 1) reducing typing effort, 2) preventing spelling errors and 3) assisting in phrasing a query.

Reducing typing effort is becoming especially important as more and more of the total online traffic is coming from mobile users. Typing on mobile devices can be particularly tedious as most readers will agree. Not having to type out the entire query but instead being able to select a matching query completion after typing only a few letters can make online search on mobile devices a much more pleasant experience. Zhang et al. report that in 2014, global user's of Yahoo! Search saved 50% of keystrokes when entering English queries by selecting QAC predictions [1]. Google estimates that in 2018, their auto-completion feature reduced typing by about 25%. Moreover, they place estimates of the total typing time saved at 200 years per day[1].

Spelling errors in a query can negatively impact the quality of search results. However, especially when typing on mobile devices, spelling errors occur quite frequently. By offering completion predictions and thus reducing the amount of characters that users have to type by themselves, the number of spelling errors in a query can be drastically reduced.

Sometimes a user might know what they are looking for but they are unsure about how to properly **phrase their request**. By offering completion predictions, QAC can guide the user into finding a proper way to phrase their query. Some QAC

[1] https://www.blog.google/products/search/how-google-autocomplete-works-search/

systems even generate their completion predictions such that the quality of search results is maximized when entering a suggested query.

In order for the user to benefit from these potential advantages of QAC, a QAC system must present a matching prediction after a minimal amount of keystrokes. Moreover, the system must present predictions in real time. A system that takes a second after each keystroke to update its predictions will not be of much help to a user. The completion predictions must also be properly ranked in order to be beneficial to the user. If the user has to scroll through a long list of predictions to find one that matches their intended query, neither time nor effort will be saved by using QAC. When several predictions are presented to the user, the goal is therefore to have the prediction that matches the user's query ranked at the highest position.

Following this intuitive explanation of the goals and tasks of QAC systems, Section 1.1 provides a more formal definition of the problem. The motivation for this work is explained in Section 1.2. An overview of our approach is given in Section 1.3.

1.1. Formal Problem Definition

Let q be a query prefix, i.e. a sequence of characters typed by a user. Let $Q(q)$ denote the set of queries that extend q. Once the set $Q(q)$ is established, a QAC system determines a ranking over the elements in $Q(q)$ and presents the top k suggestions to the user. The goal is to have the user's intended query at the highest possible rank. What exactly is included in the set $Q(q)$ depends on the QAC system. Some systems require that q is an exact prefix of the query extensions in $Q(q)$. Other systems tolerate minor mismatches such that e.g. "*Barack Obama*" $\in Q(q)$ for $q =$"*Barak Ob*". While our system does not tolerate such spelling errors, it does include aliases in $Q(q)$ such that e.g. "*Barack Obama*" $\in Q(q)$ for $q =$"*Oba*". Some systems include complete queries which extend q by several words in $Q(q)$ such that "*who was the first president of the united states*" $\in Q(q)$ for $q =$ "*who was the fi*". Our system aims not at predicting the complete query but at predicting the next word or entity in a query. An entity here refers very generally to something that exists - even if only in fiction -, such as "*Barack Obama*", "*Mount Everest*" or "*quidditch*".

Figure 1.: Google Trends for *"who is elon musk"*: Popularity of the query "*who is elon musk*" from January 2006 to October 2019 as reported by Google Trends. The graph represents search interest over time relative to the highest point on the chart. Data source: Google Trends (https://www.google.com/trends)

1.2. Motivation

Most QAC approaches use large query logs to predict user queries. These are datasets which contain anonymized queries that users have asked in the past. For a given query prefix, the QAC system can then predict those queries from the query log that occur most frequently in the log. This approach in combination with user specific data such as their previous queries, time and location has been shown to achieve good results if extensive query logs exist. However, recently created search engines or search engines with a small user base might not have access to sufficiently large query logs that allow the prediction of future queries. The few large query logs that are publicly available, such as the AOL Query Logs with ca. 36M queries from 2006 [2] or the MSN Query Logs from 2006 and 2007, have long been outdated. This of course negatively affects predictions made from these logs. In 2019, one of the top Google auto-completion predictions for the query prefix "*who is el*" is "*who is elon musk*". The popularity of this query, however, is now much higher than it was in 2006 as shown in Figure 1. As this example illustrates, the probability of a user asking a particular query is subject to drastic changes over time. A QAC system relying on outdated query logs must therefore fail to rank completion predictions according to their current likelihood. Another shortcoming of QAC systems that purely rely on query logs is that queries which have not been asked before can not be predicted. Google claims that in 2017, 15% of all its queries had never been seen before[2] which shows that this is not a negligible issue. These shortcomings make

[2]https://blog.google/products/search/our-latest-quality-improvements-search/

QAC without query logs an attractive research topic with the potential to increase QAC performance for various use cases.

1.3. Our Approach

Typical QAC aims to predict completions for any kind of query a user might enter. This could be a grammatically correct question like "*Who is the husband of Angela Merkel?*" or just a set of keywords such as "*Angela Merkel husband*". Our approach focuses on grammatically correct questions and more precisely factoid questions. It could therefore be profitably deployed as QAC for question answering systems. Another aspect that makes our system particularly useful for question answering systems is that named entities in the user's question are already marked as such and linked against a knowledge base when the user selects a completion prediction. This can lead to more reliable entity recognition and shifts a portion of work from the question answering system to our QAC system.

We use a language model to make predictions about a user's question given the prefix they have typed so far. A language model computes a probability distribution over sequences of words. Given sufficient training data, a language model can for example learn that there is a high probability for the word "*how*" to be followed by "*much*" or "*many*" and a lower probability for it to be followed by "*popular*". Given a sequence of words, a language model can be used to obtain a probability distribution over the words that are likely to follow. In this work, we use a Long Short Term Memory network (LSTM) [3] as language model. LSTM networks are explained in more detail in Section 3.1.

A problem commonly encountered when working with language models is data sparsity. This is the problem of receiving an input that did not occur in the training data. For such inputs it can be hard to make reasonable predictions about the next word. We address this issue by using a typed language model. In our typed language model, concrete entities are replaced by abstract types. That is, the question "*Who played Agent Smith in The Matrix?*" becomes "*Who played [fictional character] in [film]?*". While the concrete entities "*Agent Smith*" and "*The Matrix*" might not have occurred in this context in the training data, some fictional character and film probably have. In general, the number of possible inputs in a typed language model is drastically reduced compared to a classical language model. Data sparsity therefore becomes a

much less pressing issue. The challenge here is to replace each entity by a type that is neither too general nor too specific. If it is too general - assume for example all entities were to be replaced by the type *entity* - inserting concrete entities for the abstract type would become unnecessarily difficult as important contextual information would be lost. On the other hand, if the types are too specific, data sparsity will still be a problem. We use the hierarchical class structure of the open knowledge base Wikidata [4] to map entities to types that are neither too general nor too specific.

When the language model predicts a type to follow a user's question prefix, the entities that were mapped to the predicted type need to be ranked such that the best matching entities can be inserted for the type. For this task, we use a combination of a prominence score, co-occurrence and word vectors. If no entity is present in the question prefix so far, all entities that were mapped to the predicted type are ranked according to an entity prominence score. If an entity is present in the user's question prefix, a ranking is determined using co-occurrence between the present entity and the entities that were mapped to the predicted type. If not enough completion predictions can be produced using co-occurrence, word vector similarity between entities is used to rank possible entities.

We compare various versions of our system to determine the importance of its components. In particular, we compare different entity insertion methods. We use two evaluation methods to assess the quality of our system's completion predictions. Firstly, we evaluate our system over a hand-crafted ground truth dataset that contains question prefixes along with several reasonable completion predictions for each question prefix. Secondly, we evaluate our system over three question datasets to measure the effort a user has to make to enter a specific question. Our results show that co-occurrence is superior as entity insertion method compared to a method that is purely based on an entity prominence score as well as to a method that combines an entity prominence score with word vector similarity.

2. Related Work

2.1. Query Auto-Completion with Query Logs

The most common approach to QAC is to use query logs to estimate a probability distribution over previously observed queries. Completion predictions for a user's query prefix are then made based on the estimated probability distribution. This method is often referred to as MostPopularCompletion (MPC) [5]. MPC is often enhanced by adding contextual or temporal information.

Bar-Yossef et al. add contextual information to MPC to be able to generate better predictions for very short query prefixes [5]. Their NearestCompletion algorithm predicts those queries from a query log that are the most similar to the user's recent queries (the context) and match the already typed query prefix. They represent both, the context and each query from the query log as a vector in the same high dimensional space. In order to get a rich representation of a query, they expand the query using a query recommendation system. NearestCompletion then predicts the query that has the highest cosine similarity to the current context. By combining NearestCompletion with MostPopularCompletion they achieve good results for scenarios in which the context is relevant to the current query as well as those where it is not.

Shokouhi et al. [6] propose a time-sensitive version of MPC. Their approach aims at capturing trends and seasonality in user queries. They improve results that were achieved using the MPC algorithm by forecasting the expected popularity of a query using time-series modeling.

2.2. Query Auto-Completion without Query Logs

Bast and Weber develop a search engine that features QAC for the last query word [7]. The completion suggestions are generated from the results that their search

engine yields for the current incomplete query. Their goal is to generate completion suggestions that lead to good search results rather than to predict the user's query as is the goal of our work. Bhatia et al. extract frequent n-grams from documents of the corpus that is being searched to generate completion suggestions [8].They, too, focus on suggesting completions that lead to good search results instead of predicting the user's query.

Park and Chiba [9] apply a neural language model to improve QAC results for cases where a user's query prefix has not been seen before and the common MPC approach must therefore fail. They use an LSTM network to generate character-level predictions. One of their main reasons for taking the character-level approach is to deal with Out-Of-Vocabulary (OOV) words. We tackle this issue by using a typed language model where we can insert concrete entities even if they did not occur in the training corpus. Fiorini and Lu follow a similar idea and enhance a neural language model QAC system with personalization and temporal information [10].

In our previous work from 2016 [11], we introduce a QAC system similar to the system proposed in this work. It is based on a typed n-gram language model as opposed to the typed LSTM language model used in this work. The n-gram language model is trained on a much smaller question dataset (ca. 1,4 million questions) compared to the LSTM language model we propose here (ca. 11,3 million questions). Freebase [12] entities and types are used instead of Wikidata entities and classes. For the insertion of entities, a combination of the Freebase Easy prominence score [13] and word vector similarity is used as opposed to a combination of Wikidata sitelinks counts, word vector similarity and co-occurrence. The results achieved with the QAC system proposed in this work show great improvements over the results achieved with the previous version of the system.

2.3. Class-Based Language Modeling

Most of the earlier research on class-based language modeling has been focusing on n-gram language models ([14], [15], [16]) and assigning a class to each word ([14], [15], [16], [17]) as opposed to assigning classes only to named entities.

Goodman introduces a class-based language model to speed up training of maximum entropy models [17]. He creates two separate models, where the first model predicts the class of a word given its context and the second model predicts the word given

its class and context. His goal is different from ours in that we want to tackle the data-sparsity problem whereas Goodman aims at speeding up the training of the language model.

Parvez et al. address the issue of generating a language model for texts with named entities [18]. They propose a typed language model as a combination of two LSTM language models: a type model and an entity composite model. Similarly to our typed language model, the type model disregards concrete entities and learns a probability distribution over normal words and entity types. While we apply an entity prominence score, co-occurrence and word vector similarity to insert concrete entities, Parvez et al. use a second language model, the entity composite model, which provides probabilities for concrete entities given the type predicted by the type model. Parvez et al. demonstrate the effectiveness of their approach for two specific tasks: recipe generation and code generation. By making use of the general knowledge base Wikidata and giving insights on how any kind of entity can be mapped to an abstract type, we hope to provide a more general and holistic approach. Moreover, co-occurrence counts and entity prominence scores can be retrieved with little effort and even a Word2Vec model can be trained more easily than an entire second language model.

To the best of our knowledge we perform the first research on deploying a typed language model for QAC.

3. Theoretical Background

In order to obtain a good understanding of the functionality of our system there are a few non-straightforward concepts that should be understood first. We use a Long Short Term Memory (LSTM) network to train our typed language model. In Section 3.1 we give a short introduction to neural networks in general and LSTMs in particular. We apply three different methods to insert entities for abstract types. While the entity prominence score and word co-occurrence are mostly self-explanatory, word vectors require a more detailed explanation which we provide in Section 3.2. We use prefix trees for the efficient retrieval of entities and words that start with a given prefix. Section 3.3 gives an intuition for the basic concept and the complexity of this data structure.

3.1. Long Short Term Memory Networks

To provide the reader with an intuition about how a Long Short Term Memory (LSTM) network can be used to model language, we will first explain how neural networks work in general. We will then describe what recurrent neural networks (RNNs) are and how an LSTM network can be used to overcome shortcomings of RNNs. Finally, we will explain how an LSTM network can be applied as language model.

3.1.1. Artificial Neural Networks

As the name suggests, artificial neural networks are nets of interconnected neurons inspired by the neural networks of animal brains that can learn patterns from large amounts of data. Nowadays, artificial neural networks are used to solve manifold tasks that range from image recognition over medical diagnosis to language modeling. Neurons constitute the basic building blocks of a neural network and are arranged in

layers. Each neuron takes inputs $x_1, x_2, ...x_n$ where each input is multiplied with a weight $w_1, w_2, ..., w_n$. The neuron adds up the weighted inputs together with a bias b, passes the sum through an activation function f and outputs the result

$$y = f((w_1 \cdot x_1) + (w_2 \cdot x_2) + ...(w_n \cdot x_n) + b)$$

A common activation function is the sigmoid function

$$f(x) = \frac{1}{1 + e^{-x}}$$

which conveniently turns the result into a value between 0 and 1. In a multilayer network, the output of a neuron is in turn one of the inputs of the next layer of neurons. A neural network consists of an input layer, one or several hidden layers and an output layer. Consider for example the task of classifying handwritten digits in images as an introductory example. The classical dataset for this task is the MNIST dataset that contains 28 x 28 pixel greyscale images of single handwritten digits. In this case, the input layer could consist of $28 \cdot 28 = 784$ input neurons where each one encodes the greyscale value of a single pixel. The hidden layers process the input. By training their weights, they can be used to compute features that are helpful in classifying the given image. The output layer could consist of 10 neurons, one for each digit from 0 to 9, which encode the probability that the current image depicts the corresponding digit. The image is then classified as the digit that corresponds to the output neuron with the maximum probability.

For the training, a neural network is fed with large amounts of data referred to as training data. In the digit-classification example that would be images of digits. The network makes a prediction for each training sample according to its internal weights. A loss function is then used to quantify how well the network is performing in its current state. A frequently used loss function is the mean squared error defined as

$$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_{true} - y_{pred})$$

where n is the number of training samples, y_{true} is the true value for the given input and y_{pred} is the value predicted by the network. The goal when training the network is to minimize the loss function by adjusting the weights and biases in the network.

For this, an optimization algorithm is used, for example stochastic gradient descent (SGD). Gradient descent attempts to find the minimum of a convex function by iteratively stepping down the function in the direction of its negative gradient at the current point. In a neural network, backpropagation is used to calculate the gradients. The details are not necessary to understand the work presented in this thesis but the basic idea is to propagate the error defined by the loss function back through the network by calculating partial derivatives. This identifies the contribution of each weight to the overall error and the weights can be adjusted to minimize the error. Stochastic gradient descent uses random samples or batches of training data to calculate the gradients instead of using all available data since classical gradient descent becomes very slow on large datasets.

In summary, one cycle (often referred to as *step*) of forward and backward passing a batch of data consists of

1. Computing the output of the network by a forward pass of the input values through the network

2. Measuring the current error using a loss function

3. Computing the gradients for minimizing the error and adjusting the network weights accordingly

The processing of all data in the dataset once, e.g. processing $\frac{dataset\_size}{batch\_size}$ batches, is called an *epoch*.

The neural networks described above are also called feedforward neural networks. The input for the network is only the current data sample, e.g. the current image that is supposed to be classified.

3.1.2. Recurrent Neural Networks

While feedforward neural networks only allow fixed-sized vectors as input and only output fixed-sized vectors, recurrent neural networks (RNNs) allow series of vectors both as input and as output. They operate not just over the current data sample but also consider data from previous time steps in the series. Input data from previous time steps therefore influences the output at the current time step. Information from previous time steps is stored in the so called *hidden state*. Both, the current hidden

state and the weights of the network determine its output at any given time. The same current input could therefore result in a different output depending on previous inputs in the series. Unlike in feedforward neural networks, weights are shared in an RNN over all time steps of the same series. When generating the output for a given time step, the hidden state is updated such that the current input will influence the output in the next time step.

Taking previous inputs into account is unnecessary and probably even detrimental for some tasks such as for example digit classification: observing a certain digit at time step $t-1$ should not play a role in classifying a digit that is observed at time step t. However, for tasks where data samples come in series or have long-term dependencies, this property is essential for successful learning. RNNs can for example be trained on a large set of text and then be used to produce words one character at a time. An RNN can learn that the character "*t*" has a high probability to be followed by "*h*" and that the character sequence "*th*" is likely to be followed by a vowel like "*e*" or "*a*". It can learn that the character sequence "*the*" is probably followed by a space or an end-of-word token and so on. A feedforward neural network that has no memory of the data observed in a previous time step would necessarily fail at this task.

A problem that arises when dealing with long-term dependencies in RNNs is the problem of vanishing gradients. Due to the recurrence of weights, weights < 1 can shrink the gradient exponentially with the length of the time series during backpropagation. Thus, a vanilla RNN struggles to relate information from faraway time steps to current inputs.

3.1.3. LSTM Networks for Language Modeling

Long Short-Term Memory networks were proposed by Hochreiter and Schmidhuber as a solution to the vanishing gradient problem [3]. They are especially designed to deal with long-term dependencies in series of data. LSTMs store information about previous time steps in gated cells. Ruled by their own set of weights, these cells decide when to allow read, write and delete operations over their content.

LSTM networks outperform vanilla RNNs in many tasks and are commonly used for language modeling as is the case in this work. We want to build a language model that, given a question prefix, makes predictions about which words are most likely to follow the words in the prefix. In order for the LSTM network to be able to work

with words as input, we need to represent words as vectors. The solution to this are word embeddings, or word vectors, which are explained in more detail in Section 3.2. Essentially, word embeddings capture semantic and syntactic properties of the words they represent and can be learned separately or during the training of the LSTM network. An embedding layer can be integrated into the network to encode input words as word vectors. In a typical LSTM network for language modeling, this input embedding layer is followed by two stacked hidden LSTM layers. The output layer is typically a softmax layer of the size of the vocabulary. A softmax layer takes the raw output values of a neuron layer which are in the range $(-\infty, \infty)$ and converts them into probabilities summing up to 1. This way, the output of a LSTM network can be formed into a probability distribution over all words in the training vocabulary.

3.2. Word Embeddings

In order to insert concrete entities for abstract types when an entity is already present in the question prefix, we need a measure of semantic similarity between the present entity and entities that could possibly be inserted for the abstract type. Word embeddings can be applied to obtain such a measure.

Word embeddings are vector representations of words which ideally capture semantic or syntactic similarities between the words they represent. Different methods exist to generate these vector representations such as using neural networks [19] or dimensionality reduction on co-occurrence matrices via Latent Semantic Analysis [20]. In our work, we use Google's Word2Vec continuous bag-of-words (CBOW) method [19] to create word embeddings. Word2Vec was published in 2013 by Mikolov et al. and has since received a great amount of attention in the natural language processing community. Word2Vec uses a neural network which consists of an input, a projection and an output layer to learn vector representations of words. The architecture ignores word order and takes into account both the history and the future context of a word. The resulting word vectors are - when trained on a large enough corpus - highly capable of capturing semantic similarities. The power of Word2Vec models is popularly demonstrated in word association tasks of the form "*man is to king as woman is to x*". Using a Word2Vec model, this task can be solved by simply computing

$$vector(\text{“}King\text{“}) - vector(\text{“}Man\text{”}) + vector(\text{“}Woman\text{”})$$

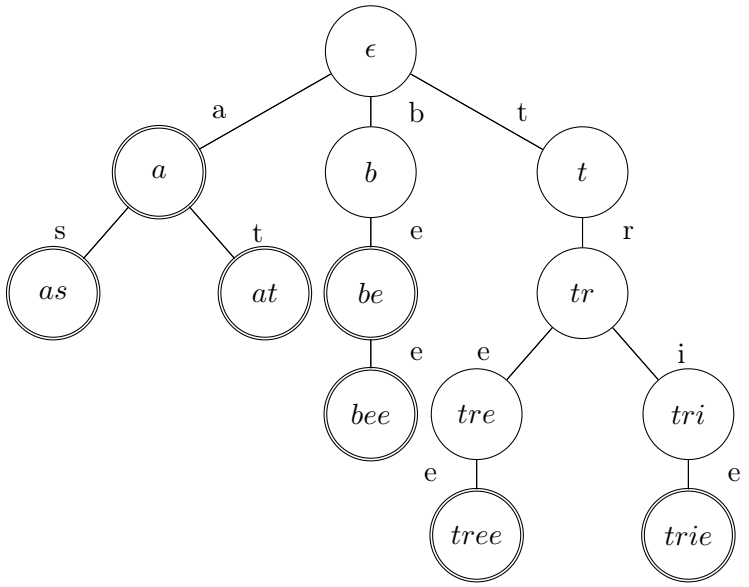
The result is a vector that is closest to the vector of the word *“Queen”*. In general, it holds that the more common context two words share, the higher the cosine similarity of their vectors. Therefore, the cosine similarity between “*The Matrix*” and “*Neo*” will be higher than the cosine similarity between “*The Matrix*” and “*Gandalf*”. We use this property of word vectors e.g. to insert entities for types when another entity already exists in the question prefix. To train the Word2Vec model and integrate it into our system we use the Python library Gensim [21]. The use cases of the trained Word2Vec model in our system are described in Section 4.4.2 and Section 4.5.1.

3.3. Prefix Trees

Performance is crucial to any QAC system. A QAC system that cannot provide completion predictions within a fraction of a second fails to fulfill one of its main objectives which is reducing time and effort spent on inputting a query. In our system, once the user has entered a question prefix, the language model should predict only those words from its vocabulary of more than 100,000 words that match the current word prefix, i.e. the word the user is currently typing, along with any types. Similarly, when the language model predicts a type, only those entities which are assigned to the predicted type should be inserted which match the current word prefix. However, the type “*human settlement*” alone is assigned to over 100,000 entities in our system and typically, the language model predicts several types. To ensure our system's responsiveness we need a data structure that allows us to efficiently retrieve all those items that start with a given prefix. As in the previous version of our system, we use prefix trees for this task.

A prefix tree, also referred to as trie, is a tree data structure where each edge is assigned a character and each node is assigned the concatenated list of characters that is obtained when following the edges from the root to the node. Figure 2 is an example prefix tree built from the words *a*, *as*, *at*, *be*, *bee*, *tree* and *trie*. These words are also referred to as the *keys* of the prefix tree. Note that a key node can not only correspond with a leaf node but also with an inner node. Each path in the tree corresponds to a prefix of a key. All keys that share a common prefix also share a common path in the tree. Typically, each key is associated with a value that does

Figure 2.: Prefix tree: A prefix tree built from the words *a*, *as*, *at*, *be*, *bee*, *tree* and *trie*. The keys, i.e. words that were used to built the prefix tree, are marked by double circles.



not need to be unique and can be of any type. If each node can access a list of all keys in its child nodes, all keys starting with a given prefix can be retrieved in $O(m)$ where m is the maximum length of any key in the prefix tree.

In our system, we use prefix trees for several purposes. We maintain a prefix tree that stores the vocabulary of our language model. That way, we can filter the language model predictions in regards to whether they match the current word prefix. Types are not filtered out since in this case, the filtering is performed once concrete entities are inserted. For each entity type, we maintain a prefix tree that stores the labels of all entities that were assigned to this type. This way, when the language model predicts a type, entities of this type that match the given prefix can be retrieved efficiently. We use the Python library datrie [1] to implement prefix trees in our QAC system.

[1] https://pypi.org/project/datrie/

4. Our Approach

This chapter provides a detailed description of our QAC system. An overview of the system is depicted in Figure 3. The first step is building the typed language model (Section 4.1). For this, concrete entities need to be mapped to abstract types (Section 4.1.1). The language model is then trained over a dataset containing over 11 million typed questions (Section 4.1.2) using an LSTM network (Section 4.1.3). In our QAC system, the language model is used to predict words and types for a question prefix entered by a user (Section 4.2). If types are predicted, the system inserts entities using a prominence score and co-occurrence (Section 4.3). The completion predictions obtained after these steps are ranked according to a final score (Section 4.4). In a post-processing step, entities that occur multiple times in the suggestions are removed. Other entities, such as entities whose label has been completely typed by the user, are appended to the completion predictions that are presented to the user (Section 4.5).

4.1. Building the Typed Language Model

4.1.1. Mapping Entities to Types

In order to build the typed language model, concrete entities need to be mapped to an abstract type. We work with Wikidata to obtain such a mapping. Wikidata is an open knowledge base that contains structured data in the form of items and connections between these items. We will refer to these items as entities. Each entity has a unique identifier (QID) consisting of the initial letter "*Q*" followed by a unique number. An entity typically has a label and a description. Some entities are assigned aliases. For example, the entity that represents the fictional character Gandalf has the QID *Q177499*, the label "*Gandalf*", the description "*fictional character created by J. R. R. Tolkien*" and among others the aliases "*Gandalf the Grey*", "*Olórin*" and "*Mithrandir*". Unless otherwise indicated, we denote entities as "⌊<*QID*>:<*label*>⌋". In our QAC system, the assigned type of an entity is added to this notation as

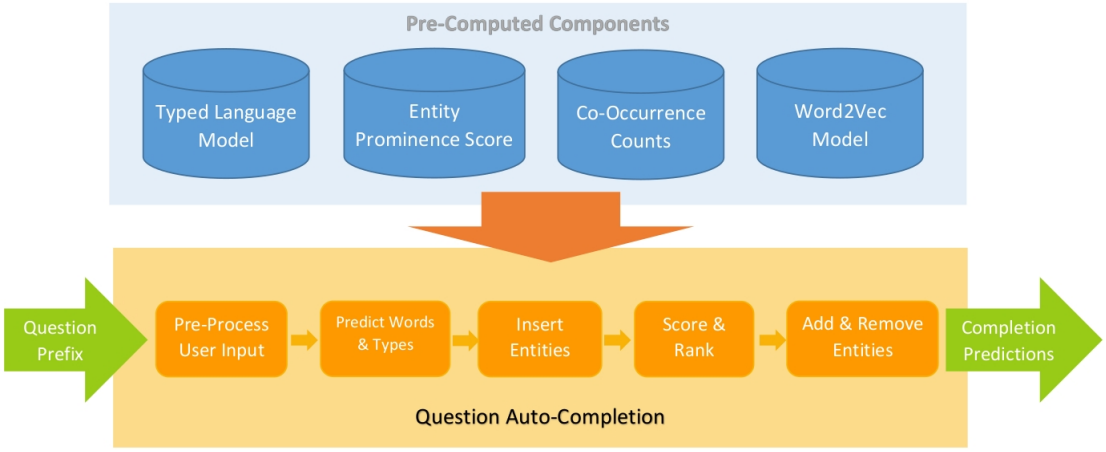
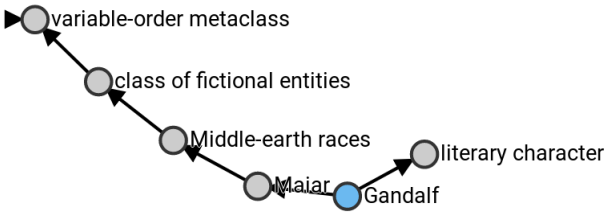
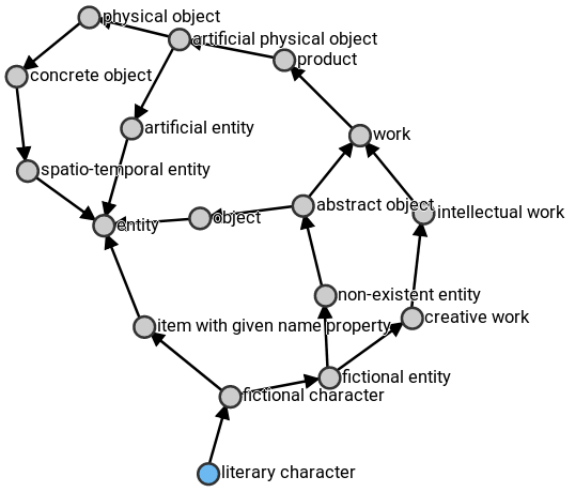


Figure 3.: The basic pipeline of our system: The figure gives an overview of the most crucial components of our system and the steps that are necessary to produce completion predictions for a question prefix entered by the user.

"*[<type>/<QID>:<label>]*" to make it easier for the user to discriminate between entities with the same label. Wikidata entities are connected via properties. The entity "*[Q177499:Gandalf]*" for example is connected via the "*sex-or-gender*" property to the entity "*[Q6581097:male]*" and via the property "*performer*" to the entity "*[Q170510:Ian McKellen]*". Entities are assigned classes via the "*instance-of*" property. Classes, which are Wikidata entities themselves, are hierarchically structured via the "*subclass-of*" property. An entity can be the instance of several classes and a class can be a subclass of several classes. Classes themselves can also be an instance of another class. Figure 4 gives an intuition for the Wikidata class hierarchy. We define the set of classes of an entity as all its direct classes, i.e. classes connected to the entity via the "*instance-of*" or "*subclass-of*" property and all indirect classes, i.e. classes connected to classes of the entity via the "*subclass-of*" property. As can be seen in Figure 4b, the root class of almost all entities is the class "*entity*". The challenge is to choose a type out of these classes such that it is neither too general, nor too specific. If all entities in the typed language model are replaced by the type "*entity*", it will be difficult to insert a matching entity later using co-occurrence or even only an entity prominence score. If on the other hand "*[Q177499:Gandalf]*" was to be replaced by "*Maiar*", a class that at the time of writing is assigned to only 17 Wikidata entities, the language model probability of this type after the prefix "*Who played*" would be extremely low compared to types such as "*human*". This is despite



(a) Wikidata graph for the "*instance-of*" property: Starting from the entity *[Q177499:Gandalf]*, the graph shows connections between Wikidata items via the "*instance-of*" property.



(b) Wikidata graph for the "*subclass-of*" property: Starting from the entity "*[Q3658341:literary character]*", the graph shows connections between Wikidata items via the "*subclass-of*" property.

Figure 4.: Wikidata class hierarchy. Source: Wikidata Graph Builder (https://angryloki.github.io/wikidata-graph-builder)

the fact that "*[Q177499:Gandalf]*" is just as likely to appear in this context as is "*[Q1001:Mahatma Gandhi]*" if not more so. This goes to show that the quality of the completion predictions of our QAC system greatly depend on the choice of type for each entity. We exploit the hierarchical structure of Wikidata classes to select types that are neither too general nor too specific.

In our previous work, we assigned exactly one type to each entity. For this work, we experimented with using a single type per entity, two types and a varying number of types that could differ from entity to entity. The best results were achieved assigning two types per entity: one more specific type, in the following referred to as primary type, and one more general type, referred to as secondary type. In this mapping, "*[Q177499:Gandalf]*" for example is assigned the primary type "*fictional character*" and the secondary type "*creative work*".

For the entity-to-type mapping, we create two sorted lists of preferred types - one for the primary types and one for the secondary types. If one of the preferred types from that list occurs in the classes of the entity, the entity is mapped to that type. If several of the preferred types occur in the set of entity classes, the type that appears first in the preferred type list is chosen. In addition, there is a small set of rules that apply for certain types, in the following referred to as *leads-to* rules. These rules define classes that should be chosen as type if a certain other class exists in the set of classes of an entity. E.g if the class "*software*" is present in the set of classes, then the class "*product*" should be chosen as secondary type. Without this extra rule, a software would be assigned the type "*creative work*". Both "*creative work*" and "*product*" are a super class of "*software*", but "*creative work*" is ranked higher in the preferred type list for secondary types. If one was to rank "*product*" higher than "*creative work*", things like the Lord of the Rings book series are assigned the type "*product*" instead of "*creative work*". Therefore the *leads-to* rules are necessary to ensure a proper type assignment. The preferred type lists for both the primary and secondary types can be found in the appendix, as well as the *leads-to* rules which only exist for secondary types.

If no class of an entity's class set appears in the preferred type list for primary types and no *leads-to* rule applies, then the primary type becomes the same as the secondary type. If no secondary type could be assigned either, then one of the classes that are directly linked to the entity via the "*instance-of*" or "*subclass-of*" property becomes the primary type. If no secondary type was found using the process described above, the primary type is assigned as secondary type as well. Few Wikidata entities do

neither have an "*instance-of*" nor a "*subclass-of*" property. These entities are assigned the type "*unknown*" as both primary and secondary type.

4.1.2. The Question Dataset

We use a dataset of 11,290,367 questions as training data for the typed language model. Most of them (ca. 97%) stem from a variation of the WikiQuestions dataset [22]. The WikiQuestions dataset was created in a previous work using a Wikipedia dump from 2015 with recognized Freebase entities. The questions were generated as follows: First, a dependency parse of the dump was created. Then, for each sentence, an entity in the sentence was selected as answer entity. This answer entity was replaced by a question word that matched the type and function of the entity within the sentence. Question words were restricted to "*who*", "*what*", "*when*", "*where*" and "*which <type>*". Various transformations were performed over the sentence, such as subject auxiliary inversion and sub clause removal, in order to rephrase it as a question. Filters were applied to avoid ungrammatical or otherwise unreasonable questions. Two filters that were applied in the original WikiQuestions dataset were skipped for this work to ensure a sufficiently large amount of training questions. The first one filtered out questions that contained no entity. The second one filtered out questions that contained recognized entities where the original word consisted only of lowercase letters. This sometimes meant that a normal word was falsely identified as an entity. However, this filter resulted in some types of entities, which are always written in lowercase, not being properly represented in the dataset. We augment the modified WikiQuestions dataset with questions from the ClueWeb12 Facc1 corpus [23]. The ClueWeb12 corpus contains text from 456,498,584 English web pages with Freebase entity mentions. We use questions that start with one of the question words "*who*", "*what*", "*when*" and "*where*". We then apply the same filters over the questions as are applied to the WikiQuestions dataset. The WikiQuestions and ClueWeb12 questions are combined and shuffled. Freebase entities are mapped to Wikidata entities. For this mapping, we use the Wikidata property "*Freebase ID*", as well as a mapping created by Samsung[1]. Freebase entities that do not have a mapping to any Wikidata entity are replaced by the token "*[unknown]*". The resulting dataset consists of 11,390,367 questions. We hold back 100,000 questions for our test sets which are described in detail in Section 5.1. The remaining questions are used as training set. Entities in the training dataset are replaced by their combined primary

[1] https://github.com/Samsung/KnowledgeSharingPlatform

and secondary type, e.g. "*[Q177499:Gandalf]*" is replaced by "*[Q95074:fictional character/Q17537576:creative work]*". The prefix QID is necessary as class names in Wikidata are not unique. The result is a training dataset consisting of 11,290,367 typed questions.

4.1.3. Training the LSTM Network

The training questions are fed into an LSTM network in order to train the typed language model. A general introduction to LSTM networks is given in Section 3.1. For the design of our LSTM network, we follow a network architecture that is commonly used for language modeling. The input layer of our LSTM network consists of a word embedding layer with an embedding size of 100. The embedding vectors are fed into two stacked LSTM layers of size 512. A dropout [24] of 0.3 is applied to each of the LSTM layers. The output layer is a softmax layer of the size of the training vocabulary such that the network outputs a probability distribution over the vocabulary. We train our LSTM network for 15 epochs with a batch size of 512 over the training dataset of 11,290,367 typed questions. The beginning of each question is padded with the beginning-of-sentence token "*_BOS_*" and the end of each question is padded with the end-of-sentence token "*_EOS_*". Words that occur less than 3 times in the training data are replaced by the out-of-vocabulary token "*_UNK_*". Training the typed language model took 9 days and 21 hours on an NVIDIA GeForce Titan X (Pascal) GPU with 3072 CUDA cores, a boost clock rate of 1.531 GHz and a memory bandwidth of ca. 420 GB/s.

4.2. Predicting Words

We aim to provide the following user experience: For each user input, starting with the empty word, we want to yield predictions for the next complete word or entity that matches the current word prefix based on the context of the previous words. In a conventional language model, the distinction between the current word prefix and the context words is straight forward. Each character typed after the last whitespace is part of the current word prefix i.e. part of the word that has to be predicted. Each word before the last whitespace is a context word. If the user input does not contain any whitespace yet, each character is part of the current word prefix and the list of context words is empty. For the user input "*Who directed the lord of th*", a

conventional language model would predict completions for the current word prefix "*th*" given the list of context words (*who*, *directed*, *the*, *lord*, *of*). When dealing with a typed language model, this distinction is more complicated. In our typed language model, any number of words entered by the user can be part of the current word prefix since an entity label can consist of several words. In the example input, the most likely current word prefix is "*the lord of th*". The only restriction we make for the current word prefix in our system is that a marked entity in the user input can not be part of the current word prefix. That is, entities cannot be nested. Therefore, if there already exists an entity in the user input, the longest possible current word prefix starts after the whitespace following the last entity. Otherwise, the longest possible current word prefix starts from the first letter of the user input. Our system computes completion predictions for all possible current word prefixes.

The input for the LSTM language model is the sanitized list of context words. That is, the words the user has typed so far excluding the current word prefix. The sanitation process consists of converting the input words to lowercase and filtering out special characters like commas or quotation marks. Words that are not in the vocabulary of the LSTM language model are replaced by the out-of-vocabulary token "*_UNK_*". The list is padded to the left with the beginning-of-sentence token "*_BOS_*". Entities are replaced by their combined primary and secondary type as they appear in the training dataset for the LSTM language model. The result of the LSTM language model prediction is a list of words and their respective probabilities together with the index of the current word prefix they are supposed to complete.

4.3. Inserting Entities

In the next step, entities are inserted for each type that is predicted by the language model and assigned a score. This insertion score is later combined with the language model probability and certain penalty factors to a final score that determines the ranking of the completion predictions generated by the system.

First, for each predicted type, we retrieve all those entities that match the current word prefix and that have the same type (either primary or secondary) as the primary type of the prediction. In doing so, we only consider Wikidata entities with a Wikibase sitelink count ≥ 15 in order to reduce the time and space consumption of our system. Wikibase sitelinks connect Wikidata entities to pages on other Wikimedia sites like

Wikipedia or Wikibooks. The sitelink count of an entity can therefore be used as a measure of prominence. We refer to the resulting set of entities as candidate entities for a predicted type. For efficient retrieval of candidate entities we maintain one prefix tree per type. See Section 3.3 for a brief overview of the prefix tree data structure. Each prefix tree contains as keys the labels of all entities assigned to that type and as values the ids that map the entity labels to unique entities. If several entities of the same type share the same label, we append the entity's id to its label to ensure unique keys in the prefix tree. We also include entity aliases in the prefix tree to be able to insert entities whose label does not match the current word prefix but whose alias does.

It depends on the context words which method is used to compute the insertion score of candidate entities for a predicted type. Those context words that play a role in scoring candidate entities and therefore in inserting entities for predicted types are in the following referred to as insertion context words. We denote the set of insertion context words as $I(C)$ for a set of context words C. The insertion context words consist of entities that are already present in the question prefix as well as the lemmatized version of *<type>* if the question prefix starts with the words "*which <type>*". For example, for the context words $C =$ (*which*, *countries*, *were*, *members*, *of*, *the*, *[Q1065:United Nations]*, *in*) the set of insertion context words would be $I(C) =$ (*country*, *[Q1065:United Nations]*). If the set of insertion context words is empty, i.e. if the set of context words does not contain any entity yet and the question prefix does not start with the words "*which <type>*" then an entity prominence score is used to score candidate entities for the predicted type. This is explained in Section 4.3.1. Otherwise, co-occurrence between the insertion context words and the candidate entities is used as described in Section 4.3.2.

4.3.1. Scoring via Entity Prominence Score

When the set of insertion context words is empty, i.e. $I(C) = \emptyset$, an entity prominence score is used to score candidate entities. The prominence score of an entity is based on its Wikibase sitelink count. Let W be the vocabulary of our QAC system consisting of normal words W_N and entities W_E such that $W = W_N \cup W_E$. Let $W_{E \geq 15}$ be the set of entities with a sitelink count ≥ 15 such that $W_{E \geq 15} \subset W_E$. We assign each entity $w \in W_{E \geq 15}$ a score between 0 and 1 by normalizing the number of its sitelinks using the formula

$$s_{sitelinks}(w) = \frac{c(w) - c_{min}(w)}{c_{max}(w) - c_{min}(w)}$$

where $c(w)$ is the sitelink count of the entity. Instead of taking the minimum and maximum sitelink count over all entities for the normalization, we take the minimum and maximum count over all entities with the same type as the current entity as $c_{min}(w)$ and $c_{max}(w)$. Formally, let $T = (t_1, t_2, ..., t_n)$ be the set of entity types, i.e. $t \in T$ is the combined primary and secondary type of at least one entity $w \in W_E$. Let $t(w)$ denote the type of a word $w \in W$. If the word is an entity, i.e. $w \in W_E$, then $t(w) \in T$. If w is a normal word, i.e. $w \in W_N$, then $t(w) = w$. Using this notation, $c_{min}(w)$ and $c_{max}(w)$ are defined as

$$c_{min}(w) = min(c(w') : w' \in W_{E \geq 15},\, t(w') = t(w))$$

$$c_{max}(w) = max(c(w') : w' \in W_{E \geq 15},\, t(w') = t(w))$$

The intuition behind normalizing separately for each type is that entities of certain types tend to have a much higher sitelink count than entities of other types. The 40 Wikidata entities with the most sitelinks excluding Wikidata meta entities are either countries or continents. However, the prominence score should mainly be used to determine which entity is best inserted for a predicted type. The language model should have the highest influence on which type of entity is predicted as only the language model has the necessary contextual information to make this prediction. The normalization within entities of the same type ensures that the maximum score of an entity of type A is just as high as the maximum score of an entity of type B. Therefore the decision which type of entity is predicted is mainly made by the language model.

4.3.2. Scoring via Co-Occurrence

If the set of insertion context words is not empty, i.e. $I(C) \neq \emptyset$, we use co-occurrence between the insertion context words and the candidate entities to compute the insertion score for the candidate entities. We compute the co-occurrence counts over the Wikipedia dump with entity mentions which we described in Section 4.1.2. Co-occurrence between two words is computed by counting how often the two words occur within the same sentence. A co-occurrence count is computed for two kinds of word pairs: 1) for each pair of entities (w_i, w_j) with $w_i, w_j \in W_{E \geq 15}$ and 2) for each

pair (w_t, w_i) where $w_i \in W_{E \geq 15}$ and w_t is a lemmatized version of a word that appears as *<type>* in the question prefix "*which <type>*" in at least one of the questions from the WikiQuestions dataset. The reason for computing co-occurrence between entities and these *<type>* words is to improve the quality of questions that start with the words "*which <type>*". In such a question, *<type>* can provide valuable information about which entity should be inserted for a type predicted by the language model. E.g. in the question prefix "*Which countries are members of the* ", the language model will predict the type *organization*. The word "*countries*" in the context words provides information about which kind of organization should be inserted. The organization "*Central Intelligence Agency*" should have a lower probability to be inserted than the organization "*United Nations*". Co-occurrence enables us to capture this difference in probability. While the word "*countries*" or a lemmatized version of it occurs 3867 times in the same sentence as the entity "*[Q1065:United Nations]*" in our Wikipedia dump, it occurs only 242 times with the entity "*[Q37230:Central Intelligence Agency]*". This shows how co-occurrence can be used to improve the quality of questions that start with the words "*which <type>*".

We denote the set of *<type>* words as W_T with $W_T \subset W_N$. We denote the lemmatized version of a word or a set of words using the lemmatization function $l(\cdot)$. The co-occurrence count for a pair of words (w_i, w_j) with $w_i \in W_{E \geq 15} \vee l(w_i) \in l(W_T)$ and $w_j \in W_{E \geq 15}$ is denoted as $o(w_i, w_j)$. The counts are normalized to values between 0 and 1 using the formula

$$o_{norm}(w_i, w_j) = \frac{o(w_i, w_j) - o_{min}(w_i)}{o_{max}(w_i) - o_{min}(w_i)}$$

where $o_{min}(w_i)$ and $o_{max}(w_i)$ are the respective minimum and maximum co-occurrence counts for the word w_i with

$$o_{min}(w_i) = min(o(w_i, w') : w' \in W_{E \geq 15})$$

$$o_{max}(w_i) = max(o(w_i, w') : w' \in W_{E \geq 15})$$

Note that while $o(w_i, w_j) = o(w_j, w_i)$, the same does not hold for o_{norm}.

If the set of insertion context words consist of multiple words, i.e. $|I(C)| > 1$, we take the mean of the normalized co-occurrence counts as the candidate entity's co-occurrence score. Thus, given the non-empty set of insertion context words $I(C)$,

the insertion score for a candidate entity $w \in W_{E \geq 15}$ is computed as

$$s_{coocc}(w|I(C)) = \frac{\sum_{w' \in I(C)} o_{norm}(w', w)}{|I(C)|}$$

4.4. Ranking Predictions

The final score of a completion prediction is composed of three components: the language model probability for the predicted word or type (Section 4.4.1), the insertion score for inserted entities (Section 4.4.2) and three different penalty factors (Section 4.4.3). The completion predictions are ranked according to the final score computed from these three components (Section 4.4.4).

4.4.1. Language Model Probability

In order to properly rank completion predictions for different current word prefixes as described in Section 4.2, it is not enough to consider the probability for each prediction given its context words. Instead, we also need to incorporate the probability to observe the context words into the probability for the completion prediction. We refer to the probability of observing a certain set of context words as context probability. In order to compute the context probability with our typed language model, entities in the context words need to be mapped to their type. For better readability, we use $T(C) = (w_1^t, w_2^t, ..., w_i^t)$ to denote the set of typed context words instead of $T(C) = (t(w_1), t(w_2), ..., t(w_i))$. Note that $t(w) = w$ for $w \in W_N$. The context probability $p_c(T(C))$ for the set of typed context words $T(C)$ is then computed as

$$p_c(T(C)) = p(w_i^t|w_{BOS}, w_1^t, w_2^t, ..., w_{i-1}^t) \cdot p(w_{i-1}^t|w_{BOS}, w_1^t, w_2^t, ..., w_{i-2}^t) \cdot ... \cdot p(w_1^t|w_{BOS})$$

where $p(w_i^t|w_{BOS}, w_1^t, w_2^t, ..., w_{i-1}^t)$ is the probability estimated by the LSTM language model to observe the word w_i^t given the words $w_{BOS}, w_1^t, w_2^t, ..., w_{i-1}^t$. w_{BOS} denotes the beginning-of-sentence token. Using this formula, shorter contexts - and thus longer current word prefixes - are assigned a higher probability than longer contexts (unless a word has a probability of 1 given its context words, which practically never happens). While this is desired, we do not want the effect to be quite as strong. To

reduce the effect, we compute the discounted context probability $p_c^*(T(C))$ as

$$p_c^*(T(C)) = log_{10}(p_c(T(C)) \cdot 100 + 0.1) + 1$$

The final language model probability $p_{lm}(t(w)|T(C))$ for a predicted word or type $t(w)$ with $w \in W$ given the typed context words $T(C)$ is then computed as

$$p_{lm}(t(w)|T(C)) = p(t(w)|w_{BOS}, w_1^t, w_2^t, ...w_i^t) \cdot p_c^*(T(C))$$

4.4.2. Insertion Score

Depending on whether the set of insertion context words $I(C)$ is empty, the insertion score $s_{insert}(w|C)$ for a candidate entity $w \in W_{E \geq 15}$ given the context words C is either the normalized sitelink count $s_{sitelinks}(w)$ or the co-occurrence score $s_{coocc}(w|I(C))$ as explained in detail in Section 4.3.

Normal words, for which we do not use a prominence score or co-occurrence counts, are assigned an insertion score that balances the prediction of entities versus normal words. We use word vector similarity for this normal word insertion score if $I(C) \neq \emptyset$. The intuition for this is the following: if an entity is present in the list of context words and we use only the language model probability and constant factors to score normal words, these predictions do not have any link to the entity in the context words other than its type. However, certain normal words will be more likely to follow a specific entity than others. If for example the context words contain the entity "*[Q1779:Louis Armstrong]*", the words "*play*" or "*perform*" should be scored higher than if the context words contained the entity "*[Q567:Angela Merkel]*". Similarly, we use word vectors for the scoring of normal words, if the context words contain a *<type>* word from a question that starts with the words "*which <type>*". This is to emphasize the particular importance of this word to any further completion predictions. We create a Word2Vec model using the Python library Gensim [21]. The model is trained on the Wikipedia dump with entity mentions described in Section 4.1.2 which is also used to compute the co-occurrence counts. For the training, the sentences are pre-processed in the following way: Entities are replaced by their unique QID in the format *[<qid>]*. Entities with a sitelink count < 15 are filtered out. Special tokens such as punctuation are filtered out. Stopwords are filtered out and the remaining words are converted to lowercase and lemmatized. Words that appear less than 5 times in the corpus are discarded during training. We achieved

best results using a model with an embedding size of 200 trained over 20 epochs. For the normal word insertion score, we compute the cosine similarity between the vector of the predicted normal word and the mean of the vectors of the insertion context words $I(C)$ and lemmatized non-stopwords in the context words denoted as $L(C)$. We denote the resulting word vector similarity as $s_{sim}(w, I(C) \cup L(C))$. For a proper balance between the prediction of normal words versus entities, we multiply the word vector similarity with a factor of 0.1. If $I(C) = \emptyset$, we use a constant insertion score of 0.01. Formally, the insertion score for a normal word $w \in W_N$ is computed as

$$s_{nw\_insert}(w|C) = \begin{cases} s_{sim}(w, I(C) \cup L(C)) \cdot 0.1 & if\ I(C) \neq \emptyset \\ 0.01 & else \end{cases}$$

The insertion score $s_{insert}(w|C)$ for any completion prediction $w \in W$ given the context words C can thus be defined as

$$s_{insert}(w|C) = \begin{cases} s_{sitelinks}(w) & if\ w \in W_E \wedge I(C) = \emptyset \\ s_{coocc}(w|I(C)) & if\ w \in W_E \wedge I(C) \neq \emptyset \\ s_{nw\_insert}(w|C) & else \end{cases}$$

4.4.3. Penalty Factors

Penalty factors are multiplied to the final score in three scenarios: 1) when an entity is predicted directly following another entity, 2) when the type *[Q5:human]* is predicted and 3) when the current word prefix matches an alias of the predicted entity, but not its label.

If an entity is predicted directly following another entity, a penalty factor of 0.04 is multiplied to the final score of the completion prediction. Given the context words $C = (w_1, w_2, ...w_i)$ we define the penalty factor $g_{ce}(w)$ for the completion prediction $w \in W$ as

$$g_{ce}(w|C) = \begin{cases} 0.04 & if\ w \in W_E \wedge w_i \in W_E \\ 1 & else \end{cases}$$

This penalty is partly necessary due to erroneous entity recognition in the language model training dataset. In some cases, the entity recognition system identifies a sequence of words as two entities that actually are part of the same entity. When

creating the training corpus, we counteracted flawed entity recognition by merging consecutive identical entities where the original words together form the entity label. There exist cases, however, where counteracting the issue is not as straight forward and flawed entity recognition was not resolved. Consider for example the sentence "*When was the [Q7350:Panama Canal/Canal] [Q498979:Panama Canal Zone/Zone] garrison reinforced ?*" with entities in the format *[<QID>:<entity label>/<original word>]*. Cases like this still exist in the question corpus and are one reason why a penalty for consecutive entities is necessary. Another reason is that the kind of questions that contain consecutive entities, where they are not falsely recognized as such, are rarely asked by a search engine user, yet are commonly generated from Wikipedia sentences. These are questions like "*Who is an [Q41323:American football/American football] [Q918224:wide receiver/wide receiver] who is a free agent ?*". Questions like these are frequently generated from sentences which describe a particular entity. They are asked with a very specific answer in mind. They might be a perfect question for a quiz show. However, a user looking for information they do not yet have will rarely ask this kind of question. By penalizing consecutive entities, we diminish the negative effect which such questions have on the quality of our completion predictions.

If the type "*human*" is predicted, a penalty factor 0.2 is multiplied to the final score of the completion prediction. We define the penalty factor $g_h(w)$ for the completion prediction $w \in W$ as

$$g_h(w) = \begin{cases} 0.02 & if\ t(w) = \text{“}Q5 : human'' \\ 1 & else \end{cases}$$

The type "*human*" is over-represented in the language model. This shows for example when the language model predicts that the type "*human*" is more likely to follow the question prefix "*Who is the author of* " than the type "*written work*", or that the type "*human*" is more likely to follow "*When did [human/q92764:Sergey Brin] found* " than the type "*business*". One reason for this behavior is that in the Wikipedia dump, which is the base for our language model training dataset, possessive pronouns such as "*his*", "*her*" or "*their*" are marked as the entity to which they refer. For example "*[Q892:J. R. R. Tolkien/Tolkien] wrote [Q892:J. R. R. Tolkien/his] book in 1937.*" with entities denoted as "*[<QID>:<label>/<original word>]*". In the training dataset, these occurrences of possessive pronouns are replaced by the type "*human*" and an appended " *'s*". By penalizing the prediction of the type "*human*", the quality of the completion predictions is drastically improved.

If the current word prefix matches only an alias of an inserted entity, but not the entity's label, a penalty factor of 0.6 is multiplied to the final score of the completion prediction. Given the current word prefix w_{pre} we define the penalty factor $g_a(w)$ for completion prediction $w \in W$ as

$$g_a(w) = \begin{cases} 0.6 & if\ w \in W_E \wedge \neg(w\ startswith\ w_{pre}) \\ 1 & else \end{cases}$$

The intuition behind this is that while in some special cases a user might be more likely to enter an alias of an entity instead of its label such as "*USA*" instead of "*United States of America*" or "*CDU*" instead of "*Christian Democratic Union*", in general, the opposite is the case. A user will rarely enter "*Olorin*" instead of "*Gandalf*" or "*Big Apple*" instead of "*New York City*". Therefore, entities whose label matches the current word prefix should have an advantage over entities whose label does not.

4.4.4. Final Score

The final score $s(w|C)$ for a completion prediction $w \in W$ given the context words C is computed as

$$s(w|C) = p_{lm}(t(w)|T(C)) \cdot \big(s_{insert}(w|C) \cdot g_a(w)\big)^{0.3} \cdot g_{ce}(w|C) \cdot g_h(w)$$

The concrete values of the factors and exponents used to compute the final score were determined by experimentation.

The completion predictions are ranked according to their final score.

4.5. Post-Processing

4.5.1. Appending Entities using Word Vector Similarity

If the set of insertion context words is not empty and the number of completion predictions computed via the previously described method is smaller than the number of predictions that are supposed to be displayed to the user, an additional set of completion predictions is computed by using word vector similarity to score candidate

entities. To this end, we use the same Word2Vec model as for computing the normal word insertion score described in Section 4.4.2. We compute the cosine similarity between the vector of a candidate entity $w \in W_{E \geq 15}$ and the mean of the vectors of the insertion context words $I(C)$. We denote the resulting word vector similarity as $s_{sim}(w, I(C))$. As we observed already in our previous work, the Word2Vec model suffers from a major shortcoming. Rare entities in general seem to have a higher similarity to other entities than more common ones. For example, the similarity between the entity "*[Q567:Angela Merkel]*" and "*[Q713750:West Germany]*" is much higher (0.44) than the similarity between "*[Q567:Angela Merkel]*" and "*[Q183:Germany]*" (0.35). This is despite the fact that "*[Q567:Angela Merkel]*" has a much lower co-occurrence with "*[Q713750:West Germany]*" (8) than with "*[Q183:Germany]*" (307). To counteract this flaw, we compute the word vector insertion score as a combination of the word vector similarity and the candidate entity's normalized sitelink count using the formula

$$s_{w2v}(w|I(C)) = s_{sim}(w, I(C)) \cdot (ln(s_{sitelinks}(w) + 1) - (ln(2) + 1)) \cdot 0.01$$

where $s_{sitelink}(w)$ is the normalized sitelink count of the candidate entity w as defined in Section 4.3.1. The new set of completion predictions is ranked according to this score and then appended to the previously computed completion predictions. This way, they are always ranked lower than co-occurrence-based completion predictions. The reason for this is that completion predictions made on the basis of co-occurrence in general have a higher quality than Word2Vec-based completion predictions. Word2Vec-based completion predictions are therefore only used to fill up the completion predictions if not enough predictions were produced using co-occurrence.

4.5.2. Appending Completely Typed Entities

If a current word prefix is identical to an entity's label, we add the entity to the predictions presented to the user - regardless of its score. The reason for this is, that a user has no means of telling our system which entity they want to type other than typing out its full label. If the user enters the complete entity label and the entity is still not predicted, the user has no choice but to continue typing their question and leave the entity unrecognized. This of course, has a negative impact on any further predictions. Therefore, completely typed entities are added to the completion predictions presented to the user. The only restriction to adding completely typed

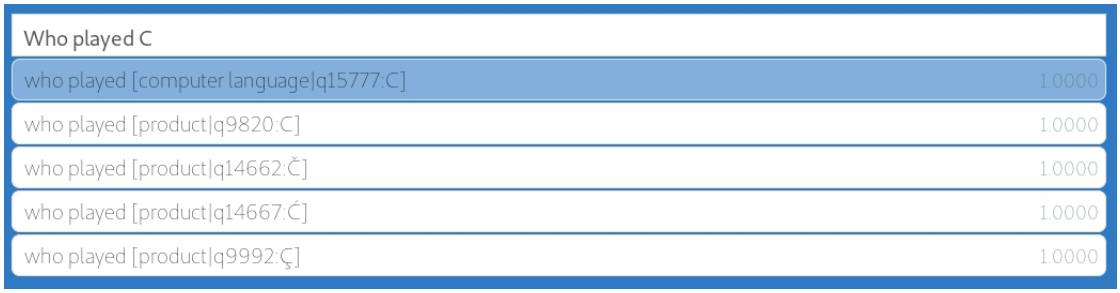


Figure 5.: Minimum label length of completely typed entities: Completely typed entities are only added if the current word prefix is at least 4 characters long. Otherwise, usually undesired but completely typed entities clutter up the predictions presented to the user as shown in this example.

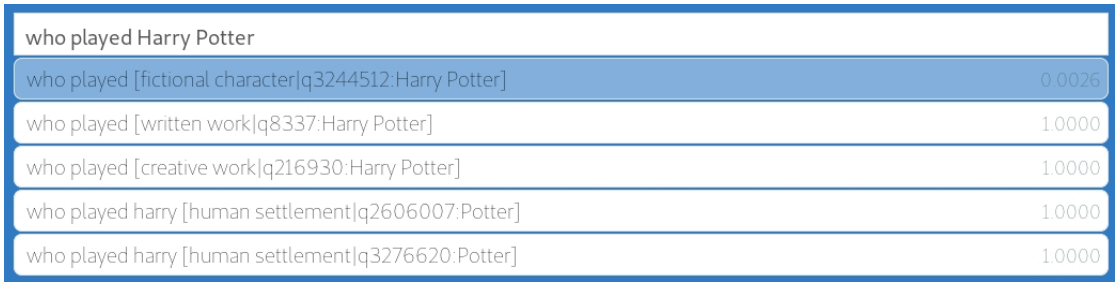


Figure 6.: Sorting order of completely typed entities: Completely typed entities are sorted primarily by the length of their label, i.e. "*Harry Potter*" before "*Potter*" and secondarily by their sitelink count. In this example, the fictional character "*Harry Potter*" is presented before the written work even though it has a lower sitelink count. This is because it was not appended as a completely typed entity during post-processing but it was predicted as completion in the regular prediction and entity insertion process.

entities is that the current word prefix has to be at least 4 characters long. Without this restriction, the completion predictions presented to the user are often cluttered up with unwanted predictions since short entity labels are very common. This problem is illustrated in Figure 5. We append completely typed entities primarily in order of the length of their label and secondarily in order of their sitelink count.

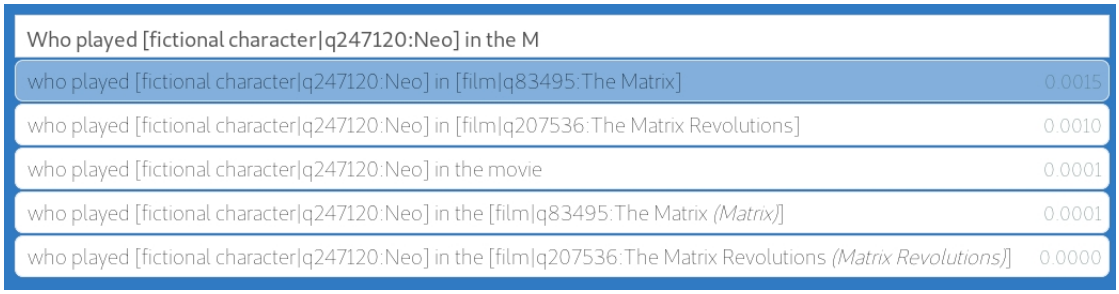


Figure 7.: Removing double completion predictions: Without the removal of double predictions, entities whose label is a substring of one of its aliases or vice versa can appear multiple times in the predictions presented to the user as shown in this example.

4.5.3. Removing Double Completion Predictions

When completion predictions are made for different current word prefixes, the same entity can occur in the predictions several times. This happens, if the label or an alias of the predicted entity is a substring of another alias of the entity or vice versa. Consider for example the question prefix "*Who played [fictional character|Q247120:Neo] in the M*". Without any further processing, the system will yield the predictions shown in Figure 7. As can be seen, both "*[Q83495:The Matrix]*" and "*[Q207536:The Matrix Revolutions]*" are predicted twice, once for matching the entity label and once for matching an alias. Showing two or more completion predictions for the same entity is usually a waste of options that can be displayed to the user. In most cases, the entity occurrence that matches the longer current word prefix is the preferred one. We therefore remove predictions that already occur in the predictions for a longer current word prefix.

After these post-processing steps, the top ranked predictions are presented to the user. The complete QAC system requires about 11GB of RAM and takes ca. 3.5 minutes to load.

5. Evaluation

We evaluate our system using two different methods, in the following referred to as multiple-true-completions evaluation and single-true-completion evaluation. For the multiple-true-completions evaluation, we measure precision at 5, average precision and normalized discounted cumulative gain of our system over a manually generated ground truth. This ground truth consists of question prefixes along with a list of reasonable completion predictions for the corresponding question prefix. For the single-true-completion evaluation we measure mean reciprocal rank and required user interaction of our system over three test sets each of which consists of 10,000 questions. For this evaluation, there exists only one true completion prediction per question prefix, namely the one defined by the next word or entity in the given question. The evaluation datasets are described in Section 5.1. The evaluation metrics are explained in Section 5.2. In Section 5.3, we describe the evaluated versions of our system. The results of the evaluation are discussed in Section 5.4.

5.1. Test Sets

The ground truth for the multiple-true-completions evaluation consists of 100 manually generated question prefixes together with a list of completion predictions that are considered reasonable completions for the question prefix. Each completion prediction is annotated with a relevance score of either 2 or 1. A score of 2 indicates that for the given question prefix, the completion prediction is very likely to be a desired completion. A score of 1 indicates that the completion prediction is a plausible completion for the question prefix but is less likely to be the desired completion than a completion scored with 2.

In order to create the test sets for the single-true-completion evaluation, we held back 100,000 questions of the combined WikiQuestions and ClueWeb12 questions described in Section 4.1.2 from training the language model. For each question among this

set of 100,000 questions that stems from the WikiQuestions corpus, we excluded the original Wikipedia sentence from the Wikipedia dump used to train the Word2Vec model and to compute the co-occurrence counts. We exclude questions that contain entities which could not be mapped to any Wikidata item. In order to create our three test sets, we select 10,000 questions from the resulting questions for each test set. The base test set (*base*) simply consists of 10,000 random questions. 5,326 of these questions do not contain any insertion context word. That is, they do not contain any entity and do not start with the words "*which <type>*" where *<type>* would form an insertion context word. 3,414 questions contain exactly one insertion context word and the remaining 1,260 questions contain more than one insertion context word. The one-insertion-context-word test set (*1ICW*) consists of 10,000 questions that contain exactly one insertion context word. The several-insertion-context-words test set (*>1ICW*) consequently consists of 10,000 questions that contain more than one insertion context word. The reasoning behind evaluating the system on these three datasets is that the way entities are predicted if no insertion context word is present in the question prefix differs significantly from the way entities are predicted if one or more insertion context words are already present in the question prefix. Using these additional datasets allows us to examine how different aspects of our system affect the quality of the predictions in these different scenarios.

5.2. Metrics

For the multiple-true-completions evaluation where several completion predictions can be accepted for a given question prefix, we measure precision at k (P@k), average precision (AP) and normalized discounted cumulative gain at k (nDCG@k). In the single-true-completion evaluation, the goal is to produce the completion prediction that matches the next word or entity as given in the question from the test set. For the single-true-completion evaluation we measure mean reciprocal rank (MRR) and required user interaction (RUI).

5.2.1. Precision at k

P@k is the percentage of reasonable prediction completions among the top k predictions yielded by the system. Let $Q_{true}(q)$ denote the set of reasonable completion predictions for a question prefix q as defined in the ground truth. Let $Q^k_{QAC}(q)$ denote

the set of top k completion predictions returned by the QAC system for a question prefix q. P@k is computed as

$$P@k = \frac{|Q_{true}(q) \cap Q^k_{QAC}(q)|}{k}$$

We report the mean P@5 over all question prefixes in the ground truth.

5.2.2. Average Precision

In order to compute AP, we define $r_1, ..., r_n$ as the list of positions at which completion predictions from $Q_{true}(q)$ appear in $Q_{QAC}(q)$. AP is computed as

$$AP = \frac{\sum_{i=1}^{n} P@r_i}{n}$$

For completion predictions from $Q_{true}(q)$ that do not appear in $Q_{QAC}(q)$, we take $P@r_i = 0$. We report the mean average precision over all question prefixes in the ground truth.

5.2.3. Normalized Discounted Cumulative Gain at k

The nDCG@k is computed using the completion relevance scores of 1 and 2 in the ground truth. Completion predictions not present in the ground truth are considered not relevant and are assigned a relevance score of 0. The discounted cumulative gain at position k is computed as

$$DCG@k = rel_1 + \sum_{i=2}^{k} \frac{rel_i}{log_2(i+1)}$$

where rel_i is the relevance score for the completion predicted by the QAC system at rank i. The nDCG@k is computed as

$$nDCG@k = \frac{DCG@k}{IDCG@k}$$

where IDCG@k denotes the ideal discounted cumulative gain, that is, the DCG that is obtained over the completion predictions given in the ground truth sorted by relevance. We report the mean nDCG@5 over all question prefixes in the ground truth.

5.2.4. Mean Reciprocal Rank

Like AP and nDCG, the reciprocal rank (RR) takes into account at which rank the desired completion prediction appears in the list of completion predictions returned by the system. Let r denote the rank of the desired completion prediction. Then RR is simply computed as

$$RR = \frac{1}{r}$$

This formula implies that the rank of a completion prediction plays a role, but its importance decreases with lower ranks. I.e. the difference between a completion being presented at the first versus the second rank is bigger than the difference between a completion being presented at the fourth versus the fifth rank. If the desired completion prediction is not in the list of predictions returned by the system then $RR = 0$. We compute the reciprocal rank for the completion predictions for each word of each sentence after its first character has been typed. We consider only the top five completion predictions returned by the system as this is our default value of completions presented to the user. We report the mean reciprocal rank over all computed RR values.

5.2.5. Required User Interaction

We define the required user interaction as the number of user interactions needed to enter the complete question using the QAC system. Both, typing a letter and selecting a completion prediction offered by the system count as one user interaction. A matching completion prediction is selected as soon as it is presented by the system, regardless of its rank. However, only the top 5 completion predictions generated by the system are presented to the user. We report the number of user interactions as percentage over the number of user interactions needed if the entire question was typed without selecting completion predictions, i.e. over the total number of characters in the question.

5.3. Tested Versions

We compare different versions of our system to illustrate the importance of its components.

One of the most crucial aspects of our system is the insertion of entities, especially when the set of insertion context words is not empty, i.e. when another entity is already present in the question prefix or the question is a "*Which <type>*"-question. We compare three different insertion methods: as a baseline we insert entities purely based on an entity prominence score given by the entity's sitelink count (*sitelinks*). The second insertion method is based on an entity's sitelink count combined with word vector similarity (*sitelinks + w2v*). This is the same method that is used to fill up the completion predictions if not enough predictions were made using co-occurrence as explained in Section 4.5.1. The third insertion method is based on co-occurrence (*co-occurrence*). This is the method that is described in the previous chapter. In this version, the sitelinks + w2v method is used as a fall back if the co-occurrence method fails to produce enough completion predictions that can be presented to the user, i.e. less than five completion predictions.

We perform the multiple-true-completions evaluation over three additional versions of our system. We evaluate how our system performs without the penalty factor for consecutive entities to show its effect on the results (*co-occurrence w/o* g_{ce}). We also evaluate a version of our system that does not penalize the prediction of the type "*human*" (*co-occurrence w/o* g_h). Finally, we examine how filling up the completion predictions with Word2Vec-based predictions affects the results by evaluating a version of the system that does not fill up the completion predictions (*co-occurrence w/o w2v fill-up*).

5.4. Results

We report the results of the multiple-true-completions evaluation for all six versions of our system. The results are shown in Table 1. The "*Time*" column reports the average time the system takes to yield completion predictions for a given question prefix. It should be noted that the actual average time per completion when using our system is significantly lower. The reason for this is that we use caching to exploit the fact that in most cases, the current question prefix is just an extension of a previous question prefix. Many computational steps, such as e.g. extracting candidate entities for a current word prefix, only need to be performed once and can be reused or refined when the user enters the next letter of his question.

The most striking observation that can be made is that all four co-occurrence versions

Multiple-True-Completions Evaluation Results				
	P@5	AP	nDCG@5	Time
sitelinks	0.286	0.375	0.422	**0.64 s**
sitelinks + w2v	0.322	0.464	0.524	0.77 s
co-occurrence w/o g_{ce}	0.338	0.481	0.551	0.70 s
co-occurrence w/o g_h	0.316	0.473	0.543	0.70 s
co-occurrence w/o w2v fill-up	0.340	0.489	0.564	0.65 s
co-occurrence	**0.344**	**0.500**	**0.572**	0.69 s

Table 1.: Multiple-true-completions evaluation results for six different versions of our QAC system.

show a great improvement over the *sitelinks* and *sitelinks + w2v* version for all three metrics (except the *co-occurrence w/o* g_h version for the P@5 metric). While the *sitelinks + w2v* version yields better results than the *sitelinks* version, it also takes considerably longer to yield completion predictions. Comparing the four co-occurrence versions shows that all three examined features, the consecutive entity penalty, the human penalty and the filling up of completion predictions with Word2Vec predictions, improve the quality of the completion predictions. However, filling up the completion predictions seems to have only a minor effect on the completion prediction quality while increasing the average time needed per question prefix. This feature should therefore be considered optional and only implemented if a machine can run the QAC system smoothly.

We report the results of the single-true-completion evaluation for the test sets *base*, *1ICW* and *>1ICW*. Due to the long run time of the single-true-completion evaluation, only the three main versions of our system (*sitelinks*, *sitelinks + w2v* and *co-occurrence*) are evaluated on these test sets. Table 2 shows the results of the single-true-completion evaluation. Note that for the MRR higher and for the RUI lower scores are desired.

The differences in the RUI scores for the *base* test set, in which around half of the questions do not contain an insertion context word at all, are negligible. The RUI scores differ more clearly for the *1ICW* and the *>1ICW* test set. For the *1ICW* test set, the only aspect that is responsible for the differences in the RUI scores is how often entities are predicted instead of normal words. The intuition for this is the following: The order of the normal words is not affected by the insertion method. At

Single-True-Completion Evaluation Results						
	Base Test Set		1ICW Test Set		>1ICW Test Set	
	MRR	RUI	MRR	RUI	MRR	RUI
sitelinks	0.532	0.542	0.537	0.538	0.529	0.525
sitelinks + w2v	**0.533**	0.541	**0.540**	0.535	**0.533**	0.523
co-occurrence	0.531	**0.538**	0.537	**0.529**	0.531	**0.510**

Table 2.: Single-true-completion evaluation results for three versions of our QAC system and three different test sets.

the same time, the entity insertion method for the three versions only differs when the set of insertion context words is not empty. However, in the *1ICW* test set, only one insertion context word exists. Therefore, which entity is predicted given a non-empty set of insertion context words does not influence the RUI score. Hence, the differences in the RUI scores for the *1ICW* test set indicate a slightly different normal word versus entity balancing for the three versions of the system. The *co-occurrence* version in general seems to predict normal words slightly more readily than the other versions. For the *>1ICW* test set, the difference between the *co-occurrence* version and both other versions is further increased. A reasonable explanation for this (which is also supported by an observation made in connection with the MRR scores discussed later in this section) is an improvement in the prediction of entities when the set of insertion context words is not empty for the *co-occurrence* version.

Interestingly, the discrepancy between the *sitelinks* and the *sitelinks + w2v* version on the *>1ICW* test set is almost the same as on the *1ICW* test set. This is unexpected since the *sitelinks + w2v* version should (and does as shown later in this section) improve the prediction of entities which is a key factor for good results on the *>1ICW* test set. One reason for this it that for the Word2Vec-based entity insertions, we prohibit the insertion of an entity that is already present in the set of insertion context words. This is due to the fact that the highest word vector similarity (i.e. 1.0) is naturally observed between two identical word vectors, i.e. identical entities. Without the additional restriction, the system would therefore assign the highest insertion score to candidate entities which already exist in the question prefix. While this is an undesired behavior for most real-world use cases, nearly 1% of all questions in the *>1ICW* test set (94 questions) contain the same entity twice. This is mostly due to erroneous entity recognition as well as the fact

that the entity recognition marks pronouns as entities. Consider for example the question "*Who dismissed [Q23559:Benito Mussolini|Benito Mussolini] from office and ordered [Q23559:Benito Mussolini|him] arrested ?*" with entities in the format *[<QID>:<entity label>|<original word>]*. By prohibiting the insertion of the same entity, the set of characters that need to be typed in order to get to the required question is "*dismbefoffice aobenito mussoliniar*" for the *sitelinks + w2v* version and "*dismbefofaobear*" for the *sitelinks* version. This has a considerable impact on the RUI score especially since entity labels are typically quite long while usual deviations on the other hand are rather small. For future experiments, questions that contain the same entity twice should probably be excluded from the dataset.

It is important to note that the way entities are inserted plays a minor role in the MRR results. This is due to the fact that in almost all questions, the number of normal words that can contribute to a difference in the MRR scores due to a different normal word versus entity balancing is higher than the number of entities that can contribute to a difference in the MRR. Moreover, even for the most successful entity insertion method (co-occurrence), only about 10% of all entities are predicted correctly given a non empty set of insertion context words if just their first letter is given as current word prefix as is the case for the MRR computation.

We examined the results of the MRR measurement more closely and found that the three versions differed significantly when looking at how many entities were correctly predicted given a non-empty set of insertion context words. The number of correctly predicted entities for $I(C) \neq \emptyset$ on the *>1ICW* test set are shown in Table 3.

Correctly predicted entities for $I(C) \neq \emptyset$		
	entity $\in I(C)$	*<type>* $\in I(C)$
sitelinks	468	200
sitelinks + w2v	718	195
co-occurrence	**1062**	**280**

Table 3.: Number of correctly predicted entities after their first character has been typed when the set of insertion context words is not empty. I.e. $I(C)$ contains an entity or a *<type>* word. Evaluated over the *>1ICW* test set.

These results drastically demonstrate the superiority of co-occurrence-based entity insertion not only over the *sitelinks* version, which was to be expected, but also over the *sitelinks + w2v* version. Interestingly, while the *sitelinks + w2v* version

significantly outperforms the *sitelinks* version when an entity is present in the insertion context words, both versions perform similarly when a <*type*> word is present in the insertion context words. This is in line with an observation we made in our previous work. When including normal words in the set of insertion context words while using Word2Vec-based entity insertion, the results often deteriorated. Related to this is the observation that the Word2Vec model exhibits higher word vector similarities between entities that share the same type. The word vector similarity between two words is not primarily higher because they appear more frequently together in a sentence, but because they appear more frequently in the same context. In some scenarios, e.g. when computing the word vector similarity between normal words and entities, this can lead to undesirable results and is a reason why co-occurrence-based entity insertion yields better results.

6. Conclusion

In this work, we presented a QAC system that produces single-word completion predictions for a question prefix entered by a user. The QAC system uses an LSTM language model such that it does not depend on large query logs. We address the issue of data sparsity, which is typically encountered when working with language models, by deploying a typed language model. Entities are inserted for types predicted by the language model using an entity prominence score, co-occurrence and word vector similarity. Our QAC system achieves promising overall results in the evaluation. The evaluation further illustrates the importance of various components of our system e.g. by demonstrating the positive effects of applying penalty factors in certain scenarios. Moreover, we show the superiority of co-occurrence-based entity insertion compared to using a pure entity prominence score or word vector similarity as we did in our previous work.

6.1. Future Work

When scoring completion predictions that were made on the basis of an entity's alias, all aliases are deemed equally likely in the current version of our system. That is, while alias-based completion predictions are penalized such that label-based completion predictions are prioritized as described in Section 4.4.3, our system deems the alias "*NYC*" as just as likely to be entered instead of the entity label "*New York City*" as the alias "*the five boroughs*". Making use of an entity to alias mapping that assigns a probability to each alias could improve the ranking of the completion predictions.

Most QAC systems of major search engines are robust against spelling errors made by the user. Spelling errors occur quite frequently, especially when using mobile devices. The user experience of our QAC system could therefore potentially be improved by making it robust against spelling errors. Robustness against spelling errors of course comes with a price as it increases the number of words and entities that match a given

current word prefix. This increases the number of words and entities that need to be scored and ranked which leads to higher computational time and space requirements of our system. A more efficient processing of possible completion predictions might be necessary in order to keep the system responsive.

A lot of research in QAC focuses on improving the quality of completion predictions by exploiting temporal or contextual information ([6], [5], [29]). As any QAC system, our system could potentially be improved by adding contextual clues such as the user's recent queries. Investigating the possibilities these clues can offer is left for future research.

Research in natural language processing is becoming increasingly fast-paced. A recent novelty has been the introduction of transformer models [25]. Results achieved by transformer architectures such as OpenAI's GPT [26] or Google's BERT [27] show that transformer models or variations of them [28] could potentially outperform standard LSTM models in language modeling tasks. Future research could show whether using a transformer-based language model for our typed language model could further improve the results of our system.

Properly evaluating our QAC system is not a trivial task. Since our focus is on a very particular set of queries, i.e. factoid questions, and entities in the ground truth need to be recognized, we cannot simply rely on existing query logs for our evaluation. The questions from the test sets we use in our single-true-completion evaluation are often not of the desired quality. Some questions are ungrammatical, others suffer from erroneous entity recognition. The manually created ground truth for the multiple-true-completions evaluation on the other hand is with just 100 question prefixes rather small. Creating a larger ground truth of question prefixes along with reasonable completion predictions could help in gaining more reliable insights on the effects that certain components have on the quality of our system's completion predictions.

7. Acknowledgments

I would like to thank Prof. Dr. Hannah Bast for providing me with thorough and incredibly valuable advice throughout the course of my thesis.

My sincere gratitude to Niklas Schnelle for his constructive and always motivating feedback. His helpful and encouraging remarks were vital in completing this thesis.

I also would like to thank Dr. Fang Wei-Kleiner for agreeing to take on the role of the second examiner for my thesis.

I want to thank my brother Timo Prange for proof reading this thesis. (You made it!)

Finally, I would like to express my heartfelt gratitude to my family and friends for their unwavering support throughout the course of my studies and the completion of this thesis.

A. Appendix

A.1. List of Preferred Primary Types

Q5: human
Q11424: film
Q7366: song
Q482994: album
Q15416: television program
Q11410: game
Q3305213: painting
Q25379: play
Q95074: fictional character
Q2088357: musical ensemble
Q2385804: educational institution
Q7278: political party
Q15265344: broadcaster
Q12973014: sports team
Q4830453: business
Q476300: competition
Q15275719: recurring event
Q27968055: recurrent event edition
Q2627975: event
Q6256: country
Q486972: human settlement
Q271669: landform
Q15324: body of water
Q56061: administrative territorial entity
Q6999: astronomical body
Q42889: vehicle

Q811979: architectural structure
Q13226383: facility
Q629206: computer language
Q7397: software
Q178706: institution
Q431289: brand
Q178885: deity
Q4164871: position
Q216353: title
Q618779: award
Q12737077: occupation
Q34770: language
Q25295: language family
Q2695280: technique
Q121359: infrastructure
Q1792379: art genre
Q41710: ethnic group
Q34379: musical instrument
Q336: science
Q9174: religion
Q20978643: point of view
Q7257: ideology
Q349: sport
Q20202269: music term
Q47461344: written work
Q1790144: unit of time

A.2. List of Preferred Secondary Types

Q27096213: geographic entity
Q483394: genre
Q43229: organization
Q17376908: languoid
Q795052: individual

Q17537576: creative work
Q15401930: product
Q43460564: chemical entity
Q16521: taxon
Q853614: sign
Q20978643: point of view
Q4164871: position
Q12737077: occupation
Q216353: title
Q1190554: occurrence
Q16334295: group of humans
Q336: science
Q9081: knowledge
Q21070598: character that may be fictional
Q618779: award
Q28877: goods
Q1790144: unit of time
Q43460564: chemical entity

A.3. *Leads To* Rules for Secondary Types

Q7397: software $\xrightarrow{\text{leads to}}$ Q15401930: product
Q2695280: technique $\xrightarrow{\text{leads to}}$ Q15401930: product
Q42889: vehicle $\xrightarrow{\text{leads to}}$ Q15401930: product
Q431289: brand $\xrightarrow{\text{leads to}}$ Q15401930: product

Bibliography

[1] A. Zhang, A. Goyal, W. Kong, H. Deng, A. Dong, Y. Chang, C. A. Gunter, and J. Han, "adaqac: Adaptive query auto-completion via implicit negative feedback," in *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pp. 143–152, ACM, 2015.

[2] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search.," in *InfoScale*, vol. 152, p. 1, 2006.

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[4] D. Vrandečić and M. Krötzsch, "Wikidata: a free collaborative knowledge base," 2014.

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

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

[7] H. Bast and I. Weber, "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*, pp. 364–371, ACM, 2006.

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

[9] D. H. Park and R. Chiba, "A neural language model for query auto-completion," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1189–1192, ACM, 2017.

[10] N. Fiorini and Z. Lu, "Personalized neural language models for real-world query auto completion," *arXiv preprint arXiv:1804.06439*, 2018.

[11] N. Prange, "Query auto-completion using an abstract language model," *Bachelor's thesis at the University of Freiburg*, 2016.

[12] K. Bollacker, C. Evans, P. Paritosh, T. Sturge, and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1247–1250, AcM, 2008.

[13] H. Bast, F. Bäurle, B. Buchhold, and E. Haußmann, "Easy access to the freebase dataset," in *Proceedings of the 23rd International Conference on World Wide Web*, pp. 95–98, ACM, 2014.

[14] P. F. Brown, P. V. Desouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–479, 1992.

[15] W. Ward and S. Issar, "A class based language model for speech recognition," in *1996 IEEE International Conference on Acoustics, Speech, and Signal Processing Conference Proceedings*, vol. 1, pp. 416–418, IEEE, 1996.

[16] C. Samuelsson and W. Reichl, "A class-based language model for large-vocabulary speech recognition extracted from part-of-speech statistics," in *1999 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings. ICASSP99 (Cat. No. 99CH36258)*, vol. 1, pp. 537–540, IEEE, 1999.

[17] J. Goodman, "Classes for fast maximum entropy training," *arXiv preprint cs/0108006*, 2001.

[18] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, "Building language models for text with named entities," *arXiv preprint arXiv:1805.04836*, 2018.

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

[20] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, pp. 391–407, 1990.

[21] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," in *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, Citeseer, 2010.

[22] N. Prange, "Wikiquestions - a wikipedia-based factoid question dataset," *Master's project at the University of Freiburg*, 2019.

[23] E. Gabrilovich, M. Ringgaard, and A. Subramanya, "Facc1: Freebase annotation of clueweb corpora, version 1," *Release date*, pp. 06–26, 2013.

[24] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[25] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[26] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, 2019.

[27] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[28] C. Wang, M. Li, and A. J. Smola, "Language Models with Transformers," *arXiv e-prints*, p. arXiv:1904.09408, Apr 2019.

[29] J.-Y. Jiang, Y.-Y. Ke, P.-Y. Chien, and P.-J. Cheng, "Learning user reformulation behavior for query auto-completion," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 445–454, ACM, 2014.