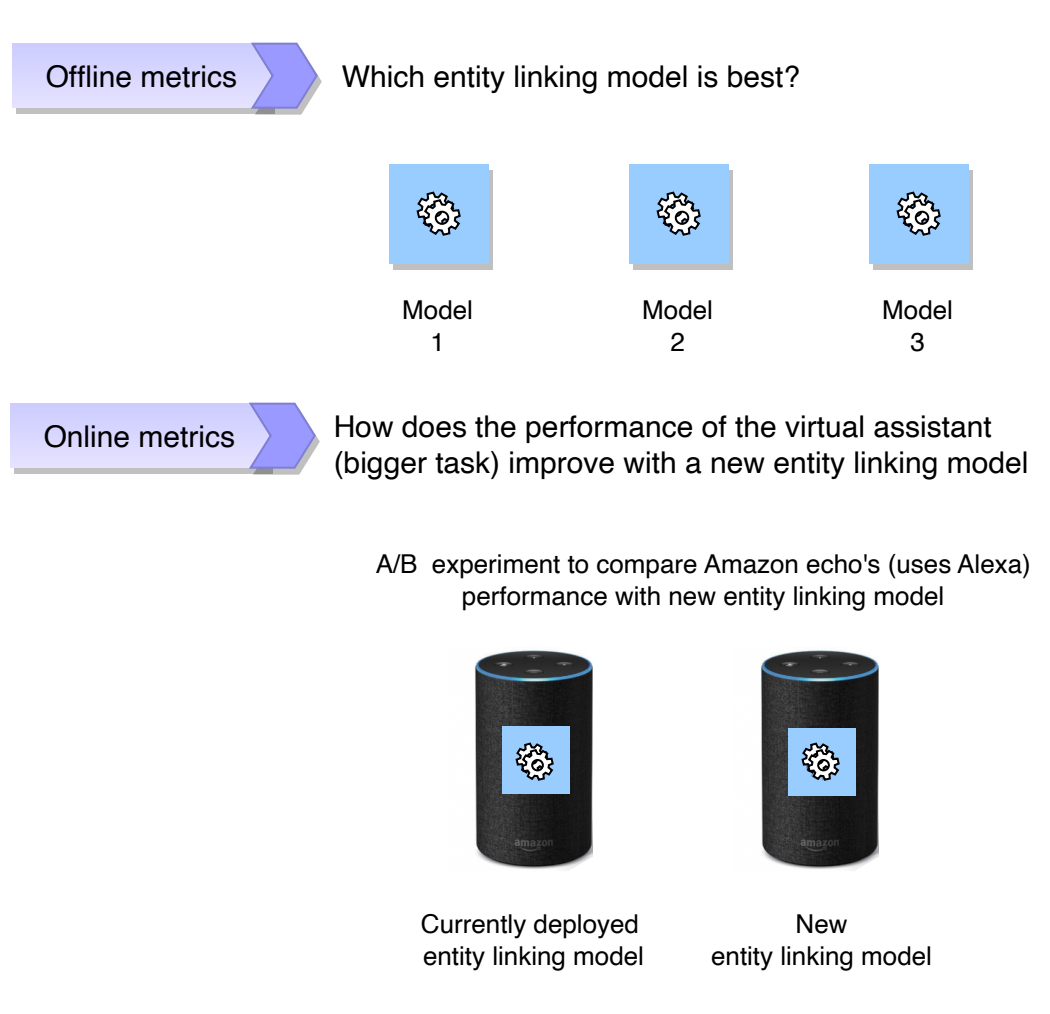
Metrics

Let's go over the metrics to evaluate the performance of the entity linking system.

We'll cover the following

In the previous lesson, we talked about various applications of *named entity linking* system and how it can be used as a component in bigger tasks/systems such as a virtual assistant system. Therefore, we require metrics that:

1. Compare different entity linking models based on their performance.
   This will be catered to by offline metrics.
2. Measure the performance of the bigger task when a particular model for entity linking is used.
   This will be catered to by online metrics.



Offline vs online metrics

> 📝 Offline metrics will be aimed at improving/measuring the performance of the entity linking component. Online metrics will be aimed at improving/measuring the performance of the larger system by using a certain entity linking model as its component.

Offline metrics

The named-entity linking component is made of two layers, as discussed previously:

1. Named entity recognition
2. Disambiguation

We will first look at offline metrics for each of the layers individually and will then discuss a good offline metric to measure the overall entity linking system.

Named entity recognition

For the first layer/component, i.e., the recognition layer, you want to extract all the entity mentions from a given sentence. We will continue with the previous sentence example, i.e., "Michael Jordan is the best professor at UC Berkeley".

It has two entity mentions:

1. Michael Jordan
2. UC Berkeley

NER should be able to detect both entities correctly. However, it may detect:

- Both correctly
- One correctly
- None correctly (wrongly detect non-entity as an entity)
- Correct entity but with the wrong type
- No entity, i.e., altogether miss the entities in the sentence

Possible NER predictions

	Sentence	Michael	Jordan	is	the	best	professor	at	UC Berkeley	
	Label	B-Person	I-Person	O	O	O	O	O	B-Organisation	
	Possible predictions by NER									
Both entities correctly detected	1.	B-Person	I-Person	O	O	O	O	O	B-Organisation	
	2.	O	O	O	O	O	O	O	B-Organisation	One entity correctly detected
	3.	B-Person	I-Person	O	O	O	O	O	O	One entity correctly detected
No entity detected	4.	O	O	O	O	O	O	O	O	
	5.	O	B-Organisation	O	O	O	B-Person	O	O	Wrong detection

Note: this diagram is made using IOB2 (aka BIO) tagging scheme for NER

Key
B: Beginning of entity
I: inner token of a multi-token entity
O: non-entity

Possible predictions by NER for given sentence

> 📝 You will call a recognition/detection of a named entity correct, only if it is an exact match of the entity in the labeled data. If NER only recognizes "Michael" as an entity and misses the "Jordan" part, it would be considered wrong. Moreover, if NER recognizes "Michael Jordan" as an entity but with the wrong type (say Organization), again, it would be considered wrong.

Given the above context on the correctness of the system, both precision and recall are important for measuring the performance of NER. They will be defined as:

$$\text{Precision} = \frac{no.\ of\ correctly\ recognized\ named\ entities}{no\ of\ total\ recognized\ named\ entities}$$

$$\text{Recall} = \frac{no.\ of\ correctly\ recognized\ named\ entities}{no\ of\ named\ entities\ in\ corpus}$$

You may have models that have excellent precision but poor recall. For instance, when such a model recognizes entities, they will be mostly correct. However, it may not recognize all the entities present in the sentence. Similarly, you could have models that have excellent recall but poor precision.

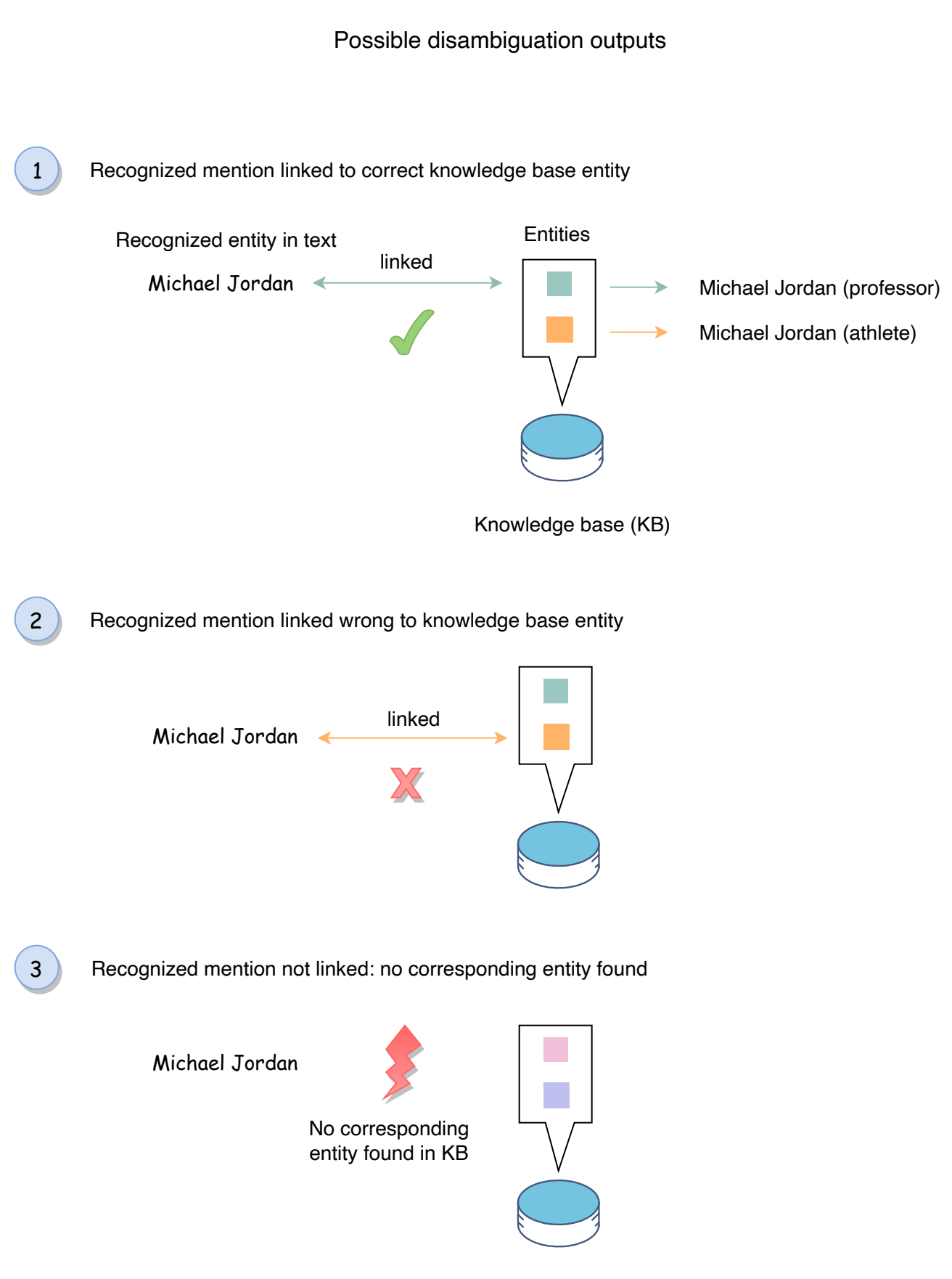
You want your model to have both high precision and high recall. To look at both, collectively, you will use the F1-score as the metric.

$$\text{F1-score} = 2 * \frac{precision * recall}{precision + recall}$$

Disambiguation

The disambiguation layer/component receives the recognized entity mentions in the text and links them to entities in the knowledge base. It might:

- Link the mention to the correct entity
- Link the mention to the wrong entity
- Not link the mention to any entity (in case it does not find any corresponding entity in the knowledge base)



Possible disambiguation outputs for recognized entity mentions in text

This component will perform linking for all entities recognized, and they will either be linked to an object in the knowledge base or not linked at all (i.e., the model predicts that it doesn't have a corresponding object in the knowledge base). So, the concept of recall doesn't really apply here as each entity is going to be linked(to either an object or Null). Therefore, it makes sense to only use precision as the metric for disambiguation.

$$\text{Precision} = \frac{no.\ of\ mentions\ correctly\ linked}{no.\ of\ total\ mentions}$$

Named-entity linking component

You have seen the metrics for two main components of the entity linking system. Now, let's devise a metric to measure the goodness provided by the entity linking component as a whole. You will use F1-score as the end-to-end metric. The definition of a true positive, true negative, false positive, and false negative, for the calculation of end-to-end F1-score (and precision and recall by extension), is as follows:

🔹 *True positive:* an entity has been correctly recognized and linked.

🔹 *True negative:* a non-entity has been correctly recognized as a non-entity or an entity that has no corresponding entity in the knowledge base is not linked.

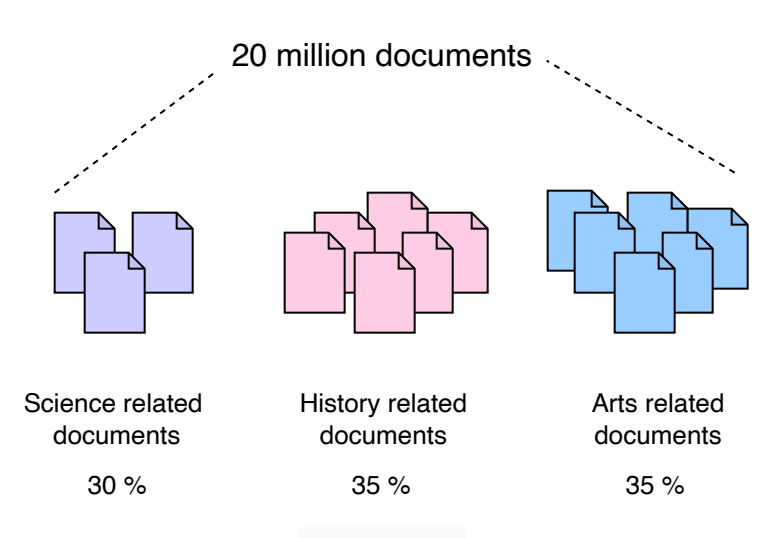
🔹 *False positive:* a non-entity has been wrongly recognized as an entity or an entity has been wrongly linked.

🔹 *False negative:* an entity is wrongly recognized as a non-entity, or an entity that has a corresponding entity in the knowledge base is not linked.

Micro vs. macro metrics

We can compute *micro-averaged* or *macro-averaged* versions of metrics, based on what is important for us in a particular situation.

Let's take an example of a corpus of documents to see scenarios where micro and macro metrics will have significantly different results. Let's assume that the corpus has one million documents that contain twenty million entities collectively. The documents can be categorized as science (30% of the corpus), arts (35% of the corpus), and history (35% of the corpus) related.



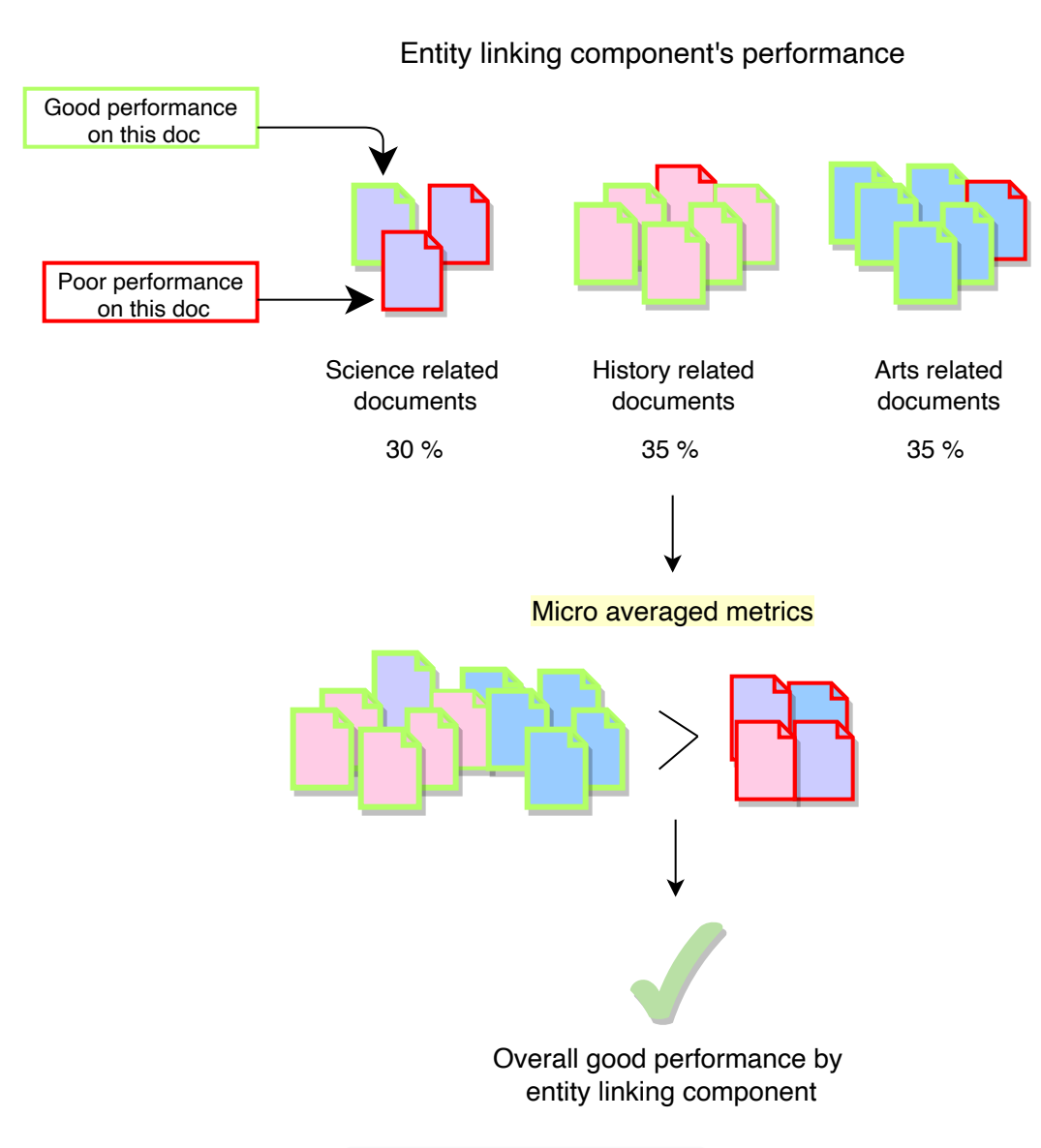
Corpus

> 📝 A macro-average computes the metric independently for each document and takes the average (giving equal weightage to all documents). In contrast, a micro-average aggregates the contributions of all documents to compute the average metric.

- Micro-averaged metrics

  Assume that you are only interested in how well you recognize these twenty million entities, without paying any attention to the performance across individual documents. Here, you will opt for micro-averaged F1-score and micro-averaged precision and recall, by extension.

  > 📝 We are focused on the overall performance of the entity linking component. We don't care if the performance is better for a certain set of documents and not good for another set, so we opt for micro-averaged metrics.



Micro-averaged metrics

Micro-averaged metrics are computed, as shown below.

$$\text{Micro-averaged precision} = \frac{\sum_{i=1}^n TP_i}{\sum_{i=1}^n TP_i + \sum_{i=1}^n FP_i}\ _{where\; n= no. \;of\; documents,\; TP_i= true \;positives\; in\; document \;'i' \;and\; FP_i= false\; positive\; in\; document \;'i'}$$

$$\text{Micro-averaged recall} = \frac{\sum_{i=1}^n TP_i}{\sum_{i=1}^n TP_i \;+\; \sum_{i=1}^n FN_i}\ _{where\; FN_i= false\; positive\; in\; document \;'i'}$$

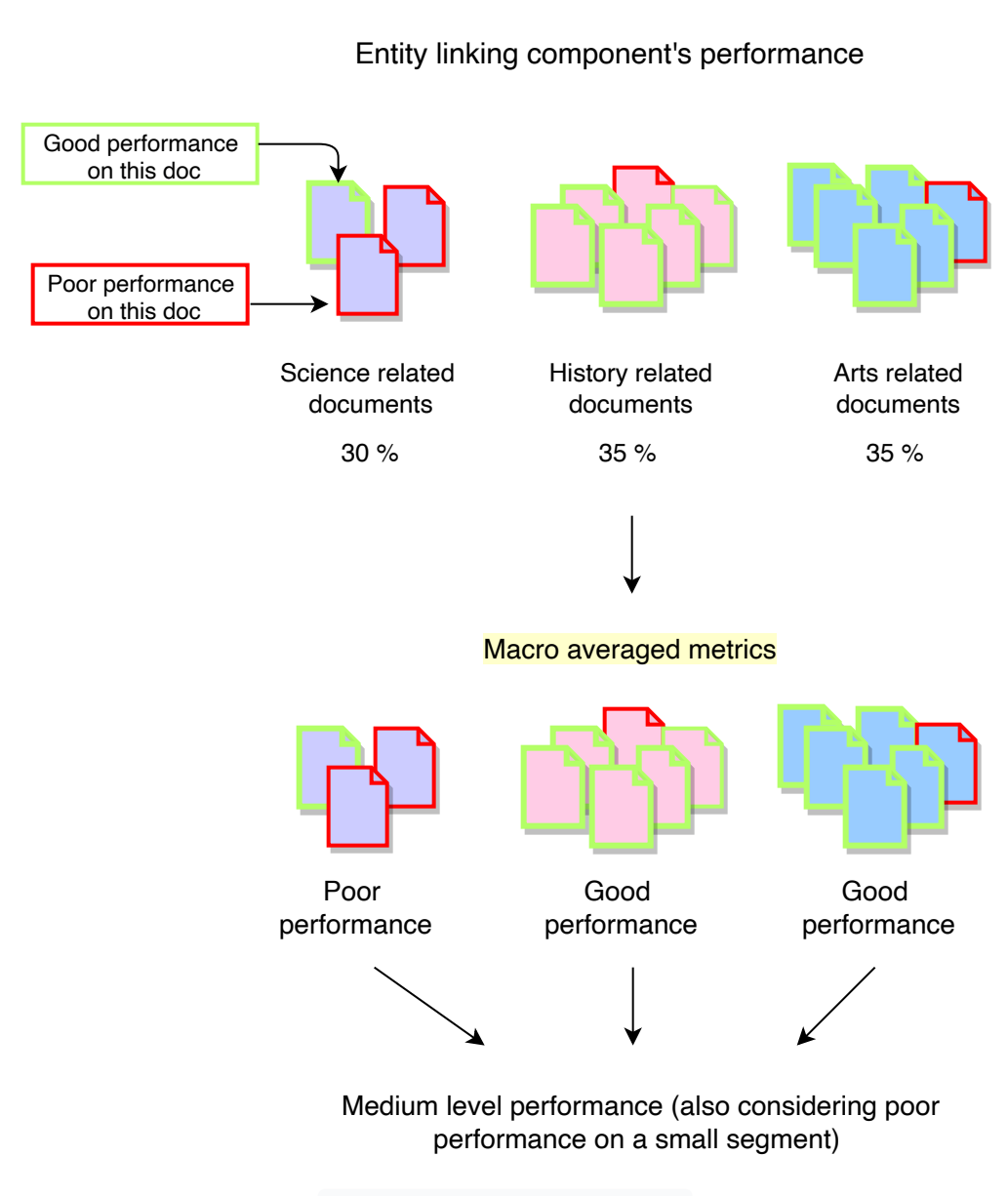
$$\text{Micro-averaged F1-score} = 2 * \frac{Micro-averaged\ precision * Micro-averaged\ recall}{Micro-averaged\ precision + Micro-averaged\ recall}$$

- Macro-averaged metrics

  When you are interested in the individual performance of entity linking across the different types of documents (e.g., science, history, and arts), you will shift to macro-averaged f1 score and macro averaged precision and recall, by extension.

  Let's look at a scenario where you would need to look at entity linking's performance across different types of documents. Assume that 30% of science-related documents contain only 10% of the entities. Now entity linking performs poorly for this small segment, but it performs well for the rest of the 70% documents. Macro averaged metrics allow you to calculate the metric for each set of documents and then take their average, thereby giving an equal chance of representation to the small segment of science-related documents.

  On the contrary, micro-averaged metrics would not pay attention to these small number of documents where performance is low and give a biased result.



Macro-averaged metrics

Macro-averaged metrics are computed, as shown below.

$$\text{Macro-averaged precision} = \frac{\sum_{i=1}^n P_{di}}{n}\ _{where\; n= no. \;of\; documents,\; P_{di}= precision\; over\; document\; 'i'}$$

$$\text{Macro-averaged recall} = \frac{\sum_{i=1}^n R_{di}}{n}\ _{where\; n= no. \;of\; documents,\; R_{di}= recall\; over\; document\; 'i'}$$

$$\text{Macro-averaged F1-score} = 2 * \frac{Macro-averaged\ precision * Macro-averaged\ recall}{Macro-averaged\ precision + Macro-averaged\ recall}$$

Online metrics

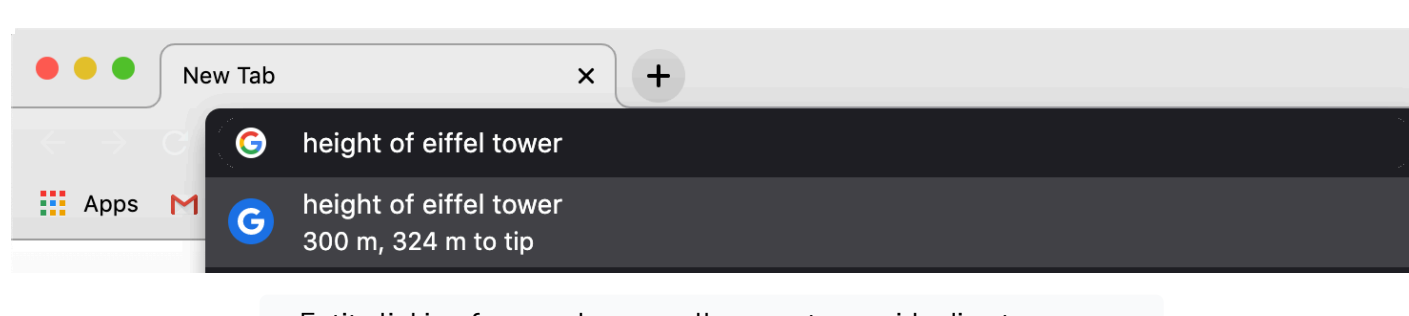
Once you have a model with good offline scores, you still need to see if the bigger task/system's performance will improve if you plug in your new model. So, you would do A/B experiments for these bigger systems to measure their performance with your new entity linking component. The metrics in these A/B tests would be devised based on the overall system, which, in turn, would indicate how well your new model for entity linking is performing as it gets integrated.

To get a better understanding, let's think about two such larger tasks/systems:

1. Search engines
2. Virtual assistants

Search engines

Semantic search allows us to directly answer the user's query by returning the entity or its properties that the user wants to know. The user no longer needs to open up search results and look for the information that is required. As shown in the example below, the user typed their query in the search bar, where entity linking recognized and linked the entity mention *"eiffel tower"*. This enables the system to fetch the entity's property "height" from the knowledge base and directly answer the user's query.



Entity linking for search query allows us to provide direct answers

You have seen how a search engine may use your entity linking component. Now, let's come up with the evaluation metric for the search engine.

For search engines, user satisfaction lies in the query being properly answered, which can be measured by *session success rate*, i.e., % of sessions with user intent satisfied. So, your online A/B experiment will measure the effect of your new entity linking systems on the session success rate.

Virtual assistants

Virtual assistants (VAs) helps perform tasks for a person based on commands or questions. For instance, a user asks Alexa, "What is the height of the Eiffel tower?". In order to answer this question, the VA needs to detect and link the entities in the user's question (the entity linking component is required here). In this case, the entity would be "Eiffel tower". Once this has been done, the VA can fetch the entity's property "height" and answer the user's question.

The evaluation metric for the VA would be user satisfaction (percentage of questions successfully answered). It could be measured in various ways by using implicit and explicit feedback. However, the key aspect here is that user satisfaction should improve by plugging in a better model for the entity linking component in the system.

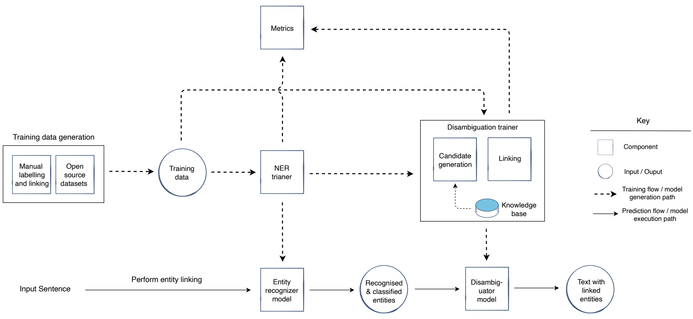
Architectural Components

Let's go over the architectural components of the entity linking unit.

We'll cover the following

The architectural components diagram for entity linking is shown below. It consists of two paths:

1. Model generation path (training flow)
2. Model execution path (prediction flow)



Architectural diagram for entity linking

> The terms *entity mention* and *recognized entity* are used interchangeably.

Model generation path

Model generation is responsible for training models for entity linking task. Let's look at the components of this path.

Training data generation

You will begin by gathering training data for entity linking through open-source datasets and manual labelling/linking of text. These methods will be detailed in the next lesson.

You will pass the training data to the named entity recognition (NER) model trainer and the named entity disambiguation (NED) model trainer.

NER

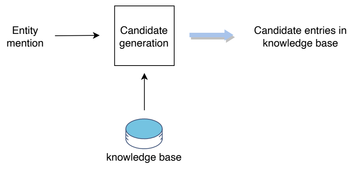
NER is responsible for building a machine learning model to recognize entities, such as a person, organization, etc., for a given input text.

NED

The disambiguation component will receive the output of the NER for linking. The disambiguation process has two phases:

1. Candidate generation

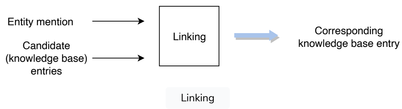
   The first phase of disambiguation is candidate generation. It finds potential matches for the entity mentions, by reducing the size of the knowledge base to a smaller subset of *candidate documents/entities*. This saves us from running the linking model on the entire knowledge base for each entity mention.



Candidate generation

2. Linking

   The second phase of disambiguation is linking. Here, you will select the exact corresponding entry in the knowledge base for each recognized entity. The linking model runs on only the candidate entries for each mention.



Linking

Metrics

As explained in the previous lesson, the metrics component will measure the performance of:

1. NER component separately
2. NED component separately
3. Entity linking as a whole

> The arrow going from disambiguation to the metrics component shows that the metrics component will not only measure the quality of the disambiguation but the quality of the whole entity linking system.

Model execution path

The model execution path is very straightforward. It begins with an input sentence that is fed to the NER component. NER identifies the entity mentions in the sentence, along with their types, and sends this information to the NED component. This component then links each entity mention to its corresponding entity in the knowledge base (if it exists). Like this, entity linking will be performed for a given piece of text.

Training Data Generation

Let's generate training data for the entity linking problem.

We'll cover the following

- Open-source datasets
- Human-labeled data

There are two approaches you can adopt to gather training data for the entity linking problem.

1. Open-source datasets
2. Manual labeling

You can use one or both depending on the particular task for which we have to perform entity linking.

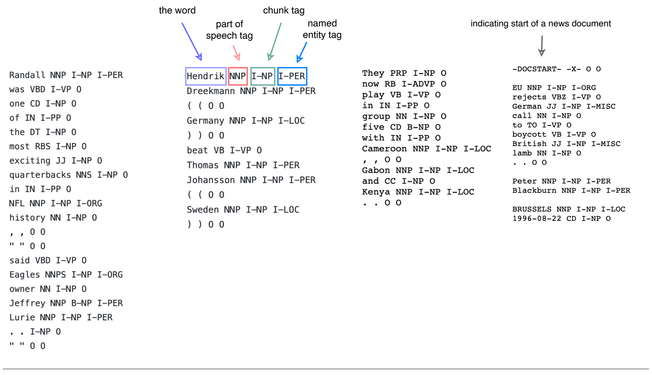
Open-source datasets

If the task is not extremely domain-specific and does not require very specific tags, you can avail open-source datasets as training data. For example, if you were asked to perform entity linking for a simple chatbot, you could utilize the general-purpose, open-source dataset CoNLL-2003 for *named-entity recognition*.

CoNLL-2003 is built on the Reuters Corpus which contains 10,788 news documents totalling 1.3 million words. It contains train and test files for English and German languages and follows the IOB tagging scheme.

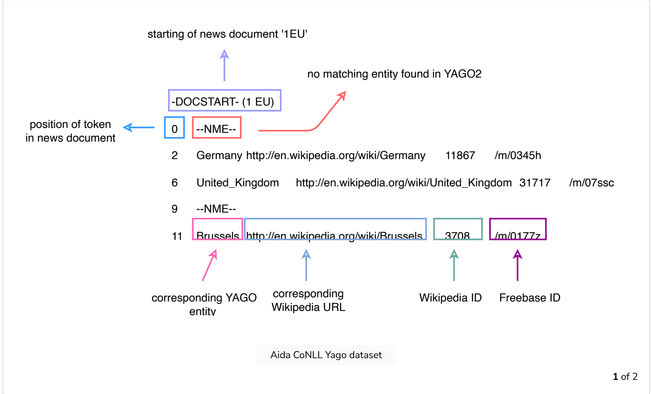
> IOB tagging scheme
> `I` - An inner token of a multi-token entity
> `O` - A non-entity token
> `B` - The first token of a multi-token entity; The B-tag is used only when a tag is followed by a tag of the same type without "O" tokens between them. For example, if for some reason the text has two consecutive locations (type LOC) that are not separated by a non-entity

The following are some snippets from the train and test files of CoNLL dataset.



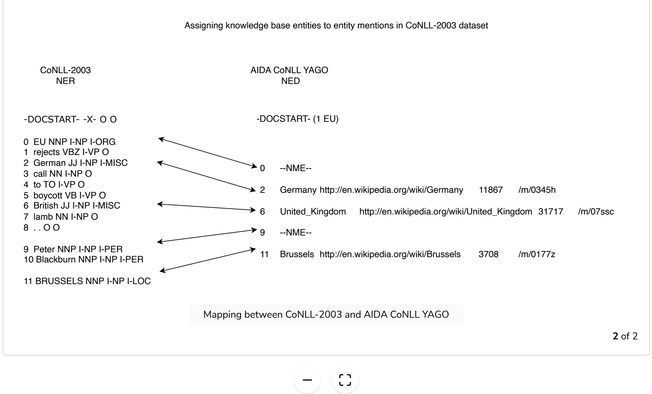
CoNLL-2003 for named entity recognition

For *named-entity disambiguation*, you can utilize the AIDA CoNLL-YAGO Dataset, which contains assignments of entities to the mentions of named entities annotated for the CoNLL-2003 dataset. The entity mentions are assigned to YAGO (database), Wikipedia and Freebase (database) entities as shown in the slide below.



Aida CoNLL Yago dataset

1 of 2



Mapping between CoNLL-2003 and AIDA CoNLL YAGO

2 of 2

Human-labeled data

Once you have utilized the open-source datasets, we may want to enhance the data and increase its size through manual labelers. The manual labelers will generate training data similar to the open-source datasets, by annotating named entities in text and linking them to corresponding entities in the knowledge base.

Another case where you would generate data through manual labelers is when you require a highly specialized dataset for a specific problem. For example, assume that the problem is related to the medical field; this requires identifying certain domain-specific entities. In such situations, you need to understand the domain in which you want to perform entity linking. What are the kind of entities you want to recognize and link? When the manual labelers are given hospital data, they will mark doctor names, symptoms, diseases, patient names, types of surgeries, and so on. Hence, you would have tags that are related to the domain of the task.

After labeling the entities the labelers will also link them to the entities in the knowledge base (database) that is being used.

Modeling

Let's look at the modeling options for entity 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:

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.

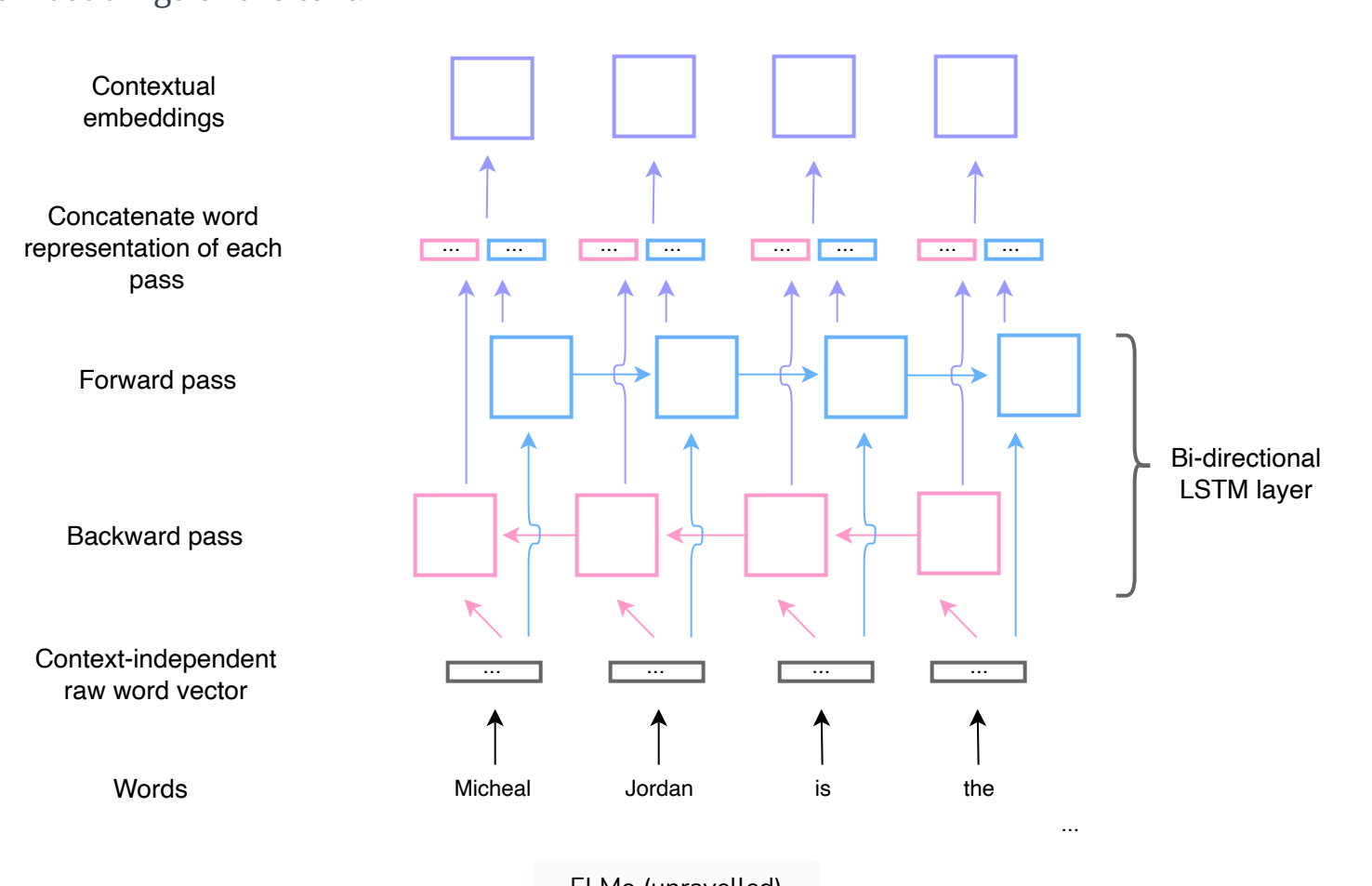
Two popular model architectures that generate term contextual embeddings are:

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**.

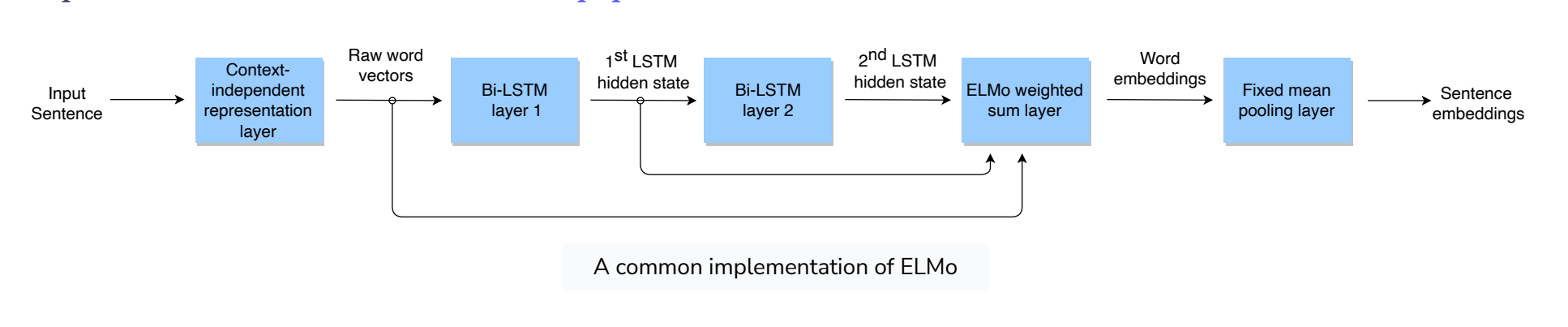
The forward pass sequentially goes over the input sentence from left to right. It looks at the context (words) **before** the active word. Whereas, the backward pass sequentially goes over the input sentence from right to left. It looks at the context (words) **after** the active word to predict the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



ELMo (unravelled)

> 📝 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.

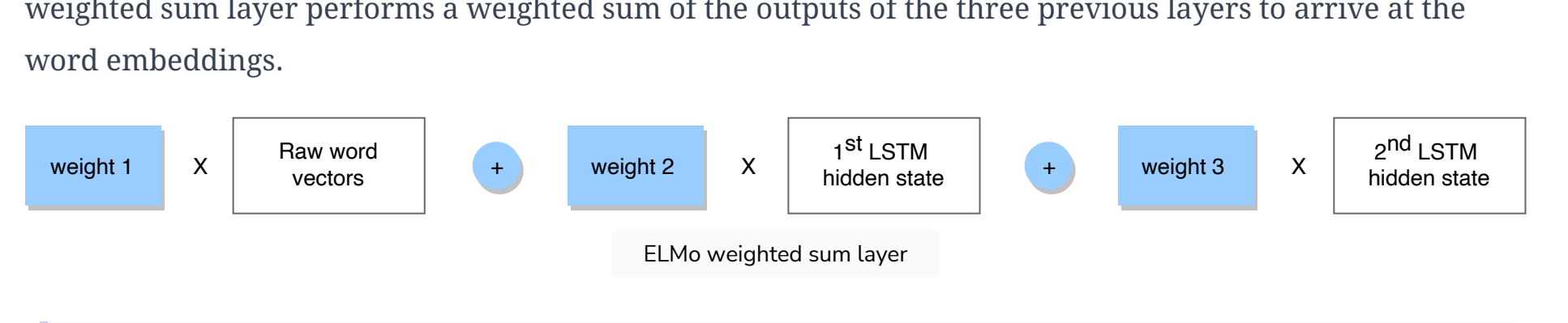
ELMo can be used to obtain *both word embeddings and sentence embeddings*. Below is a common implementation of ELMo, based on this paper.



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 (unravelled)", 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:

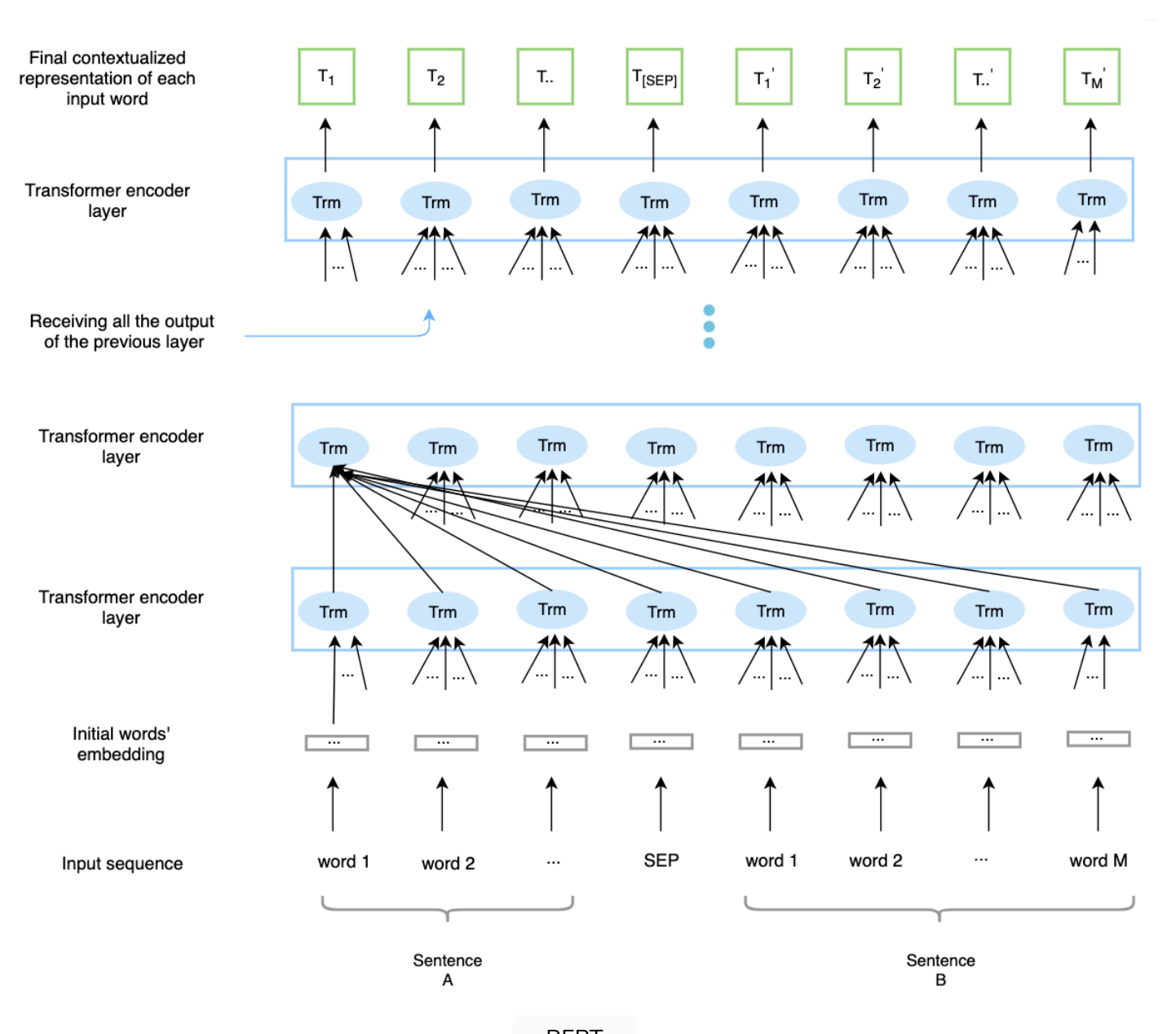
- Left to right to train the forward LSTM
- Right to left to train the backward LSTM

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

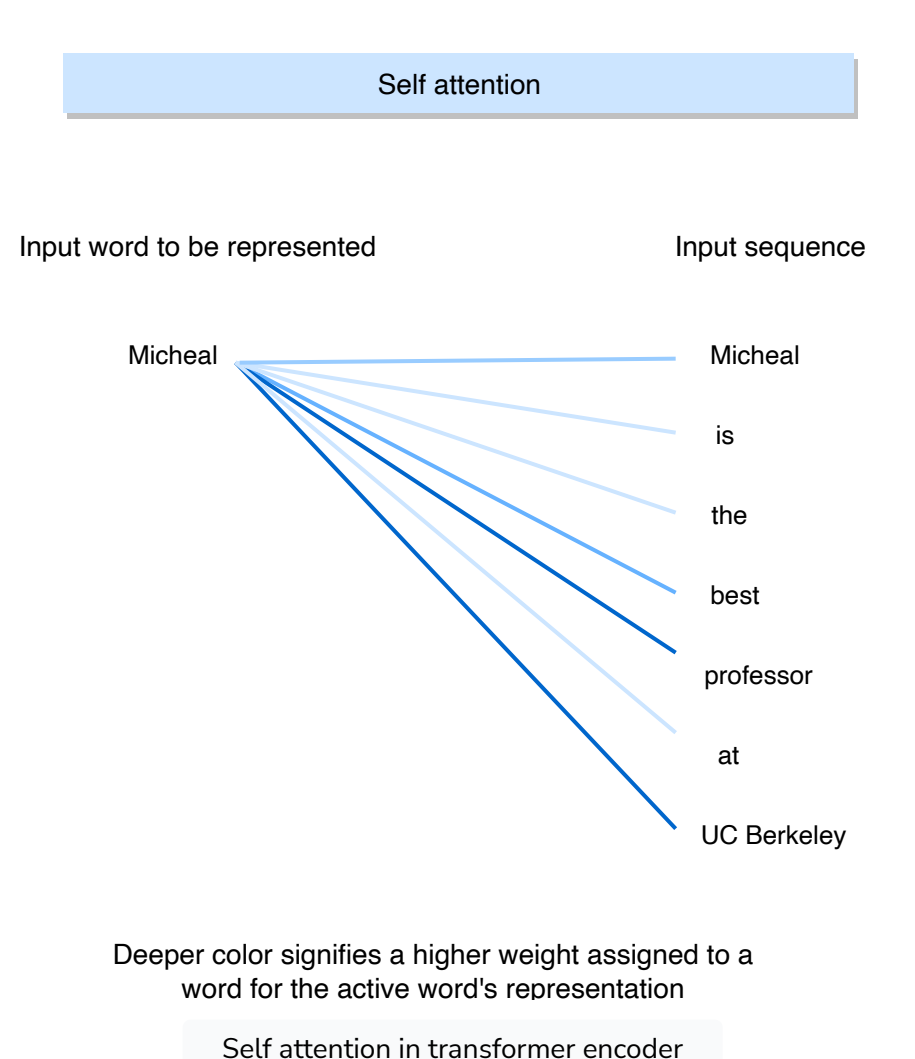
Let's see how BERT (bidirectional encoder representations from transformers) gives us contextual word embeddings.

BERT starts by taking in the input sequence which can be made up of multiple sentences separated by the *SEP* tag. Each word is converted into 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.



BERT

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.



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.

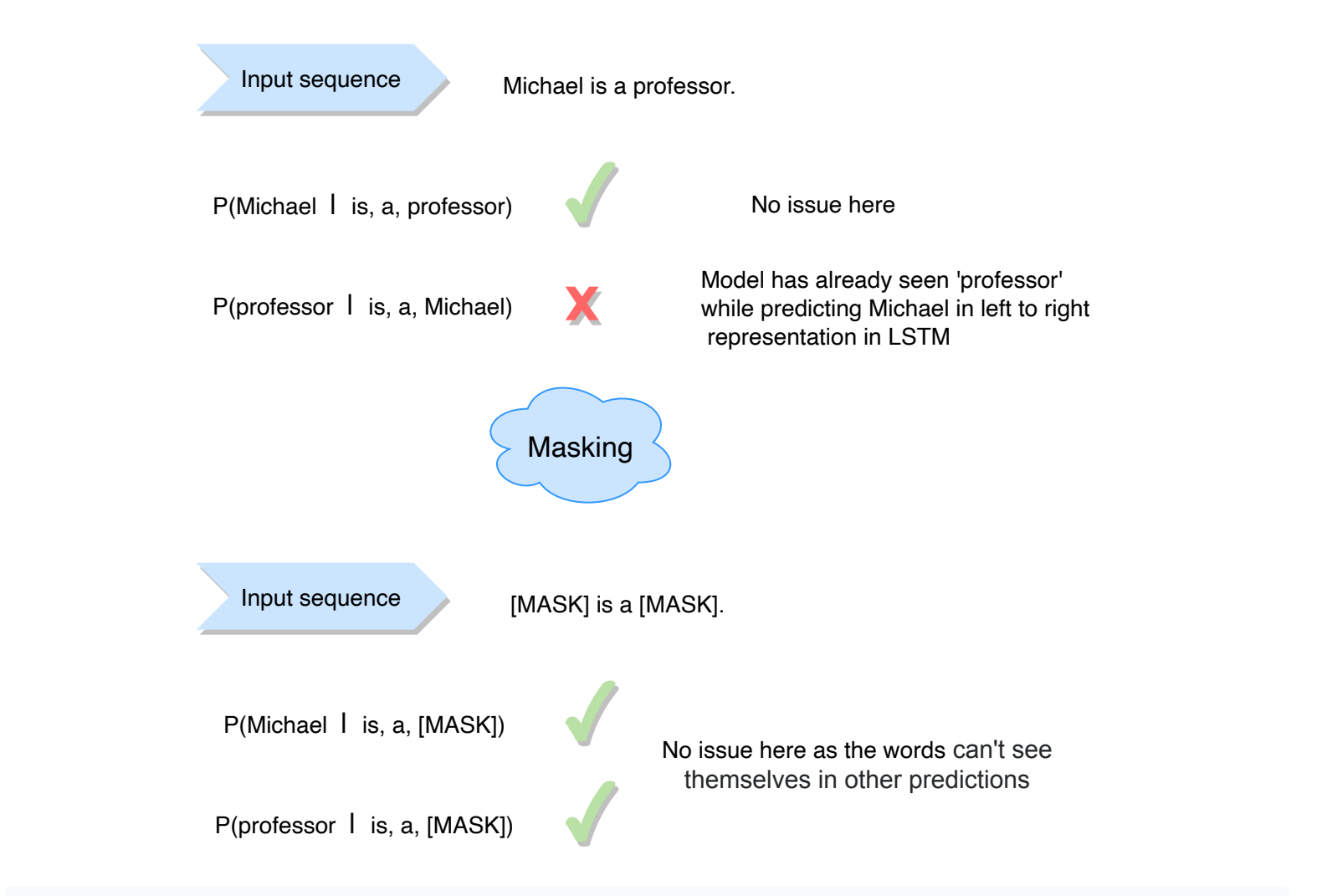
The following two prediction objectives are used during the training of the model for learning contextualized term representation:

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.

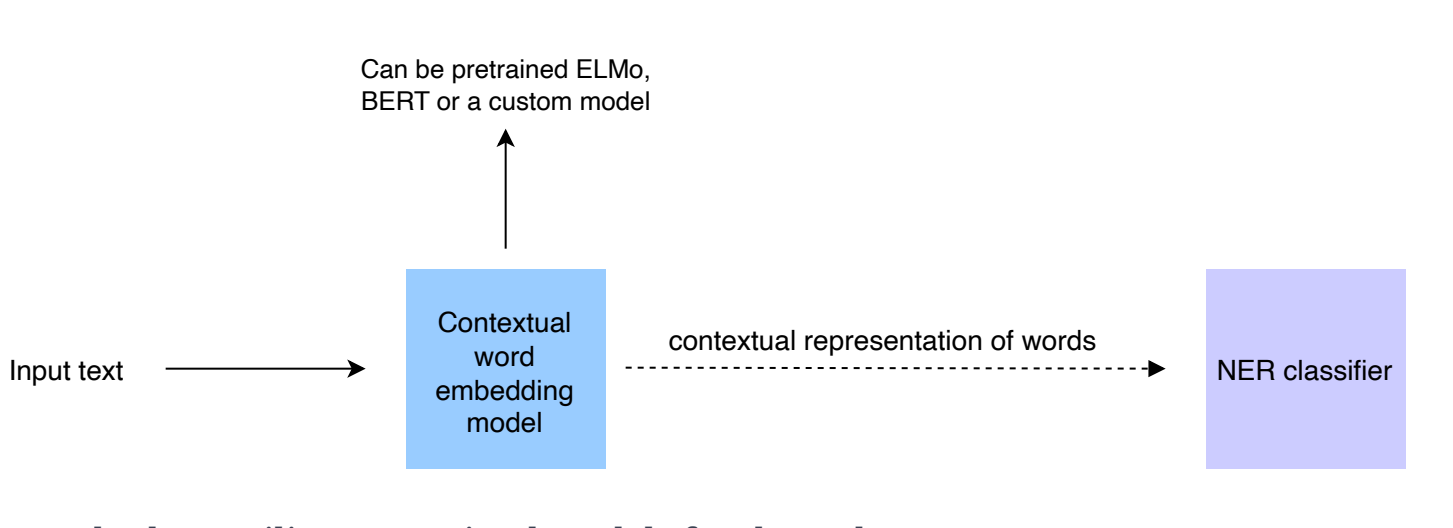
Variations

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.
DistilBERT	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 *DistilBERT*. 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.**
Cased vs uncased BERT	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).

NER modelling

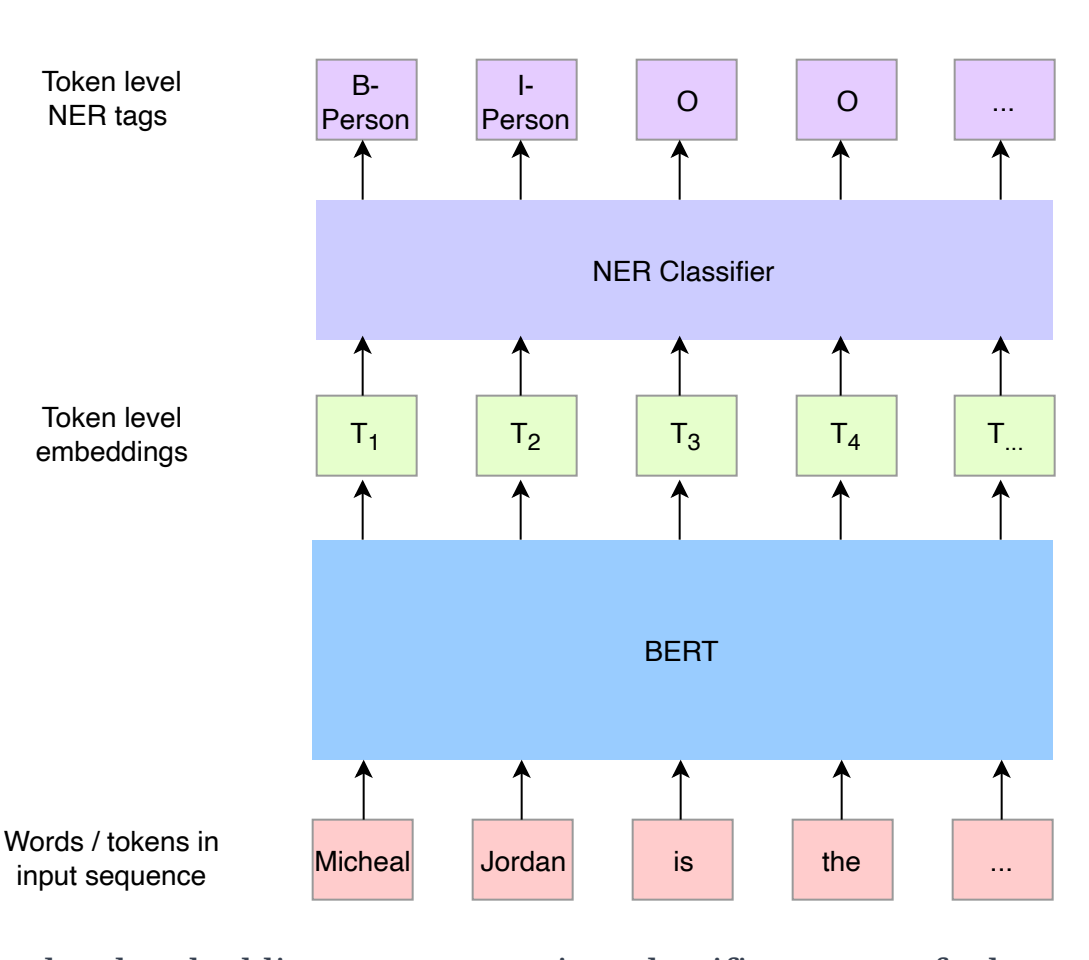
Both ELMo and BERT are trained on massive datasets and have the ability to understand language. With these two models generating contextual word embeddings, half of the work is done. Through the transfer learning approach, you can now utilize these embeddings in a NER classifier.



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.



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.

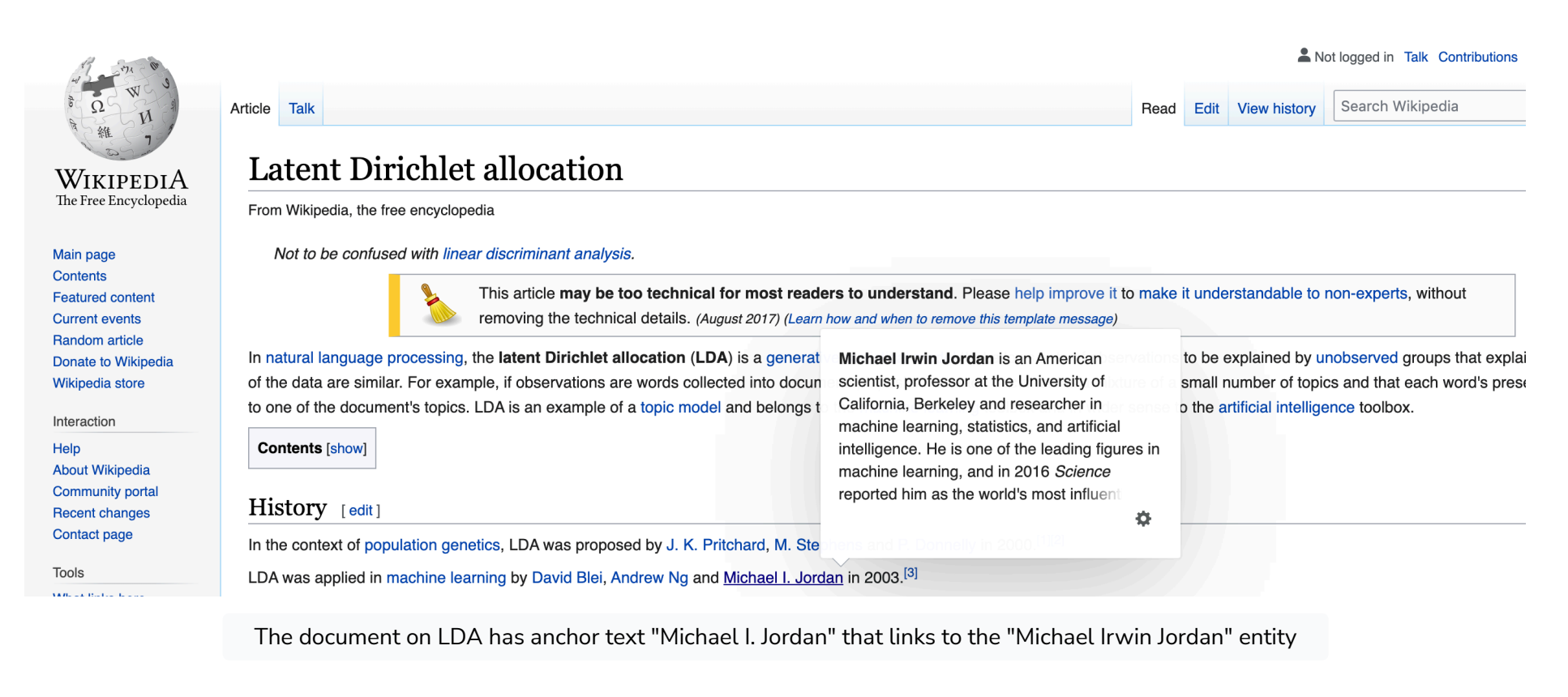
Candidate generation

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.
2. You can also make use of referrals and anchor text in the knowledge-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 on the terms in 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 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 linked to an 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.

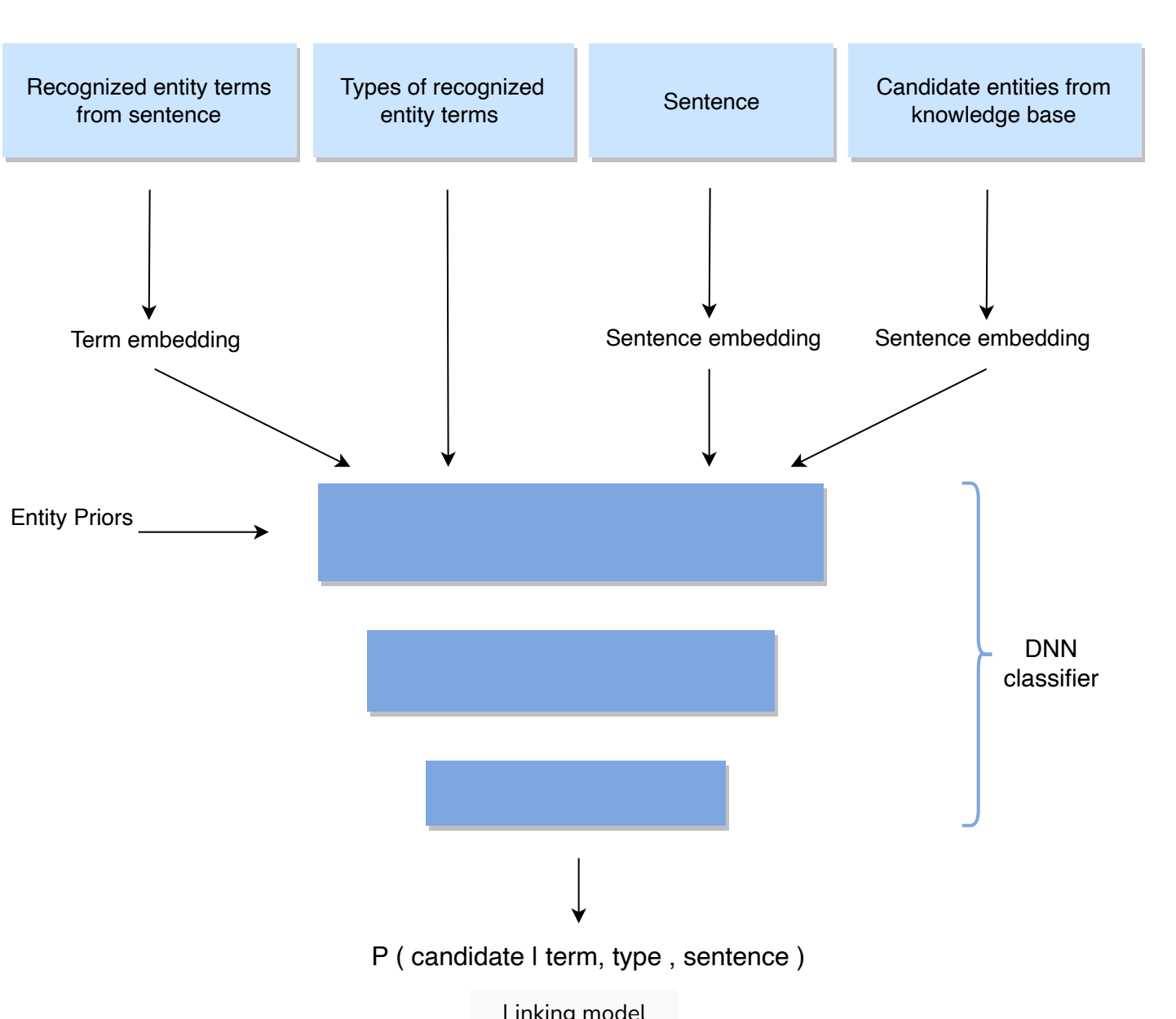
In the linking stage, you will build a model that will give us the probability of a candidate being the true match for a recognized entity. You will select the candidate with the highest probability and link it to the recognized entity mention.

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.



Linking model

> 📝 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 can have the 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.

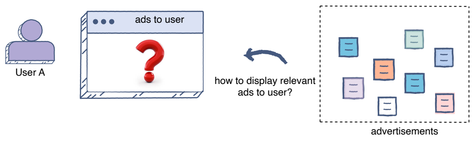
Problem Statement

Let's look at the problem statement for the ad prediction system.

We'll cover the following

Problem statement

The interviewer has asked you to build a system to show relevant ads to users.



How to display the most relevant ads to the user?

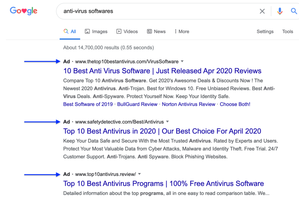
Visualizing the problem

There are two well-known advertising platforms Google and Facebook, that run advertisements paid for by businesses.

Have you ever wondered how Google and Facebook drive new (or existing) customers who are searching for the product or service that you provide to your website using advertisements?

When you type a search query in a google.com search bar, ads appear in the search results.

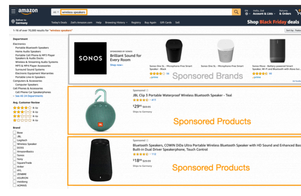
Google's search network allows advertisers to show their business advertisements to users who are actively looking for products or services that the advertiser provides.



Within the Search Network, Google matches a keyword that is relevant to a product or business advertiser, so that when a user issues a query, it triggers an ad that shows up on the search page for a user to click on.

Unlike Google, Facebook ads are mostly shown on users' news feed based on users' interests.

Amazon.com also shows ads, generally in their search result page based on query context as shown in the picture.



Interview questions

The interviewer can ask the following questions about this problem, narrowing the scope of the question each time.

- How would you build an ML system to predict the probability of engagement for Ads?
- How would you build an Ads relevance system for a search engine?
- How would you build an Ads relevance system for a social network?

> 📝 Note that the context can be different depending on the type of application in which we are displaying the advertisement.
>
> There are two categories of applications:
>
> **Search engine:** Here, the query will be part of the context along with the searcher. The system will display ads based on the search query issued by the searcher.
>
> **Social media platforms:** Here, we do not have a query, but the user information (such as location, demographics, and interests hierarchy) will be part of the context and used to select ads. The system will automatically detect user interest based on the user's historical interactions (using machine learning algorithms) and display ads accordingly.
>
> 📝 Most components will be the same for the above two discussed platforms with the main difference being the *context* that is used to select and predict ad engagement.

Let's set up the machine learning problem:

> *"Predict the probability of engagement of an ad for a given user and context(query, device, etc.)"*

← Back
Modeling

Next →
Metrics

Metrics

Let's look at the online and offline metrics used to judge the performance of an ad's prediction system.

We'll cover the following

The metrics used in our ad prediction system will help select the best machine-learned models to show relevant ads to the user. They should also ensure that these models help the overall improvement of the platform, increase revenue, and provide value for the advertisers.

Like any other optimization problem, there are two types of metrics to measure the effectiveness of our ad prediction system:

1. Offline metrics
2. Online metrics

> Why are both online and offline metrics important?
>
> Offline metrics are mainly used to compare the models offline quickly and see which one gives the best result. Online metrics are used to validate the model for an end-to-end system to see how the revenue and engagement rate improve before making the final decision to launch the model.

Offline metrics

As we build models, the best way to compare them is to measure prediction accuracy instead of measuring revenue impact directly. The following are a few metrics that enable us to compare the two models better offline.

Log Loss

Let's first go over the *area under the receiver operator curve* (AUC), which is a commonly used metric for model comparison in binary classification tasks. However, given that the system needs well-calibrated prediction scores, AUC, has the following shortcomings in this ad prediction scenario.

1. AUC does not penalize for "how far off" predicted score is from the actual label. For example, let's take two positive examples (i.e., with actual label 1) that have the predicted scores of 0.51 and 0.7 at threshold 0.5. These scores will contribute equally to our loss even though one is much closer to our predicted value.
2. AUC is insensitive to well-calibrated probabilities.

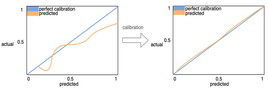
Minimize

Calibration measures the ratio of average predicted rate and average empirical rate. In other words, it is the ratio of the number of expected actions to the number of actually observed actions.

$$Calibration = \frac{predicted\ rate}{actual\ historically\ observed\ rate}$$

Why do we need calibration?

When we have a significant class imbalance, i.e., the distribution is skewed towards positive and negative class, we calibrate our model to estimate the likelihood of a data point belonging to a class.



calibration makes model predicted score closer to the actual score

Since, in our case, we need the model's predicted score to be well-calibrated to use in Auction, we need a calibration-sensitive metric. Log loss should be able to capture this effectively as Log loss (or more precisely cross-entropy loss) is the measure of our predictive error.

This metric captures to what degree expected probabilities diverge from class labels. As such, it is an absolute measure of quality, which accounts for generating well-calibrated, probabilistic output.

Let's consider a scenario that differentiates why log loss gives a better output compared to AUC. If we multiply all the predicted scores by a factor of 2 and our average prediction rate is double than the empirical rate, AUC won't change but log loss will go down.

In our case, it's a binary classification task; a user engages with ad or not. We use a 0 class label for no-engagement (with an ad) and 1 class label for engagement (with an ad). The formula equals:

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \log p_i + (1-y_i)\log(1-p_i)].$$

Here:

- N is the number of observations
- y is a binary indicator (0 or 1) of whether the class label is the correct classification for observation
- p is the model's predicted probability that observation is of class (0 or 1)

Online metrics

For online systems or experiments, the following are good metrics to track:

Overall revenue

This captures the overall revenue generated by the system for the cohorts of the user in either an experiment or, more generally, to measure the overall performance of the system. It's important to call out that just measuring revenue is a very short term approach, as we may not provide enough value to advertisers and they will move away from the system. However, revenue is definitely one critical metric to track. We will discuss other essential metrics to track shortly.

Revenue is basically computed as the sum of the winning bid value (as selected by auction) when the predicted event happens, e.g., if the bid amounts to $0.5 and the user clicks on the ad, the advertiser will be charged $0.5. The business won't be charged if the user doesn't click on the ad.

Overall ads engagement rate

Engagement rate measures the overall action rate, which is selected by the advertiser.

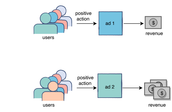
Some of the actions might be:

1. Click rate

This will measure the ratio of user clicks to ads.

2. Downstream action rate

This will measure the action rate of a particular action targeted by the advertiser e.g. add to cart rate, purchase rate, message rate etc.



More positive user engagement on the ad results in more revenue

Counter metrics

It's important to track counter metrics to see if the ads are negatively impacting the platform.

We want the users to keep showing engagement with the platform and ads should not hinder that interest. That is why it's important to measure the impact of ads on the overall platform as well as direct negative feedback provided by users. There is a risk that users can leave the platform if ads degrade the experience significantly.

So, for online ads experiments, we should track key platform metrics, e.g., for search engines, is session success going down significantly because of ads? Are the average queries per user impacted? Are the number of returning users on the platform impacted? These are a few important metrics to track to see if there is a significant negative impact on the platform.

Along with top metrics, it's important to track direct negative feedback by the user on the ad such as providing following feedback on the ad:

1. Hide ad
2. Never see this ad
3. Report ad as inappropriate

These negative sentiments can lead to the perceived notion of the product as negative.

The user clicks on the ad gives a positive impression of the product

1 of 2

The user reporting the ad gives a negative impression of the product

2 of 2

All of the metrics discussed above can be used to measure user engagement with ad and advertiser revenue on user engagement.

Architectural Components

Let's see the architectural components of the Ads prediction system.

We'll cover the following

Architecture

Let's have a look at the high-level architecture of the system. There will be two main actors involved in our ad prediction system - platform users and advertiser. Let's see how they fit in the architecture:

1. Advertiser flow

Advertisers create ads containing their content as well as targeting, i.e., scenarios in which they want to trigger their ads. A few examples are:

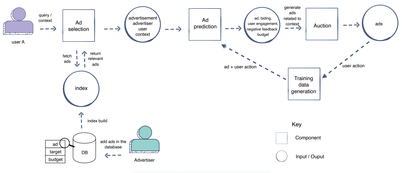
- *Query-based targeting*: This method shows ads to the user based on the query terms. The query terms can be a partial match, full match, expansion, etc.
  For example, whenever a user types a query "machine learning course", the system shows the ML course on educative.io.
- *User-based targeting*: The ads will be subjective to the user based on a specific region, demographic, gender, age, etc.
- *Interest-based targeting*: This method shows interest-based ads. Assume that on Facebook, the advertiser might want to show ads based on certain interest hierarchies. For example, the advertiser might like to show sports-related ads to people interested in sports.
- *Set-based targeting*: This type shows ads to a set of users selected by the advertisers. For example, showing an ad to people who were previous buyers or have spent more than ten minutes on the website. Here, we can expand our set and do seed audience expansion.

2. User flow

As the platform user queries the system, it will look for all the potential ads that can be shown to this user based on different targeting criteria used by the advertiser.

So, the flow of information will have two major steps as described below:

- Advertisers create ads providing targeting information, and the ads are stored in the ads index.
- When a user queries the platform, ads can be selected from the index based on their information (e.g., demographics, interests, etc.) and run through our ads prediction system.



Architectural diagram of ad prediction system

Let's briefly look at each component here. Further explanation will be provided in the following lessons.

Ad selection

The ad selection component will fetch the top k ads based on relevance (subject to the user context) and bid from the ads index.

Ad prediction

The ad prediction component will predict user engagement with the ad (the probability that an action will be taken on the ad if it is shown), given the ad, advertiser, user, and context. Then, it will rank ads based on relevance score and bid.

Auction

The auction mechanism then determines whether these top K relevant ads are shown to the user, the order in which they are shown, and the price the advertisers pay if an action is taken on the ad.

For every ad request, an auction takes place to determine which ads to show. The top relevant ads selected by the ad prediction system are given as input to Auction. Auction then looks at total value based on an ad's bid as well as its relevance score. An ad with the highest total value is the winner of the auction. The total value depends on the following factors:

- **Bid**: The bid an advertiser places for that ad. In other words, the amount the advertiser is willing to pay for a given action such as click or purchase.
- **User engagement rate**: An estimate of user engagement with the ad.
- **Ad quality score**: An assessment of the quality of the ad by taking into account feedback from people viewing or hiding the ad.
- **Budget**: The advertiser's budget for an ad

The estimated user engagement and ad quality rates combined results in the ad relevance score. They can be combined based on different weights as selected by the platform, e.g., if it's important to keep positive feedback high, the ad quality rate will get a higher weight.

The rank order is calculated based on predicted ad score (from the ad prediction component) multipled by the bid. Let's call this the ad rank score.

$$Ad\ rank\ score = Ad\ predicted\ score * bid$$

Ads with the highest ad rank score wins the auction and are shown to the user. Once an ad wins the auction, the cost per engagement (CPE) or cost per click (CPC) will depend on its ad rank score and ad rank score of the ad right below it in rank order, i.e.,

$$CPE = \frac{Ad\ rank\ of\ ad\ below}{Ad\ rank\ score} + 0.01$$

A general principle is that the ad will cost the minimal price that still allows it to win the auction.

Pacing

Pacing an ad means evenly spending the ad budget over the selected time period rather than trying to spend all of it at the start of the campaign.

Remember that whenever the user shows engagement with an ad, the advertiser gets charged the bid amount of the next ad. If the ad rank score is high, the ad set can spend the whole budget at the start of a time period (like the start of a new day, as advertisers generally have daily budgets). This would result in a high cost per click (CPC) when the ad load (the user engagement rate) is high.

Pacing overcomes this by dynamically changing the bid such that the ad set is evenly spread out throughout the day and the advertiser gets maximum return on investment(ROI) on their campaign. This also prevents the overall system from having a significantly high load at the start of the day, and the budget is spent evenly throughout the campaign.

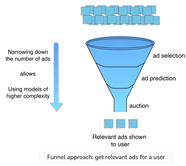
Training data generation

We need to record the action taken on an ad. This component takes user action on the ads (displayed after the auction) and generates positive and negative training examples for the ad prediction component.

Funnel model approach

For a large scale ads prediction system, it's important to quickly select an ad for a user based on either the search query and/or user interests. The scale can be large both in terms of the number of ads in the system and the number of users on the platform. So, it's important to design the system in a way that it can scale well and be extremely performance efficient without compromising on ads quality.

To achieve the above objective, it would make sense to use a funnel approach, we gradually move from a large set of ads to a more precise set for the next step in the funnel.



Funnel approach: get relevant ads for a user

As we go down the funnel (as shown in the diagram), the complexity of the models becomes higher and the set of ads that they run on becomes smaller. It's also important to note that the initial layers are mostly responsible for ads selection. On the other hand, ads prediction is responsible for predicting a well-calibrated engagement and quality score for ads. This predicted score is going to be utilized in the auction as well.

Let's go over an example to see how these components will interact for the search scenario.

- A thirty-year old male user issues a query "machine learning".
- The **ads selection** component selects all the ads that match the targeting criteria (user demographics and query) and uses a simple model to predict the ad's relevance score.
- The **ads selection** component ranks the ads according to r, where $r = bid * relevance$ and sends the top ads to our ads prediction system.
- The **ads prediction** component will go over the selected ads and uses a highly optimized ML model to predict a precise calibrated score.
- The **ads auction** component then runs the auction algorithm based on the bid and predicted ads score to select the top most relevant ads that are shown to the user.

We will zoom into the different segments of this diagram as we explore each segment in the upcoming lessons.

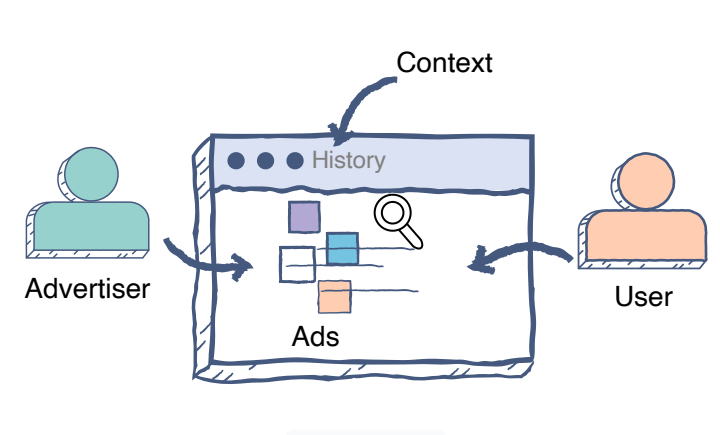
Feature Engineering

Let's engineer some features for the prediction model.

We'll cover the following

Features are the backbone of any learning system. Let's think about the main actors or dimensions that will play a key role in our feature engineering process.

1. Ad
2. Advertiser
3. User
4. Context



Actors

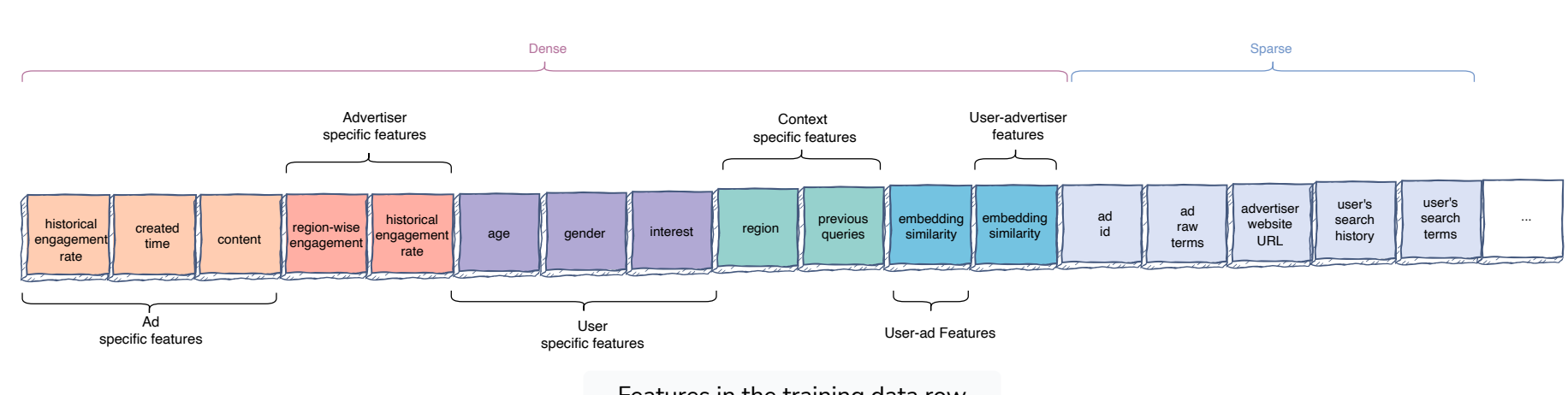
📝 Context refers to the engagement history, user interests, current location, time and date, as discussed in the previous lessons.

Features for the model

Now it's time to generate features based on these actors. The features would fall into the following categories:

1. Ad specific features
2. Advertiser specific features
3. User specific features
4. Context specific features
5. User-ad cross features
6. User-advertiser cross features

A subset of the features is shown below:



Features in the training data row

Ad specific features

- **ad_id**

  A unique id is assigned to each ad and can be used as a sparse feature. Utilizing ad_id as a sparse feature allows the model to memorize historical engagement for each ad, and it can also be used in interesting cross features for memorization (such as ad_id * user interests). Additionally, we can also generate embeddings during training time for the ad using its id, as we will discuss in the ad prediction section.

- **ad_content_raw_terms**

  Ad terms can also be very useful sparse features. They can tell us a lot about the ad, e.g., a good model can learn from the text's content to identify what the ad is about, such as politics or sports. Raw terms allow the models (especially NN models) to learn such behavior from given raw terms.

- **historical_engagement_rate**

  This feature specifies the rate of user engagement with the ad. Here we will measure engagement in different windows such as different times of the day or days of the week. For instance, we can have the following features:

  - **ad_engagement_history_last_24_hrs**

    Since ads are short-lived, recent engagement is important. This feature captures the most recent engagement with the ad.

  - **ad_engagement_history_last_7_days**

    This captures the activity on the ad on each day of the week. For example, an ad can get more engagement on weekends rather than on weekdays.

  📝 This feature can tell the model that a particular ad is performing well. This prior engagement data can help predict future engagement behavior.

- **ad_impression**

  This feature records the number of times an ad is viewed. This is helpful since we can train the model on the engagement rate as a feature when we have a reasonable number of ad impressions. We can select the cut-off point until which we want to consider the impressions. For example, we can say that if a particular ad has twenty impressions, we can measure the historical engagement rate.

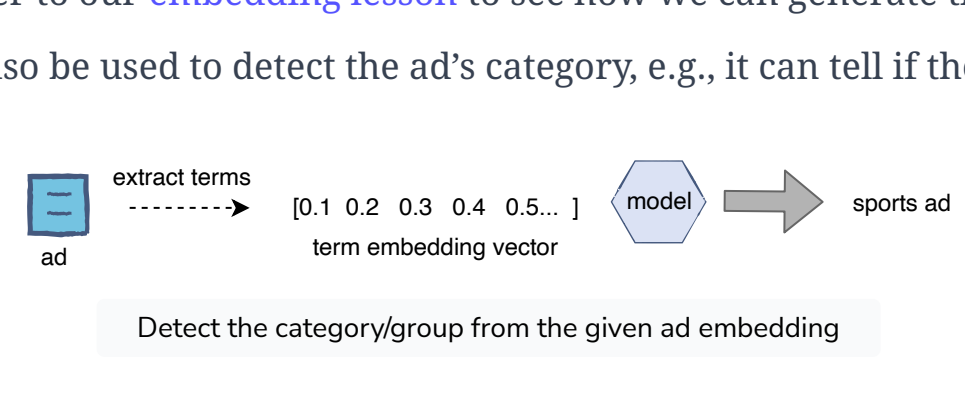
- **ad_negative_engagement_rate**

  This feature keeps a record of negative engagement (hide, report) with the ad.

- **ad_embedding**

  We can generate embedding for an ad given all the data that we know about it e.g. ad terms and engagement data. This embedding is then a dense representation of the ad that can be used in modeling. Please refer to our embedding lesson to see how we can generate this embedding.

  Ad embedding can also be used to detect the ad's category, e.g., it can tell if the ad belongs to sports, etc.



Detect the category/group from the given ad embedding

- **ad_age**

  This feature specifies how old the ad is.

- **ad_bid**

  We can record the bid for the ad specified by the advertiser.

Advertiser specific features

- **advertiser_domain**

  This is a sparse feature that keeps a record of the domain name for an advertiser. This can be used in the same way for memorization and embedding generation as discussed for ad_id.

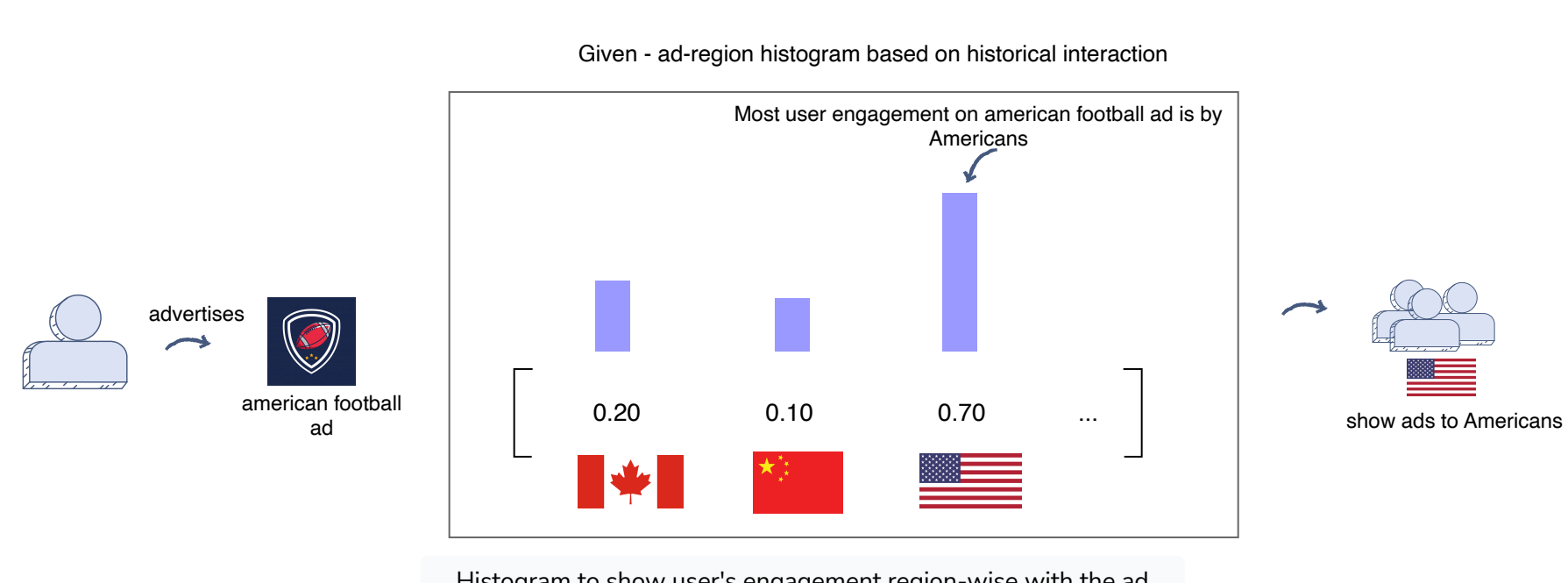
- **historical_engagement_rate**

  This feature specifies the ratio of user engagement with ads posted by a particular advertiser.

- **region_wise_engagement**

  The system should learn to show ads specific to a region from the histogram of engagement based on region. From an advertiser's perspective, the ad posted by an advertiser can be restricted to a specific region based on the ad content.

  For example, an American football ad posted by the advertiser will be most relevant to people living in the United States. This relevance can be predicted using the given histogram of engagement between the users and the ad.



Histogram to show user's engagement region-wise with the ad

User specific features

- **user_previous_search_terms**

  This sparse feature specifies what users have searched in the past. This helps in recommending ads based on past user preferences.

- **user_search_terms**

  This sparse feature keeps a record of the user's search query terms.

- **age**

  This feature records the age of the user. It allows the model to learn the kind of ad that is appropriate according to different age groups.

- **gender**

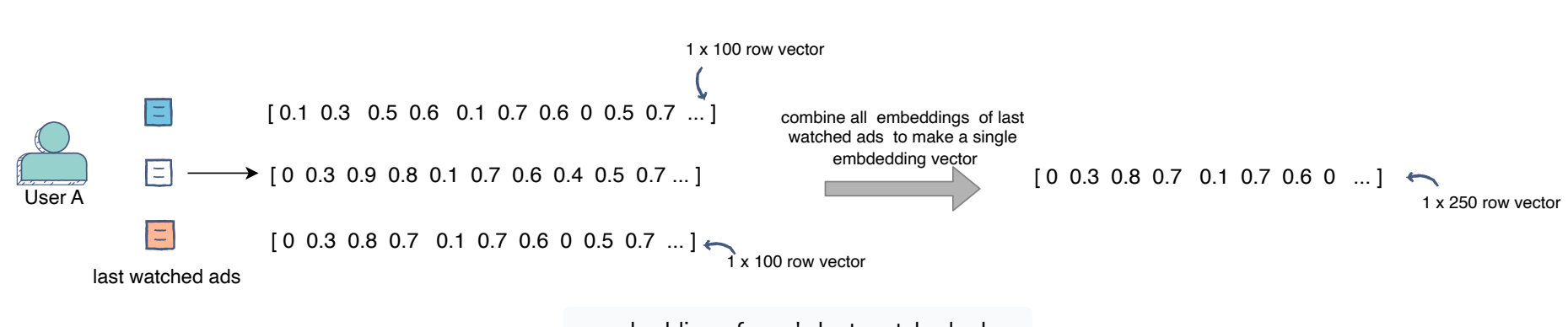
  The model learns about gender-based preferences.

- **language**

  This feature records the language of the user.

- **embedding_last_k_ads**

  The model will learn about the interest of the user using the history of the user's activity on the last k ads that were shown. We can make one embedding by combining embedding vectors of the last k ads that the user engaged with.



embedding of user's last watched ads

- **engagement_content_type**

  This takes into account the content of the ad that the user engages with. Here, we can track which type of ad a user plays around with, such as a video or an image ad.

- **engagement_days**

  This feature captures the user activity on each day of the week. For example, the user might be more active on weekends rather than on weekdays.

- **platform_time_spent**

  This feature captures how long the user has been on the platform. This can be useful because we can show a different ad set to the user every hour to maximize their engagement with the ad.

- **region**

  This feature records the country of the user. Users from different geographical regions have different content preferences. This feature can help the model learn regional preferences and tune the recommendations accordingly.

Context specific features

- **current_region**

  This feature keeps track of the geographical location of the user. Note that the context may change if the user travels to other parts of the world. The system should learn the context of the user and show ads accordingly.

- **time**

  This feature will show ads subject to time of the day. The user would be able to see a different set of ads appear throughout the day.

- **device**

  It can be beneficial to observe the device a person is using to view content on. A potential observation could be that a user tends to watch content for shorter bursts on their mobile. However, they he usually chooses to watch on their laptop when they have more free time, so they watch for longer periods consecutively.

  - **screen_size:**

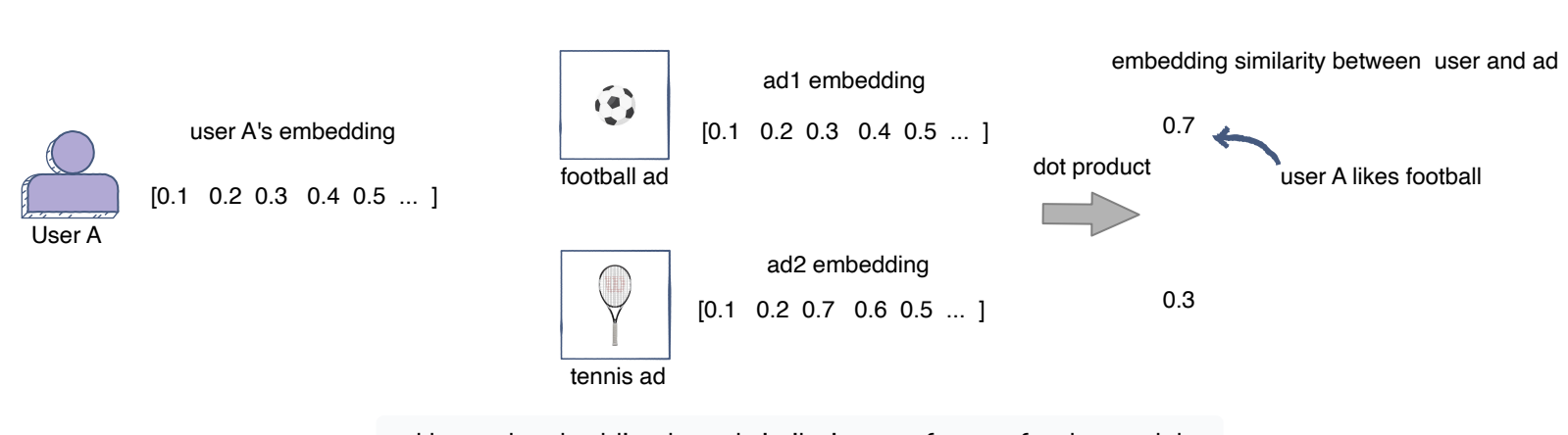
    The size of the screen is an important feature because if the users are using a device with a small screen size, there is a possibility that ads are actually never seen by the users. This is because they don't scroll far down enough to bring the ads in-view.

User-ad cross features

- **embedding_similarity**

  Here, we can generate vectors for the user's interest and the ad's content. We can generate embedding vectors for ads based on their content and for the user based on their interactions with the ad. A dot product between these vectors can be calculated to measure their similarity. A high score would equate to a highly relevant ad for the user. Please refer to our embedding lesson to see a few methods for generating these embeddings.

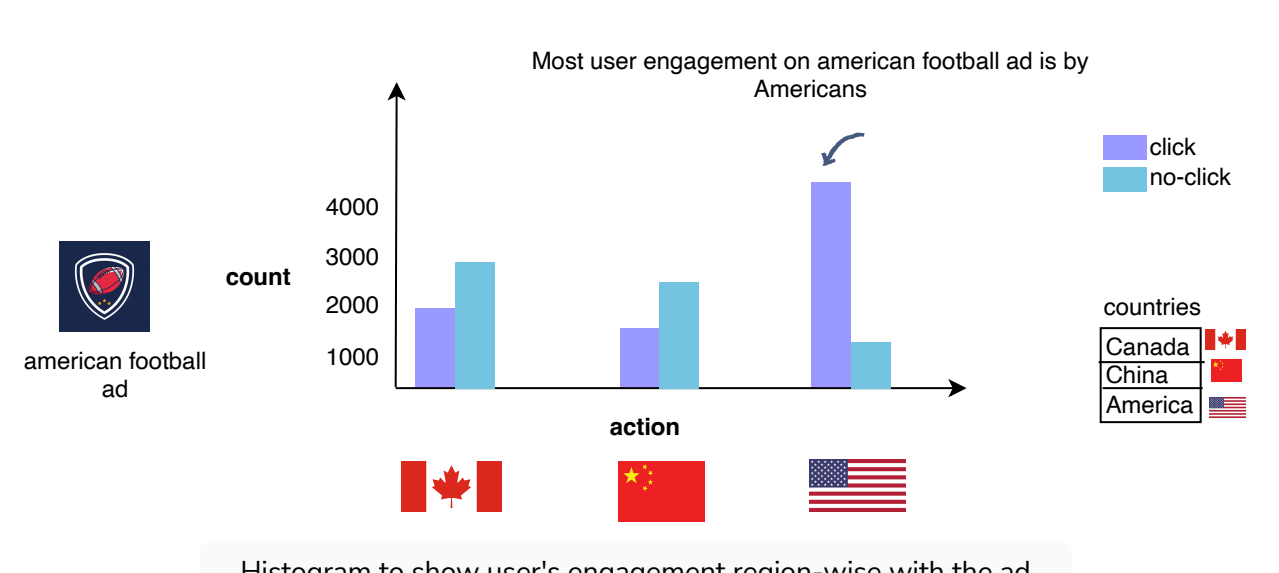
  For example, the ad is about tennis and the user is not interested in tennis. Therefore, this feature will have a low similarity score.



User-ad embedding based similarity as a feature for the model

- **region_wise_engagement**

  An ad engagement radius can be another important feature. For example, the ad-user histogram below shows that an American Football ad is mostly viewed by people living in America.

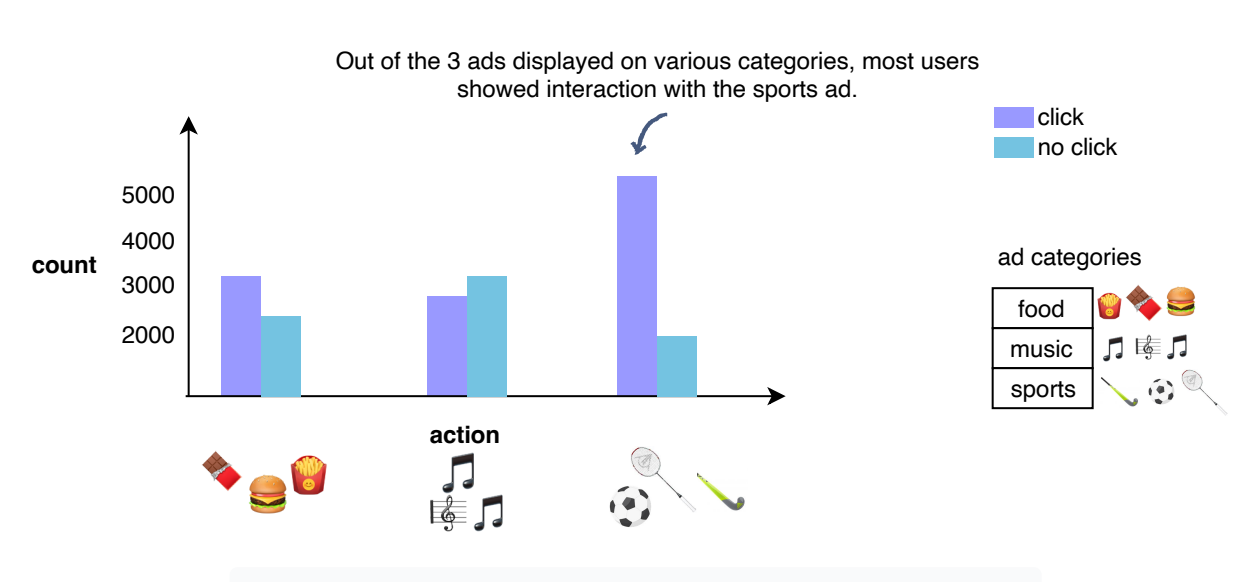


Histogram to show user's engagement region-wise with the ad

- **user_ad_category_histogram**

  This feature observes user engagement on an ad category using a histogram plot showing user engagement on an ad category.

  The following histogram shows that user engagement is the highest on the sports ad.

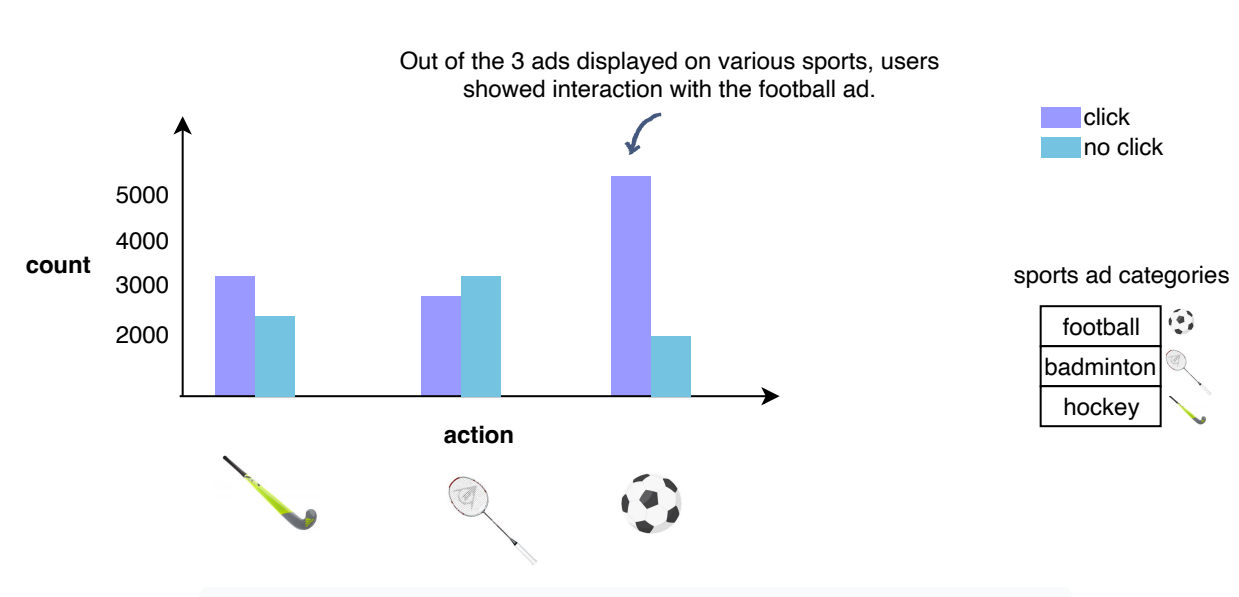


User-ad category histogram based on historical interaction

- **user_ad_subcategory_histogram**

  This feature observes user engagement on an ad subcategory using a histogram plot that shows user engagement on an ad subcategory.

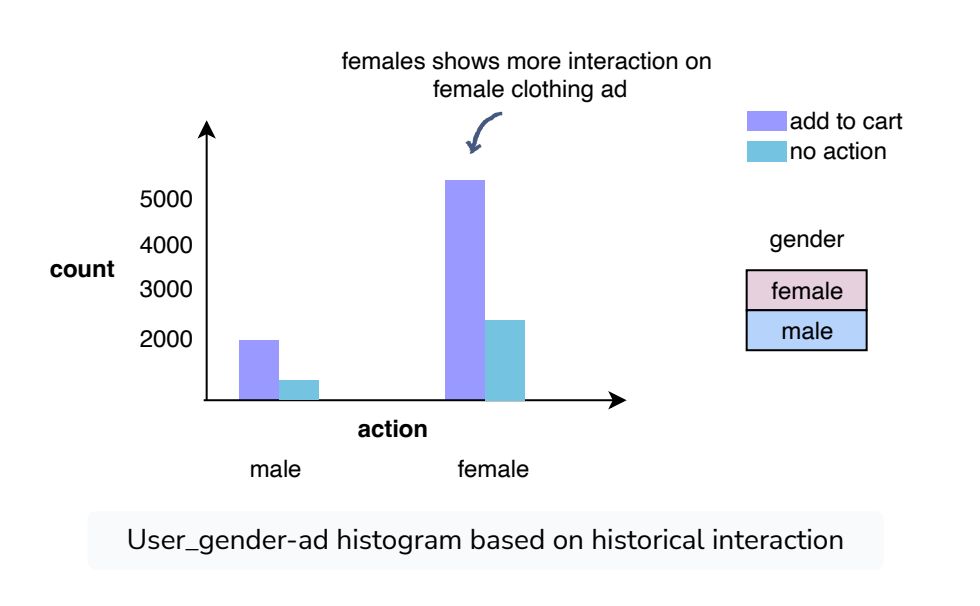
  The following histogram shows that user engagement is the highest on the football ad.



User-ad_sub category histogram based on historical interaction

- **user_gender_ad_histogram**

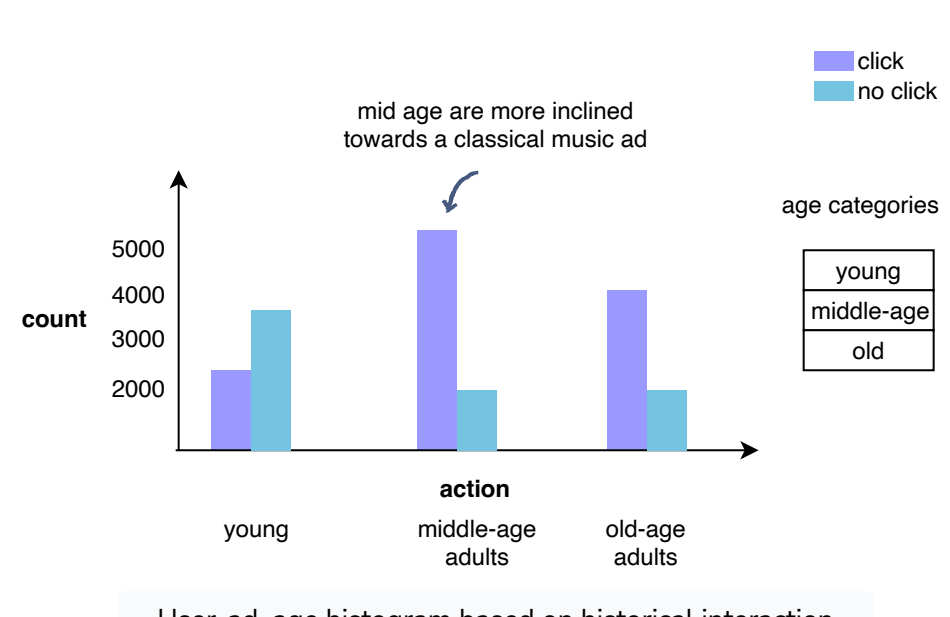
  Some ads can be more appealing to a specific gender. This similarity can be calculated by making a histogram plot showing user engagement gender-wise on an ad. For example, if the ad is for female clothing, it may be primarily of interest to women.



User_gender-ad histogram based on historical interaction

- **user_age_ad_histogram**

  An ad_age histogram can be used to predict user age-wise engagement.

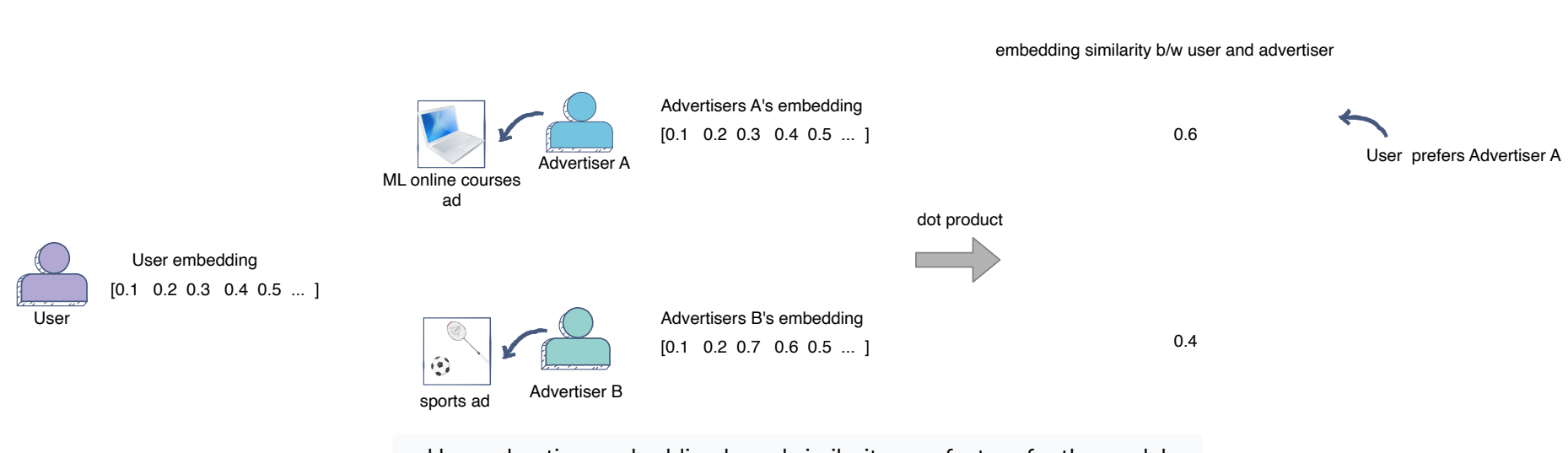


User-ad_age histogram based on historical interaction

User-advertiser cross features

- **embedding_similarity**

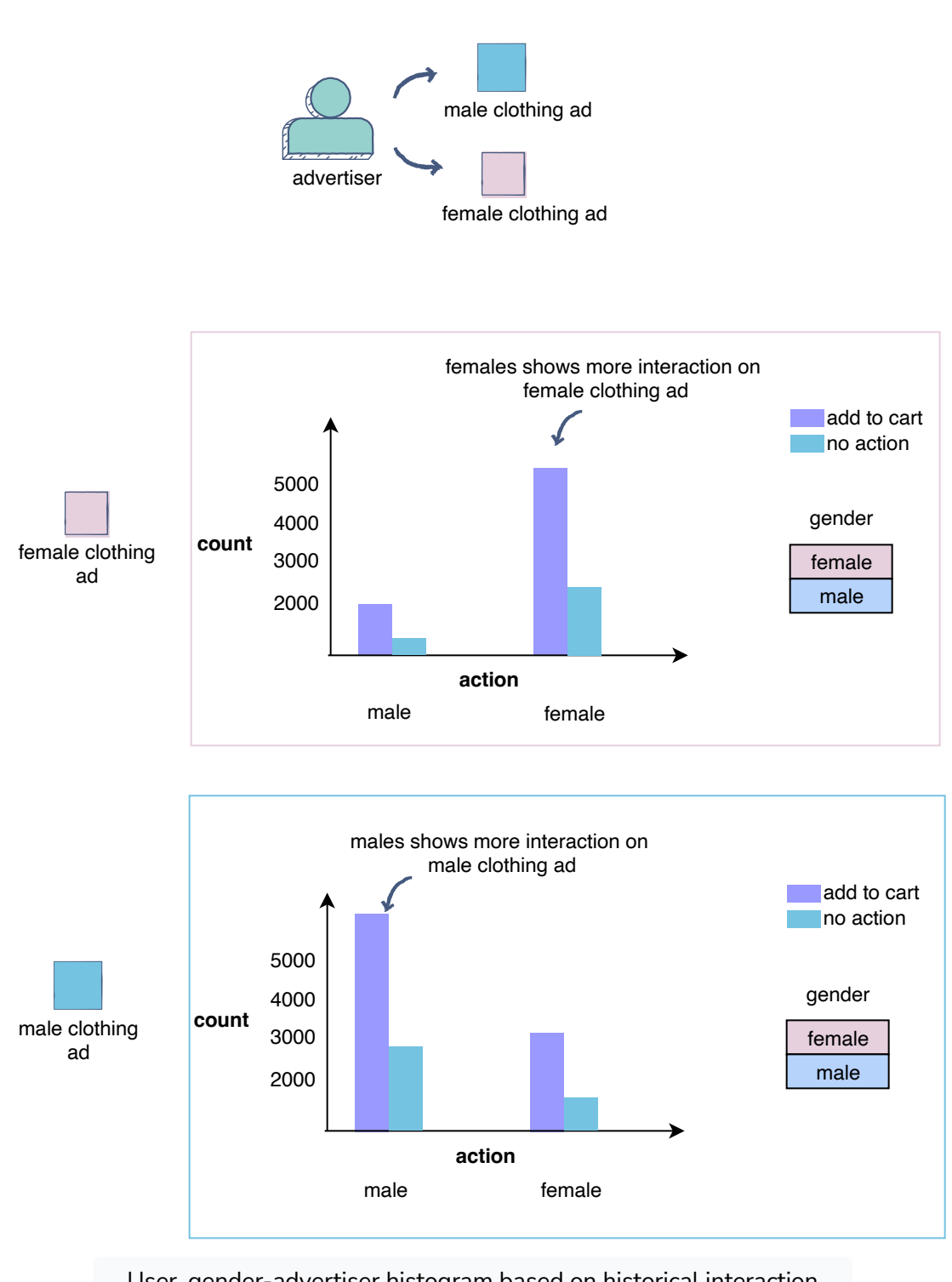
  We can project the advertisement and user in the same embedding space to see how close they are. From the embedding similarity score, we can figure out the type of ads the advertiser shows, and whether the user clicks on those types of ads.



User-advertiser embedding based similarity as a feature for the model

- **user_gender_advertiser_histogram**

  This feature observes gender-wise user engagement on an ad posted by an advertiser using a histogram plot. For example, the advertiser's ad for male clothing might be more engaging for men than women.

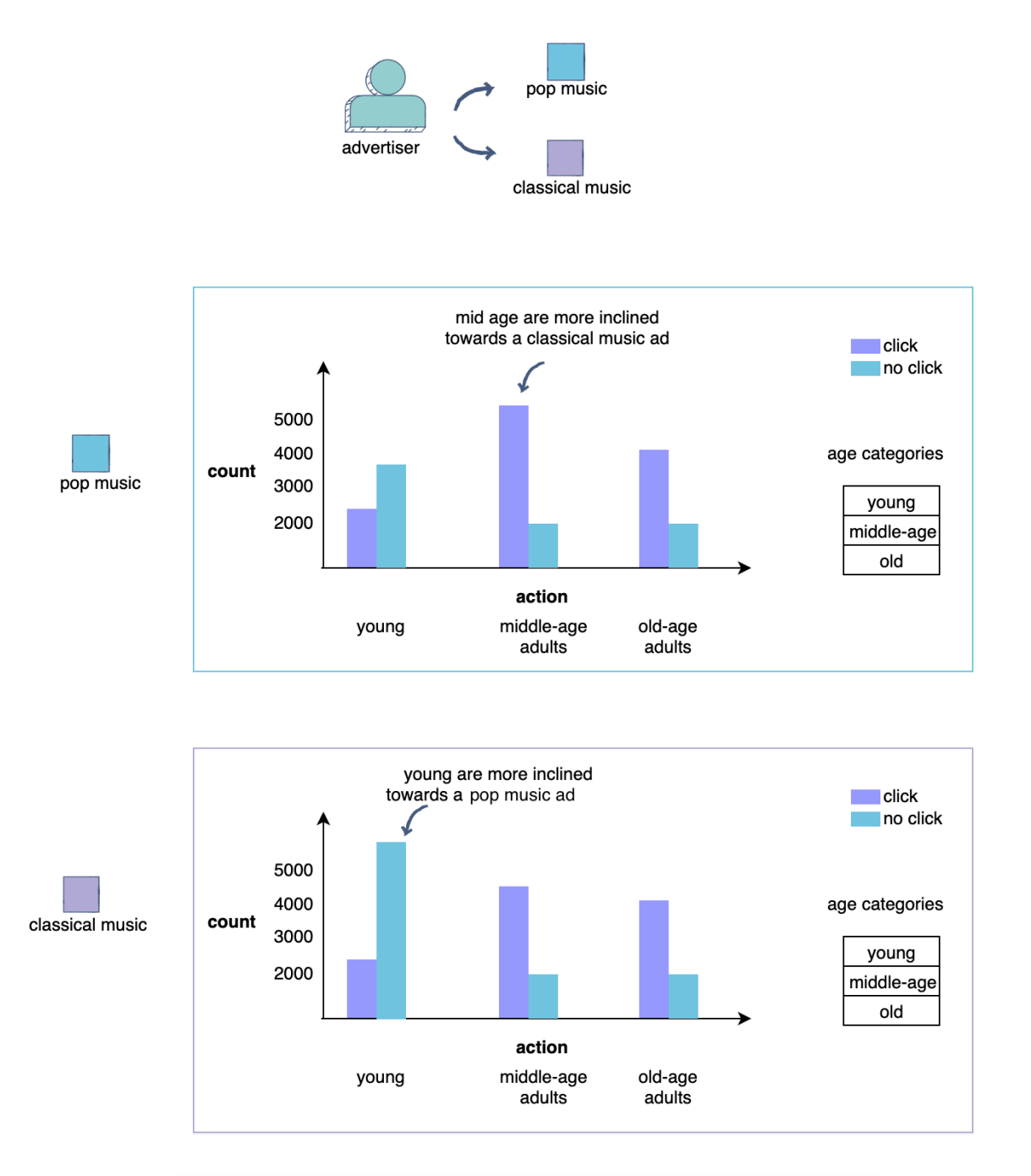


User_gender-advertiser histogram based on historical interaction

- **user_age_advertiser_histogram**

  This feature observes age-wise user engagement on an ad posted by an advertiser using a histogram plot.

  For example, a young audience might demonstrate a greater inclination towards the advertiser's ad on pop music.



User_age-advertiser histogram based on historical interaction

📝 **Key consideration for historical engagement features**

There are two ways to pass historical data that is partitioned on a specific key (such as day, age, gender, etc.) as features to our models.

The **first approach** is to give the histogram data along with the key value to the model for learning. For example, in the case of daily historical engagement rate, we will give the model a histogram of seven-day user engagement from Sunday to Monday, which looks like [0.4, 0.2, 0.21, 0.25, 0.27, 0.38, 0.42]. In addition, we will pass the current day as a feature as well, e.g., if the day is Sunday the feature value of day will be 1, and for Saturday the feature value will be 7. So, the model will see eight features in total: seven more raw features for each day historical engagement rate and one feature for the current day of the week.

The **second approach** for giving this historical information to our model is to create a single feature that gives the current day engagement rate as a value, e.g., the feature value on Sunday will be 0.4, and on Friday it will be 0.38 for the above data.

It would be better to create features for learning by using both the first and second approaches. The models that are able to learn feature interactions effectively (such as neural networks or decision trees) can learn the current day engagement rate by the features described in the first approach. They might also be able to learn more interesting patterns such as understanding weekend data interactions, previous day interactions, etc. However, it's better to provide the full histogram data for the model to learn new interactions for us. But, for linear models providing the current day engagement data as we did in the second approach is critical because they can't learn these interactions on their own.

So, these are few ideas of features that can help our models predict the engagement rate better.

Back
Architectural Components
Mark As Completed
Next
Training Data Generation

Training Data Generation

Let's learn about the techniques for generating training data for the ad prediction system.

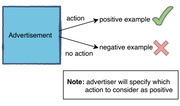
We'll cover the following

- Training data generation through online user engagement
- Balancing positive and negative training examples
- Model recalibration
- Train test split

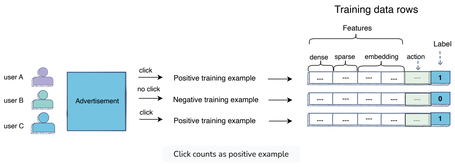
The performance of the user engagement prediction model will depend drastically on the quality and quantity of training data. So let's see how the training data for our model can be generated.

Training data generation through online user engagement

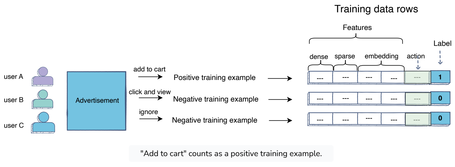
When we show an ad to the user, they can engage with it or ignore it. Positive examples result from users engaging with ads, e.g., clicking or adding an item to their cart. Negative examples result from users ignoring the ads or providing negative feedback on the ad.



Suppose the advertiser specifies "click" to be counted as a positive action on the ad. In this scenario, a user-click on an ad is considered as a positive training example, and a user ignoring the ad is considered as a negative example.



Suppose the ad refers to an online shopping platform and the advertiser specifies the action "add to cart" to be counted as positive user engagement. Here, if the user clicks to view the ad and does not add items to the cart, it is counted as a negative training example.



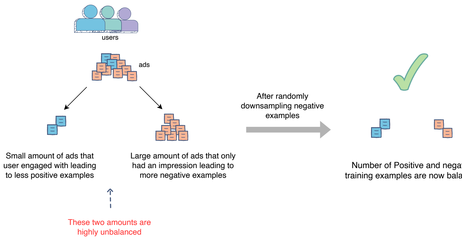
Balancing positive and negative training examples

Users' engagement with an ad can be fairly low based on the platform e.g. in case of a feed system where people generally browse content and engage with minimal content, it can be as low as 2-3%.

How would this percentage affect the ratio of positive and negative examples on a larger scale?

Let's look at an extreme example by assuming that one-hundred million ads are viewed collectively by the users in a day with a 2% engagement rate. This will result in roughly two million positive examples (where people engage with the ad) and 98 million negative examples (where people ignore the ad).

In order to balance the ratio of positive and negative training samples, we can **randomly down sample** the **negative examples** so that we have a similar number of positive and negative examples.



Model recalibration

Negative downsampling can accelerate training while enabling us to learn from both positive and negative examples. However, our predicted model output will now be in the downsampling space. For instance, consider that if our engagement rate is 5% and we select only 10% negative samples, our average predicted engagement rate will be near 50%. Auction uses this predicted rate to determine order and pricing; therefore it's critical that we recalibrate our score before sending them to auction. The recalibration can be done using:

$$q = \frac{p}{p + (1-p)/w}$$

Here,

q is the re-calibrated prediction score,

p is the prediction in downsampling space, and

w is the negative downsampling rate.

Train test split

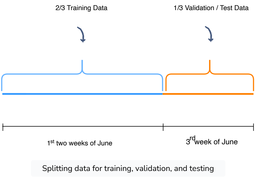
We need to be mindful of the fact that user engagement patterns may differ throughout the week. Hence we will use a week's engagement to capture all patterns during training data generation.

We may randomly select $\frac{2}{3}^{rd}$ or 66.6% training data rows that we have generated and utilize them for training purposes. The rest of the $\frac{1}{3}^{rd}$ or 33.3% can be used for validation and testing of the model.

However, this random splitting would result in utilizing future data for prediction given our data has a time dimension, i.e., we can utilize engagement on historical ads to predict future ad engagement. Hence we will train the model on data from the one-time interval and validate it on the data from its succeeding time interval. This will give a more accurate picture of how our model will perform in a real scenario.

> We are building models with the intent to forecast the user engagement rates on ads.

The following illustration how that we train the model using the engagement data generated in the first two weeks and validate/test it with the data generated in the third week.



Splitting data for training, validation, and testing