



Modeling

Let's look at the modeling options for entity linking.

We'll cover the following



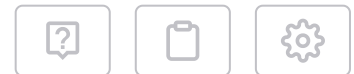
- Contextualized text representation
 - ELMo
 - BERT
- NER modelling
 - Contextual embedding as features
 - Fine-tuning embeddings
- Disambiguation modeling
 - Candidate generation
 - Linking

The first step of entity linking is to build a representation of the terms that you can use in the ML models. It's also critical to use contextual information (i.e., other terms in the sentence) as you embed the terms. Let's see why such representation is necessary.

Contextualized text representation#

It is often observed that the same words may refer to a different entity. The context (i.e., other terms in the sentence) in which the words occur helps us figure out which entity is being referred to. Similarly, the NER and NED models require context to correctly recognize entity type and disambiguate respectively. Therefore, the representation of terms must take contextual

information into account.



One way to represent text is in the form of embeddings. For instance, let's say you have the following sentences:



1

Michael Jordan is the best professor at UC Berkeley.



2

Regarded by most as the NBA's greatest all-time player, Michael Jordan resides in Florida.

The words "Michael Jordan" refer to different people in the two sentences

When you generate the embedding for the words “Michael Jordan”, through traditional methods such as Word2vec, the embedding would be the same in both sentences. However, you can see that, in the first sentence, “Michael Jordan” is referring to the UC Berkeley professor. Whereas, in the second sentence, it is referring to the basketball player. So, the embedding model needs to consider the whole sentence/context while generating an embedding for a word to ensure that its true meaning is captured.

Notice that, in the first sentence, the context that helps to identify the person comes after the mention. Whereas, in the second sentence, the helpful context comes before the mention. Therefore, the embedding model needs to be bi-directional, i.e., it should look at the context in both the backward direction and the forward direction.

Two popular model architectures that generate term contextual embeddings are:

1. ELMo
2. BERT

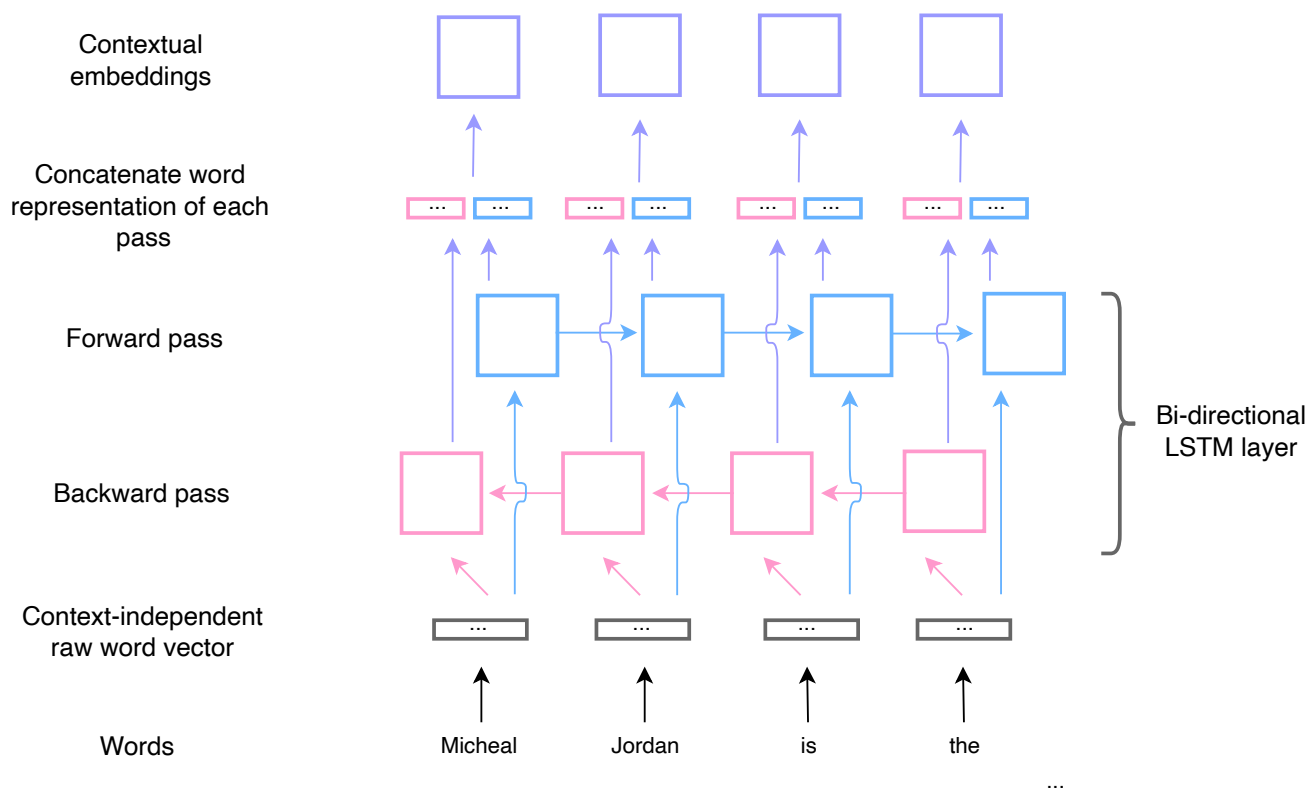


ELMo#



Let's see how ELMo (Embeddings from Language Models) generates contextual embeddings. It starts with a character-level convolutional neural network (CNN) or context-independent word embedding model (e.g., Word2vec) to represent words of a text string as raw word vectors. The raw vectors are fed to a bi-directional LSTM layer trained on a language modeling (*generating the next word in a sentence conditioned on previous words*) objective. This layer has a **forward pass** and a **backward pass**.

The forward pass sequentially goes over the input sentence from left to right. It looks at the context (words) **before** the active word. Whereas, the backward pass sequentially goes over the input sentence from right to left. It looks at the context (words) **after** the active word to predict the current word. Contextual information from both these passes is concatenated and then combined in another layer to obtain contextual embeddings of the text.



ELMo (unravalled)

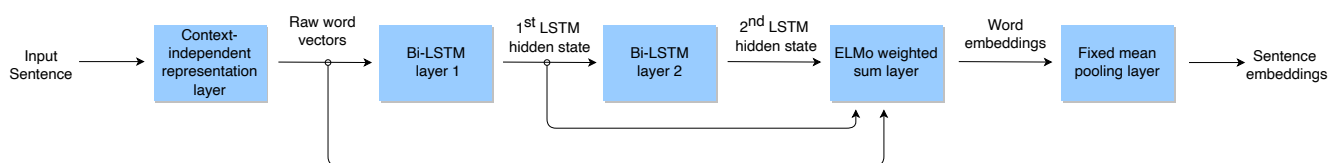




✎ If the raw word vector is made using a character level CNN, the inner structure of the word is captured. For instance, the similarity in the structure of the words “learn” and “learning” will be captured, which will be helpful information for the bi-directional LSTM layer.

The character level CNN will also make good raw vectors for out-of-vocabulary words by looking at their similarity with the vocabulary observed during the training phase.

ELMo can be used to obtain *both word embeddings and sentence embeddings*. Below is a common implementation of ELMo, based on this [paper](#).

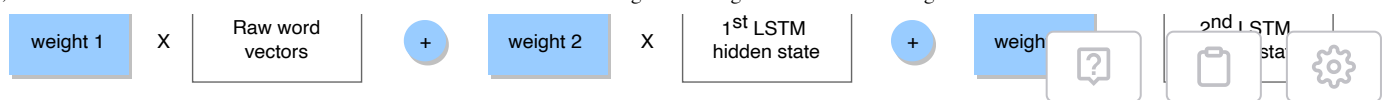


A common implementation of ELMo

✎ Adding more layers allows the model to learn even more context from the input. The initial layers help identify grammar and syntactic rules while the deeper layers help extract higher contextual semantics.

The input sentence is fed to the context-independent representation layer. It outputs raw word vectors which are passed on to the first Bi-LSTM layer. The hidden state/ intermediate word vectors from this layer are fed to the second Bi-LSTM layer. It also outputs a hidden state/intermediate word vectors. The ELMo weighted sum layer performs a weighted sum of the outputs of the three previous layers to arrive at the word embeddings.





ELMo weighted sum layer

The weights shown in the above diagram are trainable.

The fixed mean pooling layer takes the word embeddings and converts them into a sentence embedding.

The previous diagram, labeled “ELMo (unravalled)”, zooms in on the context-independent and first Bi-LSTM layer. The intermediate word vectors made by this Bi-LSTM layer can also be taken as wording embeddings.

Although ELMo uses bi-directional LSTM, it is a “*shallow*” *bi-directional model*. It is deemed “shallow” because of how it achieves bi-directionality. There is a sequential pass on the input from:

- Left to right to train the forward LSTM
- Right to left to train the backward LSTM

The forward and backward LSTMs are trained “*independently*”, and only their word representations are concatenated. Therefore, the word representations can’t take advantage of both the left and right contexts “*simultaneously*”.

BERT#

Let’s see how BERT (bidirectional encoder representations from transformers) gives us contextual word embeddings.

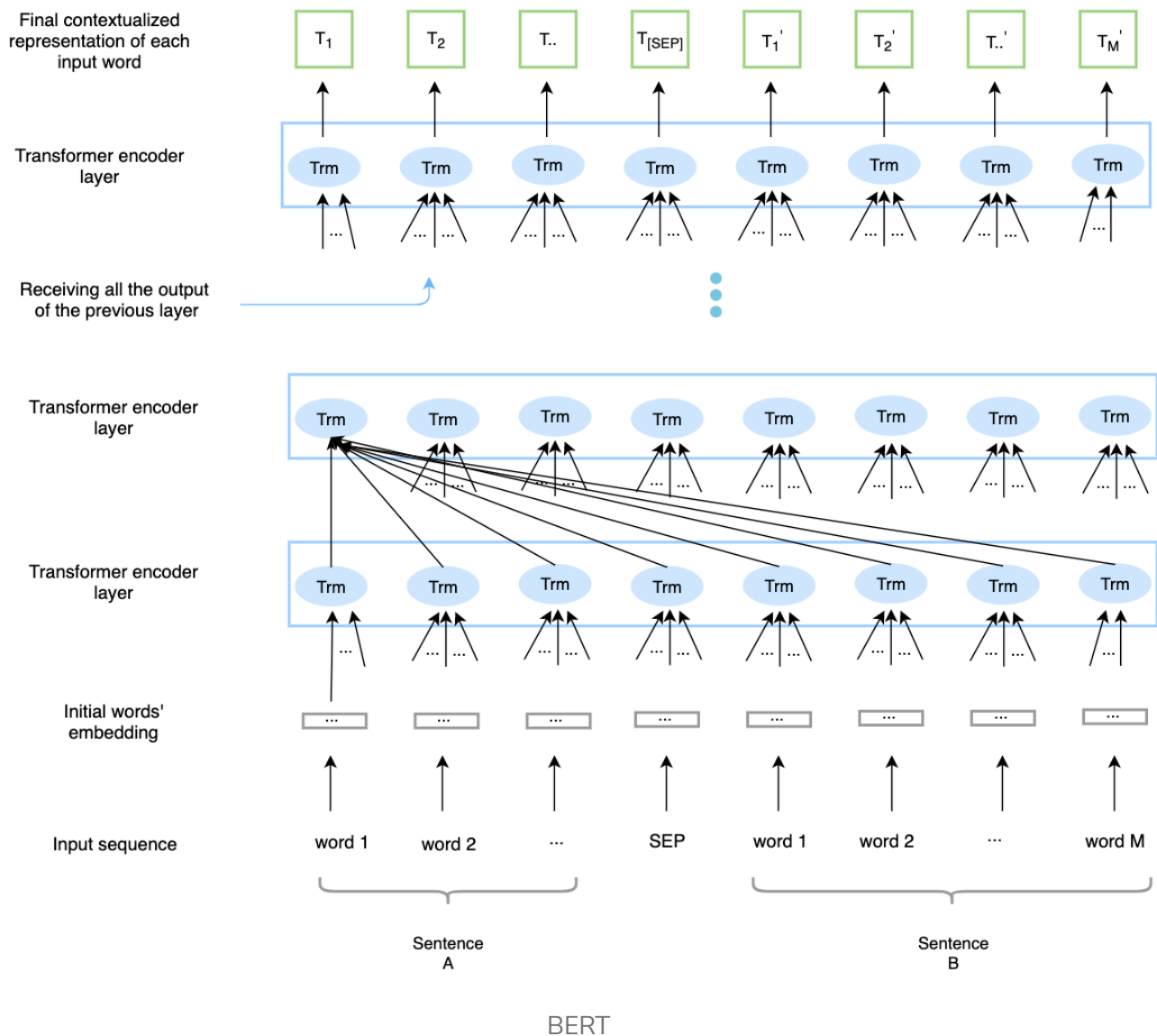


BERT starts by taking in the input sequence which can be made up of

multiple sentences separated by the *SEP* tag. Each word is co



embeddings and fed to the first transformer encoder layer. This layer has the capability to process all the words of the input sequence simultaneously. Therefore, the word representations produced are based on context from both directions, jointly. The output of this transformer layer is passed onto the next layer, and this happens for all the transformer layers. The final transformer layer outputs the contextualized representation of each word.

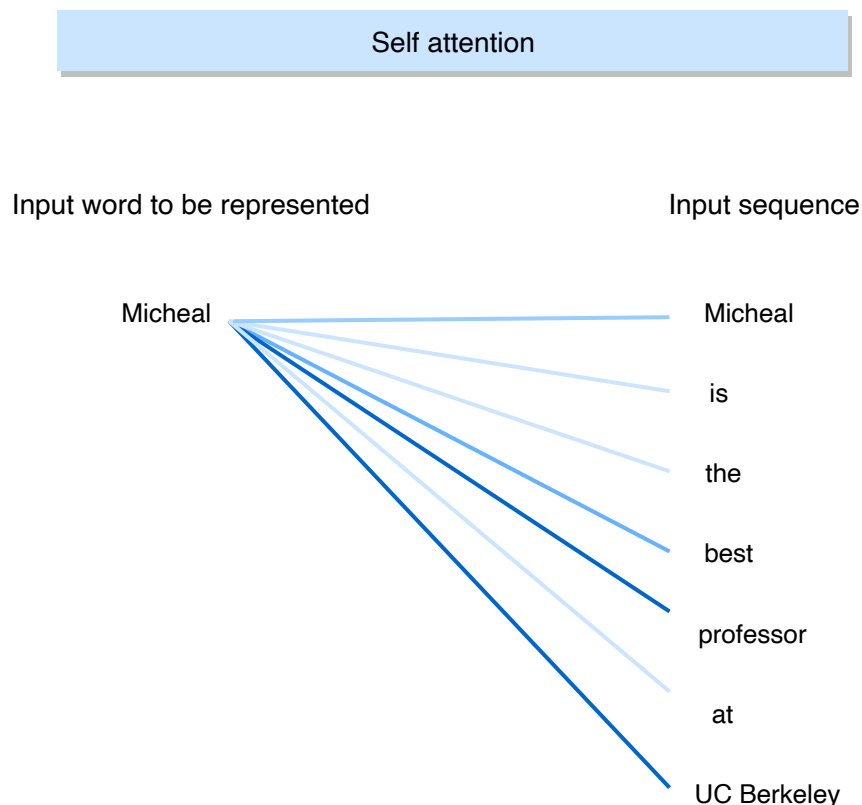


The transformer encoder block uses the concept of *self-attention* while computing the representation for the words in an input sequence. Simply put, self-attention is the process of assigning weights to the words in the





input sequence according to their relevance/contribution to the understanding of the word whose representation is being made.



Deeper color signifies a higher weight assigned to a word for the active word's representation

Self attention in transformer encoder

You see that the words: “professor” and “UC Berkeley”, followed by “best”, are given more weights in the representation of the word “Michael”. These words help the system to understand the active word better.

The following two prediction objectives are used during the training of the model for learning contextualized term representation:

1. Masked language modeling (MLM)

A few words are masked in the input, and the model predicts them based on the bi-directional context

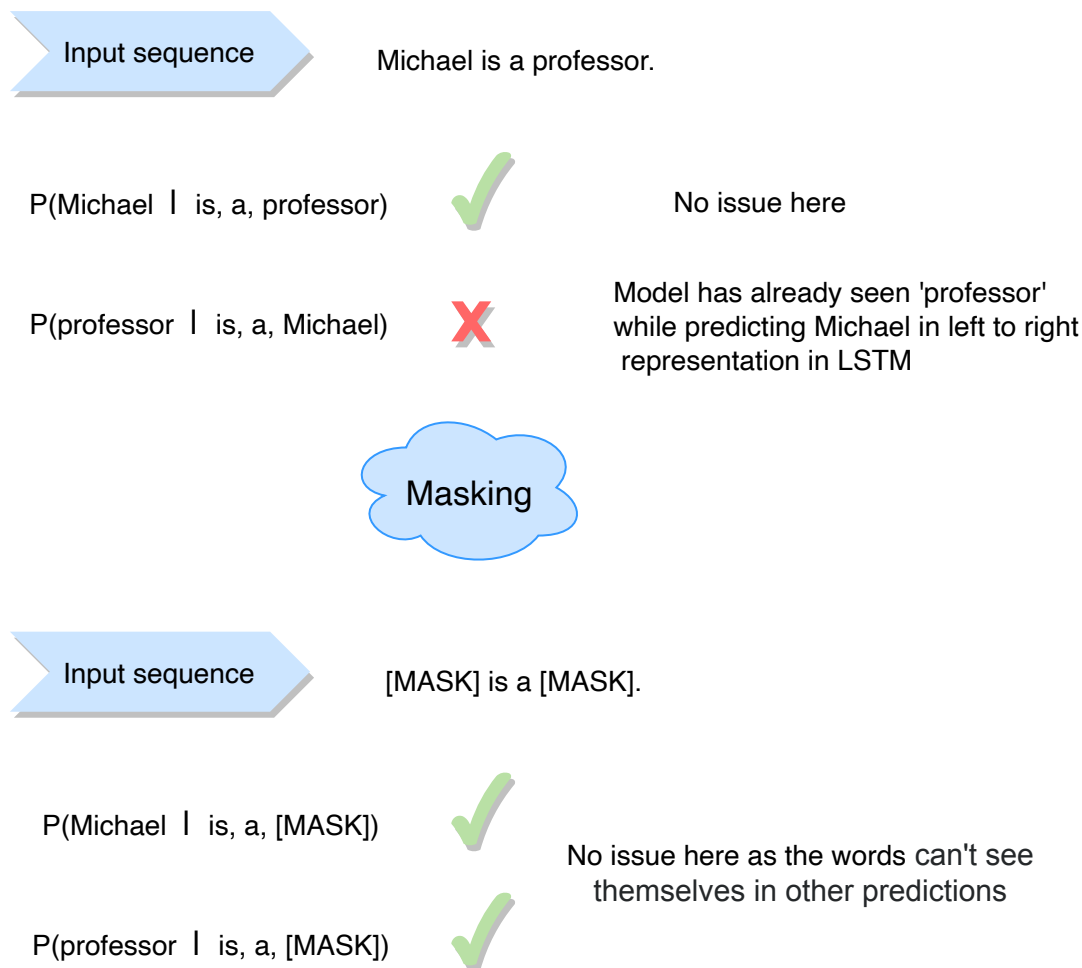





2. Next sentence prediction (NSP)


Two sentences A and B are given. The model has to predict if B comes after A in the corpus or is it just a random sentence.

Let's quickly see how these objectives make BERT a *deeply bi-directional model*, in contrast to ELMo. You will use the MLM objective as an example.



How masking alleviates the problem with bi-directional context in BERT, which you couldn't do earlier with Bi-LSTM

BERT takes the input sequence in one go as compared to sequential input. The problem with looking at the bidirectional context is that the word that's being predicted indirectly "sees itself". It appears in the bi-directional context for the prediction of another word in the input sequence. Masking  helps hide some words so that this problem is alleviated.




 ELMo has solves this problem through sequential prediction from left to right and right to left, independently. This is suboptimal because it results in shallow bi-directionality.

Variations

Let’s look at some variations of BERT.

BERT Variation	Description
BERT base	This is a smaller and faster version. It has 12 layers/transformer blocks and 110 million parameters.
BERT large	This is the larger version. It has 24 layers and 340 million parameters.



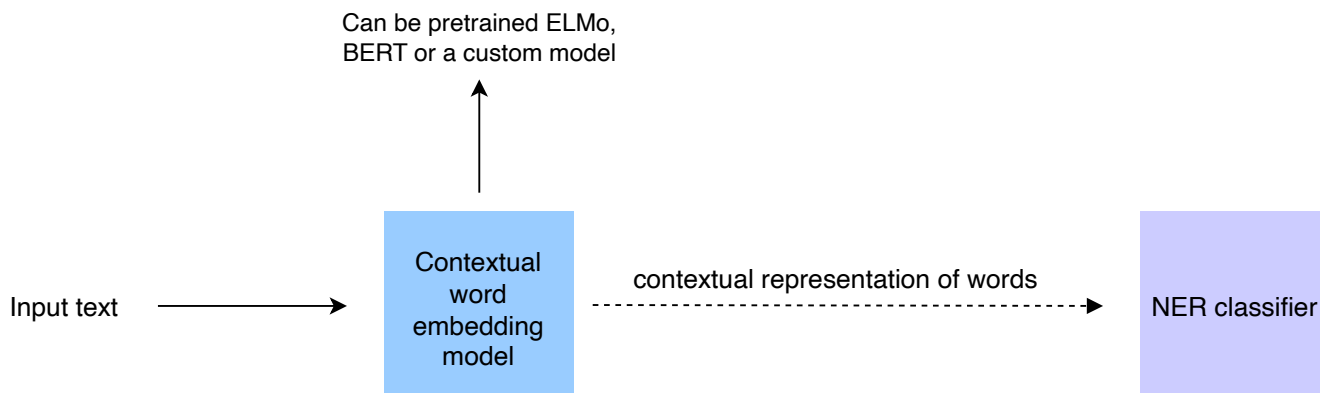
BERT Variation	Description   
DistilBERT	<p>If execution time is a concern, we have versions of BERT that are faster than BERT base. They squeeze the network by smaller embeddings, fewer layers, or some other methods. One such version is called <i>DistilBERT</i>. With a slight compromise on the quality, it retains 97% of its language understanding capabilities and is 60% faster. It makes sense to use DistilBERT to embed tokens when evaluation time performance is critical.</p>
Cased vs uncased BERT	<p>BERT has a cased and uncased variation. It makes sense to use the cased version knowing that case is important in NER (generally the upper case is used when writing names).</p>

NER modelling#

Both ELMo and BERT are trained on massive datasets and have the ability to understand language. With these two models generating contextual word



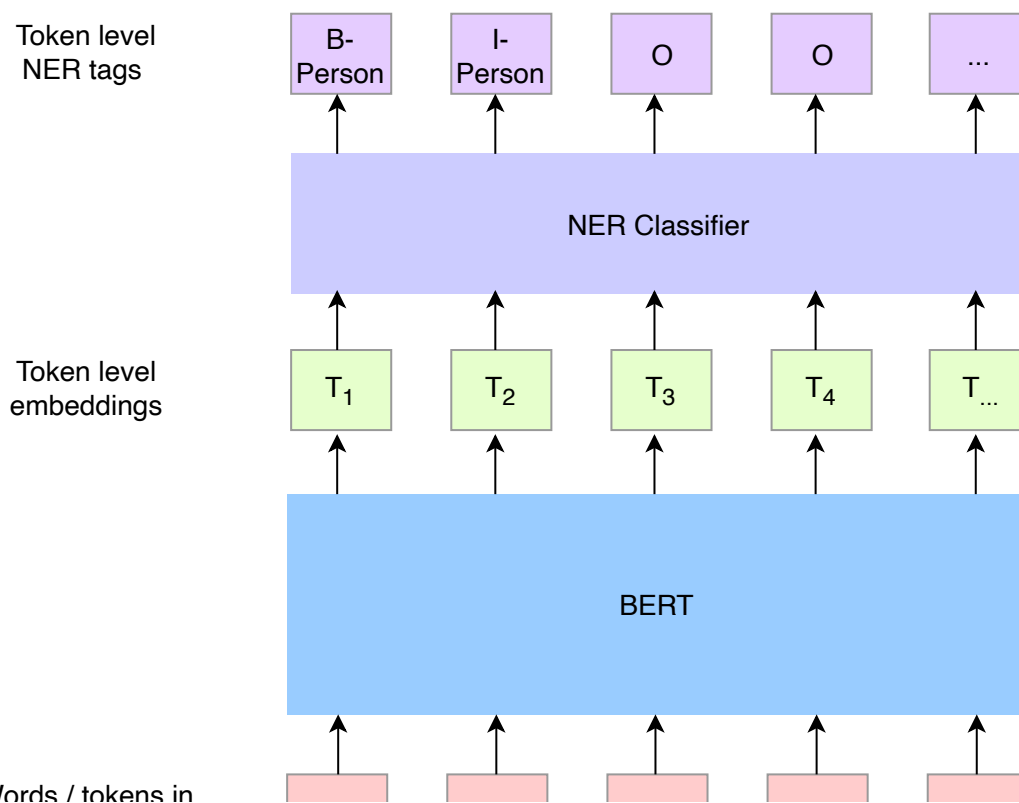
embeddings, half of the work is done. Through the transfer learning approach, you can now utilize these embeddings in a NER classifier.



There are two methods to utilize pre-trained models for the task:

Contextual embedding as features#

One quick way to utilize these contextual embeddings generated by BERT is to use them as features in your NER modelling.



words / tokens in
input sequence

Micheal

Jordan

is


the



Once BERT gives us token level embeddings, you can train a classifier on top of token embeddings to predict NER classes.

Fine-tuning embeddings#

Another option is to take the pre-trained models generated on a large corpus (e.g., BERT base, BERT large, and DistilBERT) and fine-tune them based on your NER dataset to improve the classifier quality. This makes more sense, especially when we have a large NER labelled data set. For BERT fine-tuning, we can either fine-tune the whole model or only top k layers, depending on how much training data you have and also on training time (performance).

 If the interviewer asks you to not use these large pre-trained models due to time or resource constraints, you can build your own customized model based on similar concepts.

Disambiguation modeling#

Once you have identified the entities mentions in the text through NER, it's time to link them to corresponding entries in the knowledge base. As mentioned in the architectural components lesson, the disambiguation process consists of two phases:

1. Candidate generation

In this phase, for each recognized entity, you will select a subset of the knowledge-base entities as candidates that might correspond to it.

2. Linking

In this phase, for each recognized entity, you will select a corresponding entity from among the candidate entities. Thanks to candidate





generation, you will only have to choose from a smaller subset instead of the whole knowledge base.

Candidate generation#

In candidate generation, you will look up the terms in the recognized entities in an index to retrieve candidate entities. So, the process of candidate generation requires building an index where terms are mapped to knowledge base entities. This index will help us in the quick retrieval of candidates for recognized entities. To build this index, you need *ways* to figure out what terms should be used to index each entity in the knowledge base.

Before you start looking at such *ways*, it is important to point out that you need to build the index such that *candidate generation focuses on higher recall*. The index should include all terms that could possibly refer to an entity, even if it is something as trivial as a nickname or a less frequently used term for an entity. The reason behind it is that you do not want to shorten the list of candidates for the linking stage at the cost of missing out on potential matches (one of which could have been the true match).

Now let's look at some of the ways to look for terms to index each entity on.

1. You will index a knowledge base entity on all the terms in its name and their concatenations. For instance, for the entity "Michael Irwin Jordan", the index terms can include "Michael Irwin Jordan", "Michael Jordan", "Michael", "Irwin", "Jordan", "Jordan Michael", and so on. Now, if you encounter any of these terms in the text, you can say that they might be referring to the entity "Michael Irwin Jordan" in the knowledge base.



base data for this purpose. For instance, assume that an anchor text

reads “Michael I. Jordan” and refers to the “Michael Irwin Jordan” entity in the knowledge-base. Here, you will index the entity of the anchor text as well. This way, you will get a flavour of a lot of different ways in which a knowledge base entity may be referred to in text such as nicknames and abbreviations.



The screenshot shows the Wikipedia article for "Latent Dirichlet allocation". The article title is "Latent Dirichlet allocation" and it is from Wikipedia, the free encyclopedia. A notice at the top states: "Not to be confused with linear discriminant analysis." Below this, a message says: "This article may be too technical for most readers to understand. Please help improve it to make it understandable to non-experts, without removing the technical details. (August 2017) (Learn how and when to remove this template message)". The main text begins: "In natural language processing, the latent Dirichlet allocation (LDA) is a generative stochastic probabilistic model that generates a set of topics for a document and a set of words for each topic. LDA is an example of a topic model and belongs to the family of probabilistic graphical models." A sidebar on the left contains links like "Main page", "Contents", "Featured content", etc. A right sidebar contains a box about "Michael Irwin Jordan" and a section titled "History" with a link to "edit".

The document on LDA has anchor text "Michael I. Jordan" that links to the "Michael Irwin Jordan" entity

3. If you are provided with a source that can give us commonly used spellings, aliases, and synonyms for an entity's name, then you can also use these terms for indexing. However, if that is not the case, then you can use the embedding method.

Embedding method

You know about some terms that link to a particular entity by methods such as the two described above. In order to discover more terms to index a particular entity on, you can look for words that are similar to the ones an entity is already indexed on.

The first step to finding similar words is representing all the words in the knowledge base with the help of embeddings. You can use a pre-built embedding model or build one ourselves. The model will try to bring the abbreviation/ aliases/ synonyms, that refer to the same entity closer in the embedding space.



Once you have the embedding of all the words, you can find k nearest

neighboring terms for a particular term that is already l



entity. These k nearest neighboring terms can also be used to index the entity.

Linking#

Once you are done generating candidates from the knowledge base entities for the recognized entity mentions in the given input sentence, you need to perform linking.

In the linking stage, you will build a model that will give us the probability of a candidate being the true match for a recognized entity. You will select the candidate with the highest probability and link it to the recognized entity mention.

Let's look at the linking model's inputs. It will receive:

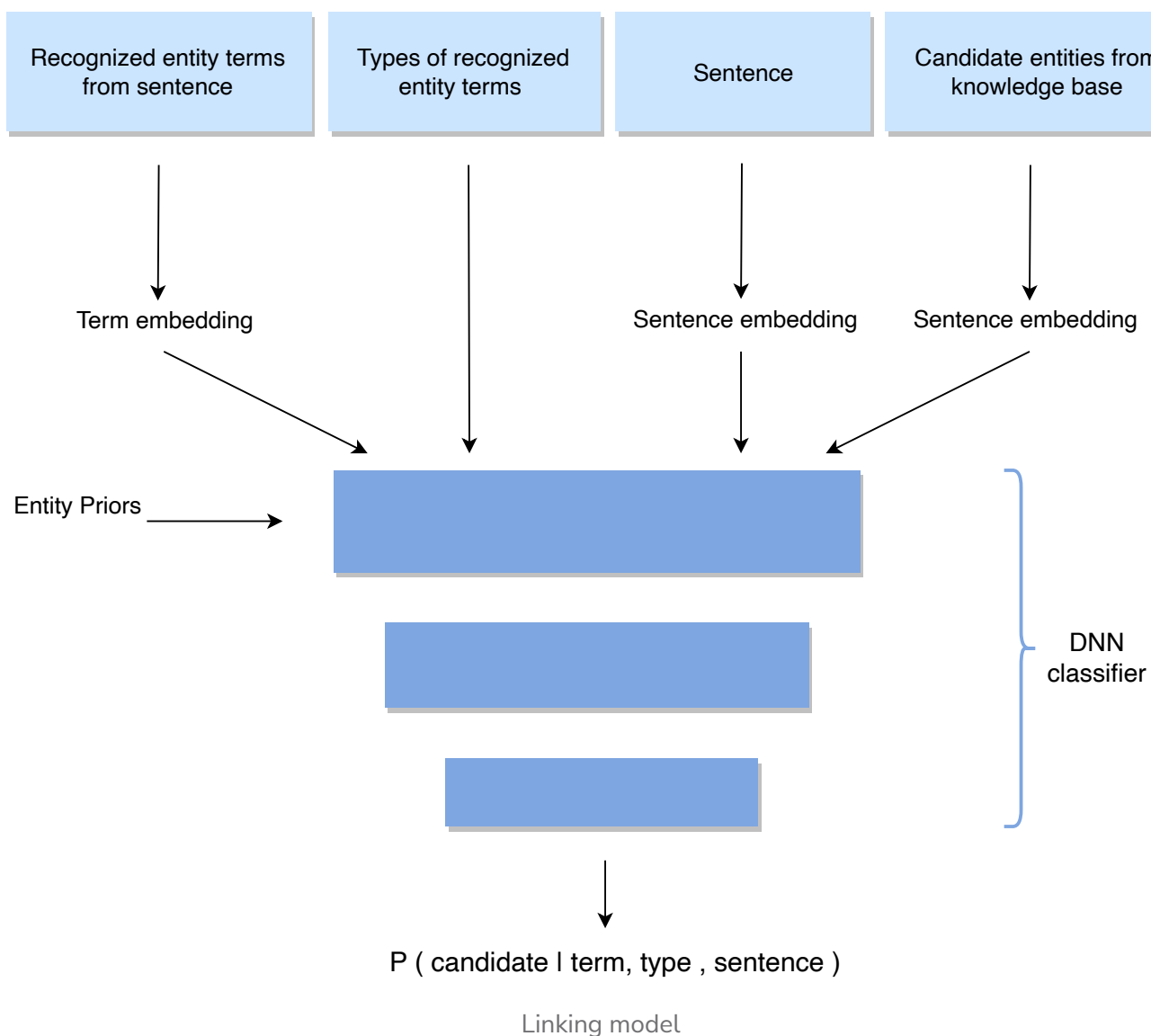
- the recognized entity mention that you want to link
- the type of the entity mention, e.g., location, person etc.
- the whole input sentence in which the entity mention is recognised
- the candidate entity from the knowledge base
- the prior: for a certain entity mention, how many times the candidate entity under consideration is actually being referred to. For example, the anchor text “Harrison Ford”, referred to the entity **American actor** 98% of the time instead of the silent film actor “Harrison Edward Ford”. It's important to tell the model that priors favor a certain entity more.




Utilization of the candidate entity in the anchor text is the “prior” in the above example.

All of these inputs will be fed to a deep neural network (DNN) classifier, which will give us the required probability.





 This stage focuses on precision, i.e., you want to identify the correct corresponding entity for a mention.

It is important to consider how the inputs (entity mention, sentence and candidate entity) will be represented. The best representations are the ones given by contextualized embeddings, which you are generating through models such as BERT and ELMO for NER.

For example, assume that you have used BERT for NER, so you have the contextual embedding for the recognized entity. BERT can also provide

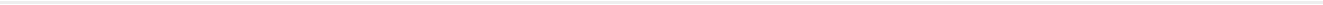


sentence embeddings (along with term embeddings), so you

?

embedding for the entire input sentence. You can also generate the embeddings of the candidate entities based on their representation in the knowledge base. For instance, Wikidata has a small description of each entity in the knowledge base, which can be used to generate an embedding for the candidate entity.

These embeddings, along with the prior and entity type, will provide the model with all the information it needs to produce a good output.



←

Back

Training Data Generation

Next

→

Problem Statement

☒

Mark as Completed



Report an Issue

