Semantic Search with MAX Engine

March 21, 2024

[](https://www.modular.com/team/ehsan-m-kermani)

Touhidul Alam Seyam

AI Advocate

In the field of natural language processing (NLP), semantic search focuses on understanding the context and intent behind queries, going beyond mere keyword matching to provide more relevant and contextually appropriate results. This approach relies on advanced embedding models to convert text into high-dimensional vectors, capturing the complex semantics of language. In this blog post, we will use Amazon Multilingual Counterfactual Dataset (AMCD) which comprises sentences from Amazon customer reviews annotated for counterfactual detection (CFD) in a binary classification task. Counterfactual statements refer to hypothetical scenarios that have not occurred or are impossible to occur. Such statements are typically recognized as having the structure – If p were true, q would also be true, where both the antecedent (p) and the consequent (q) are understood or presumed to be untrue. For instance, a review stating "If this camera had a better lens, my photos would be perfect" suggests a desired improvement (a better lens) that is currently absent, impacting the outcome (perfect photos).

Our classifier will employ the bge-base-en-v1.5 model within the MAX Engine which has 768 embedding dimensions. The BGE model is distinguished as one of the leading text embedding models on the MTEB leaderboard characterized by its minimal disk size of 416MB and available variants with 768 and 1024 embedding dimensions. Furthermore, we will leverage a vector database to store embeddings generated from the training dataset, simulating real-world conditions for batched inference processing. During inference, we will identify the top 10 most similar reviews (using cosine similarity) and assign probabilities to test queries. Subsequently, we will evaluate the classifier's effectiveness through metrics such as accuracy, F1 score, precision, and recall, applying a 0.5 cutoff threshold. Ultimately, we will contrast the performance of MAX Engine with PyTorch and ONNX runtime across various batch sizes, illustrating that

* For small batch sizes on CPU, MAX Engine outperforms PyTorch and ONNX runtime by up to 1.6 and 2.8 times, respectively.
* With large batch sizes on CPU, MAX Engine outperforms PyTorch and ONNX runtime by 2 and 1.8 times, respectively.

The code for this blog post is available in our GitHub repository. The MAX version for this blog is 24.1.1 (0ab415f7).

Dataset and input tokenizer

Let’s first examine the data in Amazon Multilingual Counterfactual Dataset (AMCD)

**Python**

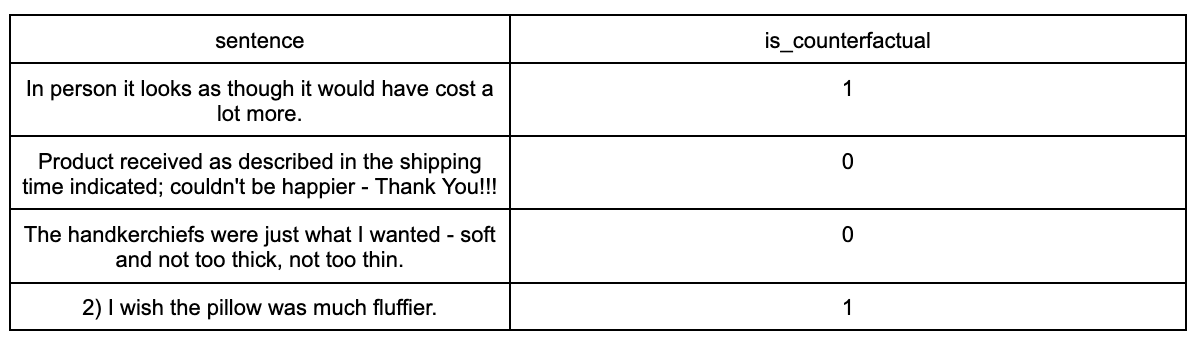
*import* pandas *as* pd

data = pd.read\_csv("amazon-multilingual-counterfactual-dataset/data/EN\_train.tsv", *sep*="\t")

data.head()

The dataframe consists of two columns: sentence (from Amazon customer review) and is\_counterfactual (the label) and a total of 4018 samples.

For example:



Next we tokenize all the input sentences in data as follows

**Python**

*from* transformers *import* AutoTokenizer

tokenizer = AutoTokenizer.from\_pretrained("BAAI/bge-base-en-v1.5")

inputs = tokenizer(list(data['sentence']), *return\_tensors*="pt", *max\_length*=512, *padding*=True, *truncation*=True)

With the inputs tokenized, we are now ready to proceed to inference and create sentence embeddings.

MAX Engine inference

In this blog post, we will utilize the ONNX version of the model, available on HuggingFace. We can obtain use the following command (ensure you have Git LFS installed)

**Bash**

git lfs install

git clone https://huggingface.co/BAAI/bge-base-en-v1.5

The ONNX model is located at bge-base-en-v1.5/onnx/model.onnx.

Below, we create a session object and load the model into maxmodel. We also examine the input and output tensors, noting their names, shapes, and data types:

**Python**

*from* max *import* engine

session = engine.InferenceSession()

maxmodel = session.load("bge-base-en-v1.5/onnx/model.onnx")

*for* tensor *in* maxmodel.input\_metadata:

print(f'input name: {tensor.name}, shape: {tensor.shape}, dtype: {tensor.dtype}')

*for* tensor *in* maxmodel.output\_metadata:

print(f'output name: {tensor.name}, shape: {tensor.shape}, dtype: {tensor.dtype}')

The model has three input tensors — input\_ids, attention\_mask, and token\_type\_ids — and one output tensor, last\_hidden\_state:

**Output**

input name: input\_ids, shape: [None, None], dtype: DType.int64

input name: attention\_mask, shape: [None, None], dtype: DType.int64

input name: token\_type\_ids, shape: [None, None], dtype: DType.int64

output name: last\_hidden\_state, shape: [None, None, 768], dtype: DType.float32

The model's pooling configuration file is as follows which we will use later to accurately obtain our sentence embeddings.

**Output**

{

"word\_embedding\_dimension": 768,

"pooling\_mode\_cls\_token": true,

"pooling\_mode\_mean\_tokens": false,

"pooling\_mode\_max\_tokens": false,

"pooling\_mode\_mean\_sqrt\_len\_tokens": false

}

Optional: Convert to ONNX using optimum

Another notable option is the conversion of models to the ONNX format using the optimum package which can be done through its command-line-interface (CLI).

Note that converting to ONNX offers benefits like framework interoperability across different platforms. For instance, to convert the BAAI/bge-base-en-v1.5 model to ONNX:

**Bash**

optimum-cli export onnx --model "BAAI/bge-base-en-v1.5" "./onnx/bge-base-en-v1.5"

Sentence embeddings

To enhance efficiency, especially with large datasets, we batch the input sentences before embedding. This approach not only accelerates the processing but also helps manage memory usage more effectively.

Here, in each batch we simply call maxmodel.execute and iterate on the training data until all sentences are embedded.

**Python**

*import* numpy *as* np

*from* torch.utils.data *import* DataLoader, TensorDataset

ds = TensorDataset(inputs["input\_ids"], inputs["token\_type\_ids"], inputs["attention\_mask"])

data\_loader = DataLoader(ds, *batch\_size*=128, *shuffle*=False)

output\_embeddings = []

*for* batch *in* data\_loader:

batch\_input\_ids, batch\_token\_type\_ids, batch\_attention\_mask = batch

batch\_outputs = maxmodel.execute(*input\_ids*=batch\_input\_ids, *token\_type\_ids*=batch\_token\_type\_ids, *attention\_mask*=batch\_attention\_mask)

last\_hidden\_state = batch\_outputs["last\_hidden\_state"]

*# Extract the CLS token embedding*

sentence\_embeddings = last\_hidden\_state[:, 0, :]

output\_embeddings.append(sentence\_embeddings)

*# concatenate all into one array*

all\_embeddings = np.concatenate(output\_embeddings, *axis*=0)

print(f"All embeddings dimensions: {all\_embeddings.shape}")

which outputs

**Output**

All embeddings dimensions: (4018, 768)

After obtaining the embeddings, they can be utilized for various NLP tasks, such as semantic similarity, clustering, or as input features for machine learning models. In the next section, we will store them in a vector database for semantic search.

Using a Vector Database

Vector databases excel in managing and querying high-dimensional data, making them ideal for storing embeddings. We chose ChromaDB which is an embedded vector database and is known for its efficiency and straightforward usage, particularly fitting for small to medium-sized applications. ChromaDB stands out due to its fast querying capabilities and lightweight nature.

To start, we create a client and a collection to store our embeddings as follows

**Python**

*import* chromadb

chroma\_client = chromadb.Client()

collection = chroma\_client.create\_collection(*name*="counterfactual\_collection", *metadata*={"hnsw:space": "cosine"})

*for* i, (documents, embeddings, label) *in* enumerate(zip(list(data['sentence']), all\_embeddings.tolist(), list(data['is\_counterfactual']))):

collection.upsert(*ids*=[str(i)], *documents*=documents, *embeddings*=embeddings, *metadatas*=[{"is\_counterfactual": label}])

Search in Vector Database collection

To demonstrate the practical application, we query the database using a test sentence. After tokenizing this sentence and generating its embedding, we search the vector database for the most similar entries, using cosine similarity to identify and return the top 10 most similar items. Cosine similarity is particularly effective for embeddings because it focuses on the orientation of vectors rather than their magnitude. Finally, we assign a probability by normalizing the count of positive is\_counterfactual results from the top 10 queries, leveraging cosine similarity.

**Python**

query = "I've worn my boots a couple times without proper socks and I can definitely tell the difference!"

query\_inputs = tokenizer(query, *return\_tensors*="np", *max\_length*=512, *padding*=True, *truncation*=True)

query\_output = maxmodel.execute(*input\_ids*=query\_inputs["input\_ids"], *token\_type\_ids*=query\_inputs["token\_type\_ids"], *attention\_mask*=query\_inputs["attention\_mask"])

*# Extract the CLS token embedding*

query\_embeddings = query\_output["last\_hidden\_state"][:, 0, :]

results = collection.query(query\_embeddings, *n\_results*=10)

counterfactual\_prob = sum([r["is\_counterfactual"] *for* r *in* results["metadatas"][0]]) / len(results["metadatas"][0])

print(f"counterfactual probability is {counterfactual\_prob \* 100}%")

which outputs

**Output**

counterfactual probability is 10.0%

Assess test accuracy, F1-score, precision and recall

We evaluate our model on a test dataset using common metrics.

* Accuracy provides a general sense of performance
* F1-score balances precision and recall
* Precision measures the model's exactness and
* Recall assesses its completeness

**Python**

*import* pandas *as* pd

*from* sklearn.metrics *import* accuracy\_score, f1\_score, precision\_score, recall\_score

test\_data = pd.read\_csv("amazon-multilingual-counterfactual-dataset/data/EN\_test.tsv", *sep*="\t")

cutoff\_threshold = 0.5

def get\_counterfactual\_prob(*sentence*):

query\_inputs = tokenizer(sentence, *return\_tensors*="np", *max\_length*=512, *padding*=True, *truncation*=True)

query\_output = maxmodel.execute(*input\_ids*=query\_inputs["input\_ids"], *token\_type\_ids*=query\_inputs["token\_type\_ids"], *attention\_mask*=query\_inputs["attention\_mask"])

*# Extract the CLS token embedding*

query\_embeddings = query\_output["last\_hidden\_state"][:, 0, :]

results = collection.query(query\_embeddings, *n\_results*=10)

counterfactual\_prob = sum([r["is\_counterfactual"] *for* r *in* results["metadatas"][0]]) / len(results["metadatas"][0])

*return* counterfactual\_prob

predictions = []

*for* index, row *in* test\_data.iterrows():

counterfactual\_prob = get\_counterfactual\_prob(row['sentence'])

prediction = 1 *if* counterfactual\_prob > cutoff\_threshold *else* 0

predictions.append(prediction)

accuracy = accuracy\_score(test\_data['is\_counterfactual'], predictions)

f1 = f1\_score(test\_data['is\_counterfactual'], predictions)

precision = precision\_score(test\_data['is\_counterfactual'], predictions)

recall = recall\_score(test\_data['is\_counterfactual'], predictions)

print(f"Accuracy: {accuracy:.2f}")

print(f"F1 Score: {f1:.2f}")

print(f"Precision: {precision:.2f}")

print(f"Recall: {recall:.2f}")

which outputs

**Output**

Accuracy: 0.88

F1 Score: 0.64

Precision: 0.75

Recall: 0.56

Comparing MAX Engine performance against PyTorch eager and ONNX runtime

Recall that to efficiently compute all sentence embeddings, we processed them in batches, using a batch size of 128. This batching is particularly important in data intensive scenarios for optimizing resource utilization and processing speed. Consequently, we aim to compare the performance of MAX Engine against PyTorch and ONNX runtime across various batch sizes to understand their respective efficiencies in handling batched data.

To make a compelling comparison between MAX Engine, PyTorch and ONNX runtime, we meticulously selected a range of batch sizes and for better visualization, we divided them up into two categories of

* smaller batch sizes: 1 up to 32 and
* larger batch sizes: 64 up to 4096

These wide arrays allow us to observe the performance scalability and efficiency of each framework under different load conditions. The runtime for each batch size is measured, offering a clear picture of how each framework handles varying volumes of data. This evaluation is crucial for developers and engineers to make informed decisions about the tools and frameworks best suited for their specific NLP tasks, especially in resource-intensive scenarios such as working with large datasets. For completeness, runtime measurements were done on an AWS c5.12xlarge instance.

**Python**

*import* gc

*import* torch

*from* torch.utils.data *import* DataLoader, TensorDataset

*import* numpy *as* np

*import* time

*from* transformers *import* AutoModel

*from* optimum.onnxruntime *import* ORTModelForFeatureExtraction

model = AutoModel.from\_pretrained("BAAI/bge-base-en-v1.5")

model.eval()

ortmodel = ORTModelForFeatureExtraction.from\_pretrained("BAAI/bge-base-en-v1.5", *revision*="refs/pr/6", *file\_name*="onnx/model.onnx")

def measure\_runtime(*inputs*, *model\_fn*, *batch\_sizes*, *is\_pytorch*=True):

results = {}

ds = TensorDataset(inputs["input\_ids"], inputs["token\_type\_ids"], inputs["attention\_mask"])

*for* batch\_size *in* batch\_sizes:

data\_loader = DataLoader(ds, *batch\_size*=batch\_size, *shuffle*=False)

times = []

*for* batch *in* data\_loader:

start\_time = time.time()

batch\_input\_ids, batch\_token\_type\_ids, batch\_attention\_mask = batch

*if* is\_pytorch:

*with* torch.no\_grad():

\_ = model\_fn(*input\_ids*=batch\_input\_ids, *token\_type\_ids*=batch\_token\_type\_ids, *attention\_mask*=batch\_attention\_mask)

*else*:

\_ = model\_fn(*input\_ids*=batch\_input\_ids, *token\_type\_ids*=batch\_token\_type\_ids, *attention\_mask*=batch\_attention\_mask)

end\_time = time.time()

times.append(end\_time - start\_time)

gc.collect()

times = np.array(times)

mean\_time = np.mean(times)

std\_time = np.std(times)

*# 95% confidence interval with normality distribution assumption of time measurements*

confidence\_interval = 1.96 \* (std\_time / np.sqrt(len(times)))

results[batch\_size] = {'mean\_time': mean\_time, 'std\_time': std\_time, 'confidence\_interval': confidence\_interval}

*return* results

Now we use the PyTorch, ONNX model and MAX Engine model individually and plot their performance.

**Python**

*import* matplotlib.pyplot *as* plt

small\_batch\_sizes = [2 \*\* i *for* i *in* range(6)]

small\_results = measure\_runtime(inputs, model, small\_batch\_sizes)

small\_maxresults = measure\_runtime(inputs, lambda \*\**kwargs*: maxmodel.execute(\*\*kwargs), small\_batch\_sizes, *is\_pytorch*=False)

small\_ortresults = measure\_runtime(inputs, ortmodel, small\_batch\_sizes, *is\_pytorch*=False)

def plot\_performance\_comparison(*batch\_sizes*, *results\_with\_labels*, *title*):

plt.figure(*figsize*=(10, 6))

*for* label, res *in* results\_with\_labels:

mean\_times = [res[bs]['mean\_time'] *for* bs *in* batch\_sizes]

conf\_intervals = [res[bs]['confidence\_interval'] *for* bs *in* batch\_sizes]

plt.errorbar(batch\_sizes, mean\_times, *yerr*=conf\_intervals, *fmt*='-o', *capsize*=5, *label*=label)

plt.xlabel('Batch Size')

plt.ylabel('Mean Time (seconds)')

plt.title(title)

plt.legend()

plt.grid(True)

plt.show()

plot\_performance\_comparison(small\_batch\_sizes,

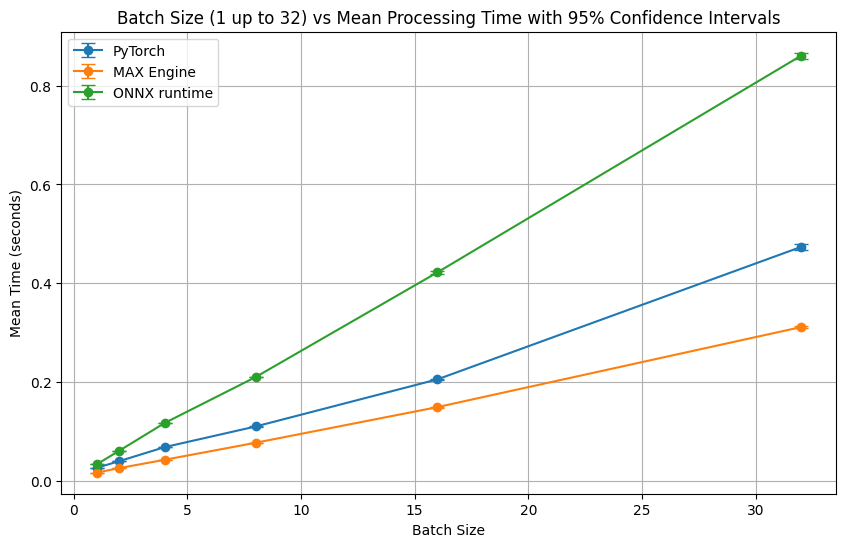
[("PyTorch", small\_results),

("MAX Engine", small\_maxresults),

("ONNX runtime", small\_ortresults)],

*title*="Batch Size (1 up to 32) vs Mean Processing Time with 95% Confidence Intervals")

This analysis revealed that for smaller batch sizes (1 up to 32), MAX Engine can be up to 1.6 times faster than PyTorch and is up to 2.8 times faster than ONNX runtime for batch inference.



And for larger batch sizes (64 up to 4096), MAX Engine can be up to 2 and 1.8 times faster than PyTorch and ONNX runtime, respectively. This is showcasing MAX Engine efficiency in handling high-volume data processing tasks.

**Python**

large\_batch\_sizes = [2 \*\* i *for* i *in* range(6, 13)]

large\_results = measure\_runtime(inputs, model, large\_batch\_sizes)

large\_maxresults = measure\_runtime(inputs, lambda \*\**kwargs*: maxmodel.execute(\*\*kwargs), large\_batch\_sizes, *is\_pytorch*=False)

plot\_performance\_comparison(large\_batch\_sizes,

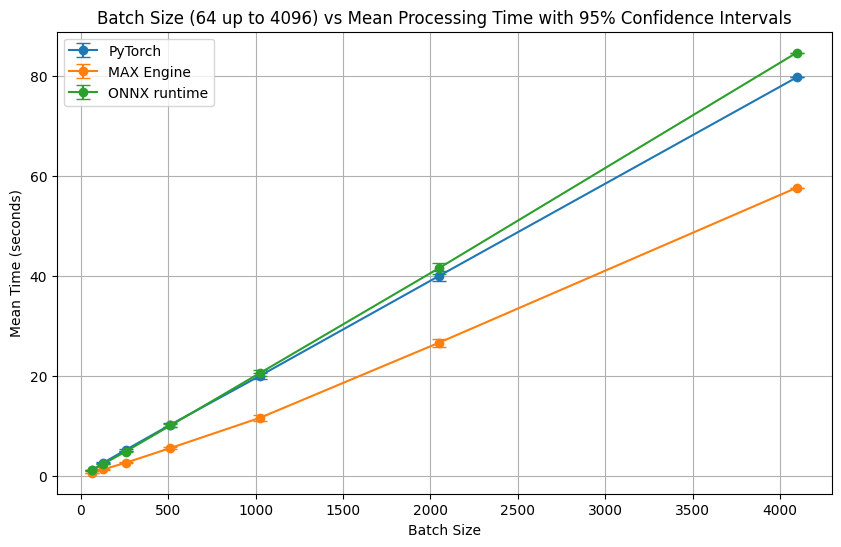
[("PyTorch", large\_results),

("MAX Engine", large\_maxresults),

("ONNX runtime", large\_ortresults)],

*title*="Batch Size (64 up to 4096) vs Mean Processing Time with 95% Confidence Intervals")

which shows



Conclusion

We have illustrated the application of MAX Engine with a pre-trained model for counterfactual binary classification, demonstrating the process of storing embeddings in a vector database suited for inference. Furthermore, our comparison between MAX Engine and PyTorch across various batch sizes has revealed that MAX Engine can achieve up to 1.6 and 2.8 times the speed up against PyTorch and ONNX runtime for varying small batch sizes when running on a CPU, and is 2 and 1.8 times faster against PyTorch and ONNX runtime on large batch sizes, respectively. This efficiency gain highlights MAX Engine's potential to significantly enhance processing speed and resource utilization in large-scale NLP tasks.