Understanding and Applying Embeddings

In this course we shall use Vertexai platform for the purpose of easier usage of text embeddings model on cloud rather than loading it into local system,

"from vertexai.language_models import TextEmbeddingModel

embedding_model = TextEmbeddingModel.from_pretrained("textembedding-gecko@001")"

Now let's create a vector on the word 'life' and print the first 10 values of the vector:

"embedding = embedding_model.get_embeddings(["life"])

vector = embedding[0].values

print(f"Length = {len(vector)}")

print(vector[:10])"

And thus we have created an embedding with the help of the model taken and we can also generate sentence embeddings similarly:

"embedding = embedding_model.get_embeddings(["What is the meaning of life?"])

vector = embedding[0].values

print(f"Length = {len(vector)}")

print(vector[:10])"

From these two examples one can figure out how similar the two embeddings are as they are of the same length and have very similar values thus preaching their similarity.

Now lets try to calculate their similarity between embeddings using cosine similarity,

Calculate the similarity between two sentences as a number between 0 and 1.

Try out your own sentences and check if the similarity calculations match your intuition.

"from sklearn.metrics.pairwise import cosine_similarity

emb_1 = embedding_model.get_embeddings(["What is the meaning of life?"]) # 42!

emb_2 = embedding_model.get_embeddings(["How does one spend their time well on Earth?"])

emb_3 = embedding_model.get_embeddings(["Would you like a salad?"])

vec_1 = [emb_1[0].values]

vec_2 = [emb_2[0].values]

vec_3 = [emb_3[0].values]"

The reason we wrap the embeddings (a Python list) in another list is because the cosine_similarity function expects either a 2D numpy array or a list of lists.

vec_1 = [emb_1[0].values]

"print(cosine_similarity(vec_1,vec_2))

print(cosine_similarity(vec_2,vec_3))

print(cosine_similarity(vec_1,vec_3))"

This gives the values where it compares the similarity with vector-1 and vector-2 and so-on. We can do this with programming codes/sentences/words/numbers.

The average of the word embeddings can be used to determine sentence embeddings from word embeddings.
Two sentences with different meanings but the same set of words will have the same sentence embedding because this disregards word order and context.

"in_1 = "The kids play in the park."

in_2 = "The play was for kids in the park.""
now if we remove punctuations and stop words:

"in_pp_1 = ["kids", "play", "park"]

in_pp_2 = ["play", "kids", "park"]"

Generating one embedding for each word results a list of three lists for each string.

"embeddings_1 = [emb.values for emb in embedding_model.get_embeddings(in_pp_1)]"

Use numpy to convert this list of lists into a 2D array of 3 rows and 768 columns.

"import numpy as npemb_array_1 = np.stack(embeddings_1)print(emb_array_1.shape)

embeddings_2 = [emb.values for emb in embedding_model.get_embeddings(in_pp_2)]

emb_array_2 = np.stack(embeddings_2)print(emb_array_2.shape)"

Take the average embedding across the 3 word embeddings

You'll get a single embedding of length 768.

“emb_1_mean = emb_array_1.mean(axis = 0) print(emb_1_mean.shape)

emb_2_mean = emb_array_2.mean(axis = 0)”

Check to see that taking an average of word embeddings results in two sentence embeddings that are identical.

“print(emb_1_mean[:4])print(emb_2_mean[:4])”

By getting sentence embeddings from the model we can that these sentence embeddings account for word order and context and verify that the sentence embeddings are not the same.

“print(in_1)print(in_2)

embedding_1 = embedding_model.get_embeddings([in_1])embedding_2 = embedding_model.get_embeddings([in_2])

vector_1 = embedding_1[0].valuesprint(vector_1[:4])vector_2 = embedding_2[0].valuesprint(vector_2[:4])”

Let us try to visualize embeddings for a better understanding of the working of the embeddings:

“in_1 = "Missing flamingo discovered at swimming pool"

in_2 = "Sea otter spotted on surfboard by beach"

in_3 = "Baby panda enjoys boat ride"

in_4 = "Breakfast themed food truck beloved by all!"

in_5 = "New curry restaurant aims to please!"

in_6 = "Python developers are wonderful people"

in_7 = "TypeScript, C++ or Java? All are great!"

input_text_lst_news = [in_1, in_2, in_3, in_4, in_5, in_6, in_7]”, these are the sentences which we are going to use for examples on which we will create embeddings and store them in a 2d array such that each row contains one embedding.

```
“embeddings = []

for input_text in input_text_lst_news:

    emb = embedding_model.get_embeddings(

        [input_text])[0].values

    embeddings.append(emb)

embeddings_array = np.array(embeddings)
```

print("Shape: " + str(embeddings_array.shape))

print(embeddings_array)"

We know that embeddings are stored in high dimensions which makes it difficult to visualize it thus we reduce the dimensions to 2 using PCA like:

"from sklearn.decomposition import PCA # Perform PCA for 2D visualization

PCA_model = PCA(n_components = 2)

PCA_model.fit(embeddings_array)

new_values = PCA_model.transform(embeddings_array)

print("Shape: " + str(new_values.shape))print(new_values)

import matplotlib.pyplot as pltimport mplcursors%matplotlib ipympl from utils import plot_2Dplot_2D(new_values[:,0], new_values[:,1], input_ text_lst_news)"

Now lets compare embeddings on sentences that are similar and that are dissimilar on a heat map:

"

in_1 = """He couldn't desert   his post at the power plant."""

in_2 = """The power plant needed   him at the time."""

in_3 = """Cacti are able to   withstand dry environments."""

in_4 = """Desert plants can   survive droughts."""

input_text_lst_sim = [in_1, in_2, in_3, in_4]

embeddings = []

for input_text in input_text_lst_sim:

emb = embedding_model.get_embeddings([input_text])[0].values

embeddings.append(emb)

embeddings_array = np.array(embeddings)

from utils import plot_heatmap

y_labels = input_text_lst_sim

 # Plot the heatmap

plot_heatmap(embeddings_array, y_labels = y_labels, title = "Embeddings Heatmap")"

Now let us practically implement and Load Stack Overflow questions and answers from BigQuery, BigQuery is Google Cloud's serverless data warehouse, We'll get the first 500 posts (questions and answers) for each programming language: Python, HTML, R, and CSS.

“from google.cloud import bigqueryimport pandas as pd

def run_bq_query(sql):

Create BQ client

bq_client = bigquery.Client(project = PROJECT_ID,

credentials = credentials)

Try dry run before executing query to catch any errors

job_config = bigquery.QueryJobConfig(dry_run=True, use_query_cache=False)

bq_client.query(sql, job_config=job_config)

If dry run succeeds without errors, proceed to run query

job_config = bigquery.QueryJobConfig()

client_result = bq_client.query(sql, job_config=job_config)

job_id = client_result.job_id

Wait for query/job to finish running. then get & return data frame

df = client_result.result().to_arrow().to_pandas()

print(f"Finished job_id: {job_id}")

return df”

“# define list of programming language tags we want to query

language_list = ["python", "html", "r", "css"]

so_df = pd.DataFrame()

for language in language_list:

print(f"generating {language} dataframe")

query = f""" SELECT CONCAT(q.title, q.body) as input_text, a.body AS output_text FROM `bigquery-public-data.stackoverflow.posts_questions` q JOIN `bigquery-public-data.stackoverflow.posts_answers` a ON q.accepted_answer_id = a.id WHERE

```
q.accepted_answer_id IS NOT NULL AND        REGEXP_CONTAINS(q.tags, "{language}")
AND        a.creation_date >= "2020-01-01"    LIMIT        500    """

language_df = run_bq_query(query)

language_df["category"] = language

so_df = pd.concat([so_df, language_df, ignore_index = True)”
```

You can reuse the above code to run your own queries if you are using Google Cloud's BigQuery service.

To generate embeddings for a dataset of texts, we'll need to group the sentences together in batches and send batches of texts to the model. The API used currently can take batches of up to 5 pieces of text per API call.

```
“from vertexai.language_models import TextEmbeddingModel

model = TextEmbeddingModel.from_pretrained(    "textembedding-gecko@001")

import timeimport numpy as np

# Generator function to yield batches of sentences

def generate_batches(sentences, batch_size = 5):

for i in range(0, len(sentences), batch_size):

yield sentences[i : i + batch_size]

so_questions = so_df[0:200].input_text.tolist()

batches = generate_batches(sentences = so_questions)

batch = next(batches)

len(batch)”
```

This helper function calls model.get_embeddings() on the batch of data, and returns a list containing the embeddings for each text in that batch.

```
“def encode_texts_to_embeddings(sentences):

try:

embeddings = model.get_embeddings(sentences)

return [embedding.values for embedding in embeddings]

except Exception:

return [None for _ in range(len(sentences))]
```

batch_embeddings = encode_texts_to_embeddings(batch)

f"{len(batch_embeddings)} embeddings of size \{len(batch_embeddings[0])}"”

Most API services have rate limits, so we have a helper function that you could use to wait in-between API calls. If the code was not designed to wait in-between API calls, you may not receive embeddings for all batches of text. This particular service can handle 20 calls per minute. In calls per second, that's 20 calls divided by 60 seconds, or 20/60.

```
“from utils import encode_text_to_embedding_batched

so_questions = so_df.input_text.tolist()
question_embeddings = encode_text_to_embedding_batched(
                        sentences=so_questions,
                        api_calls_per_second = 20/60,
                        batch_size = 5)”
```

In order to handle limits of this classroom environment, we're not going to run this code to embed all of the data. But you can adapt this code for your own projects and datasets.

We'll load the stack overflow questions, answers, and category labels (Python, HTML, R, CSS) from a .csv file. We'll load the embeddings of the questions (which we've precomputed with batched calls to model.get_embeddings()), from a pickle file.

“so_df = pd.read_csv('so_database_app.csv')

so_df.head()

import pickle

with open('question_embeddings_app.pkl', 'rb') as file:

question_embeddings = pickle.load(file)

print("Shape: " + str(question_embeddings.shape))

print(question_embeddings)”

Cluster the embeddings of the Stack Overflow questions

“from sklearn.cluster import KMeans

from sklearn.decomposition import PCA

clustering_dataset = question_embeddings[:1000]

n_clusters = 2

kmeans = KMeans(n_clusters=n_clusters, random_state=0, n_init ='auto').fit(clustering_dataset)

*kmeans_labels = kmeans.labels_*

PCA_model = PCA(n_components=2)

PCA_model.fit(clustering_dataset)

new_values = PCA_model.transform(clustering_dataset)

import matplotlib.pyplot as pltimport

from utils import clusters_2D

clusters_2D(x_values = new_values[:,0], y_values = new_values[:,1, labels = so_df[:1000], kmeans_labels = kmeans_labels)"

Clustering is able to identify two distinct clusters of HTML or Python related questions, without being given the category labels (HTML or Python).

Anomaly / Outlier detection

We can add an anomalous piece of text and check if the outlier (anomaly) detection algorithm (Isolation Forest) can identify it as an outlier (anomaly), based on its embedding.

from sklearn.ensemble import IsolationForest

input_text = """I am making cookies but don't remember the correct ingredient proportions. I have been unable to find anything on the web."""

emb = model.get_embeddings([input_text])[0].values

embeddings_l = question_embeddings.tolist()embeddings_l.append(emb)

embeddings_array = np.array(embeddings_l)

print("Shape: " + str(embeddings_array.shape))print(embeddings_array)

 # Add the outlier text to the end of the stack overflow dataframeso_df = pd.read_csv('so_database_app.csv')new_row = pd.Series([input_text, None, "baking"], index=so_df.columns)so_df.loc[len(so_df)+1] = new_rowso_df.tail()