**To Calculate the similarity of two sentences**

Install Python https://www.python.org/downloads/

1. Run cmd
2. Check Python is installed

Type> Python.exe –version

1. Install Sentence Transformers

Type> python -m pip install -U sentence-transformers

Open the Python run environment

*This method of sentence similarity uses cosine similarity on sentence embeddings using Sentence Transformers library in Python*

Here's how you can implement this using the Sentence Transformers library:

Copy and past this code:

python

**from** sentence\_transformers **import** SentenceTransformer, util

*# Load the model*

model = SentenceTransformer('all-MiniLM-L6-v2')

*# Define sentences*

sentence1 = "The apple is red and juicy."

sentence2 = "The arch is red and just."

*# Encode sentences to get their embeddings*

embedding1 = model.encode(sentence1, convert\_to\_tensor=True)

embedding2 = model.encode(sentence2, convert\_to\_tensor=True)

*# Compute cosine similarity*

similarity = util.pytorch\_cos\_sim(embedding1, embedding2)

*# Print the similarity score*

**print**(f"Similarity score: {similarity.item():.4f}")

Use this one for batch processing:

Note: sentence pairings are hard coded in code below. Can copy this script into a Python file (e.g., sentence\_comparison.py).

**from** sentence\_transformers **import** SentenceTransformer, util

**import** csv

**def** compare\_sentences(model, sentence\_pairs):

*# Encode all sentences at once*

sentences = [s **for** pair **in** sentence\_pairs **for** s **in** pair]

embeddings = model.encode(sentences, convert\_to\_tensor=True)

results = []

**for** i **in** range(0, len(embeddings), 2):

embedding1 = embeddings[i]

embedding2 = embeddings[i+1]

similarity = util.pytorch\_cos\_sim(embedding1, embedding2)

results.append((sentence\_pairs[i//2][0], sentence\_pairs[i//2][1], similarity.item()))

**return** results

*# Load the model*

model = SentenceTransformer('all-MiniLM-L6-v2')

*# Define your sentence pairs*

sentence\_pairs = [

("The apple is red and juicy.", "The arch is red and just."),

("I love programming in Python.", "Coding in Python is my favorite."),

("The cat sat on the mat.", "A feline rested on the rug."),

*# Add more pairs as needed*

]

*# Process the sentence pairs*

results = compare\_sentences(model, sentence\_pairs)

*# Print results to console*

**for** sentence1, sentence2, similarity **in** results:

**print**(f"Sentence 1: {sentence1}")

**print**(f"Sentence 2: {sentence2}")

**print**(f"Similarity score: {similarity:.4f}")

**print**("-" \* 50)

*# Optionally, save results to a CSV file*

**with** open('sentence\_similarities.csv', 'w', newline='', encoding='utf-8') **as** file:

writer = csv.writer(file)

writer.writerow(['Sentence 1', 'Sentence 2', 'Similarity Score'])

writer.writerows(results)

**print**("Results have been saved to 'sentence\_similarities.csv'")

**To calculate PERPLEXITY (ability to predict words from context)**

Perplexity is a common metric used to evaluate language models in natural language processing (NLP). Here are the key points about perplexity scores:

1. Definition: Perplexity measures how well a language model predicts a sample of text. It quantifies the model's "surprise" or uncertainty when encountering new data
2. Calculation: Mathematically, perplexity is calculated as the inverse of the geometric mean of the probability distribution over all possible outputs for a given input
3. Interpretation:
   * Lower perplexity scores indicate better predictive performance
   * A perplexity score of 1 represents perfect prediction, while higher scores suggest poorer performance
4. Usage:

* Evaluates a language model's ability to predict the next word or character based on previous context
* Used to compare different language models
* Helps in identifying problems in datasets or fine-tuning model parameters

In summary, perplexity is a valuable metric for preliminary evaluation of language models, providing insights into their predictive capabilities. However, it should be used in conjunction with other metrics for a comprehensive assessment of model performance.

To calculate perplexity using a language model, you can use the following approach with the Hugging Face Transformers library:

python

**import** torch

**from** transformers **import** GPT2LMHeadModel, GPT2TokenizerFast

**import** math

**def** calculate\_perplexity(sentence, model, tokenizer, device='cuda'):

*# Tokenize input sentence*

inputs = tokenizer(sentence, return\_tensors='pt').to(device)

*# Get the input IDs*

input\_ids = inputs.input\_ids

*# Calculate perplexity*

**with** torch.no\_grad():

outputs = model(input\_ids, labels=input\_ids)

loss = outputs.loss

*# Perplexity is the exponentiation of the loss*

perplexity = math.exp(loss.item())

**return** perplexity

*# Load pre-trained model and tokenizer*

model\_id = "gpt2" *# You can use other models like "gpt2-medium", "gpt2-large", etc.*

model = GPT2LMHeadModel.from\_pretrained(model\_id).to('cuda')

tokenizer = GPT2TokenizerFast.from\_pretrained(model\_id)

*# Example usage*

sentence = "The quick brown fox jumps over the lazy dog."

perplexity = calculate\_perplexity(sentence, model, tokenizer)

**print**(f"Perplexity: {perplexity:.2f}")

This code does the following:

1. We define a calculate\_perplexity function that takes a sentence, model, and tokenizer as input.
2. Inside the function, we tokenize the input sentence and get the input IDs.
3. We use the model to calculate the loss for the input sentence. The model calculates the loss by trying to predict each token given the previous tokens.
4. Perplexity is calculated as the exponential of the loss.
5. We load a pre-trained GPT-2 model and its tokenizer. You can change model\_id to use different versions of GPT-2 or other language models.
6. Finally, we use an example sentence to demonstrate how to use the function.

Remember that perplexity is highly dependent on the specific model and tokenizer used. Lower perplexity indicates that the model finds the sentence more probable or "less surprising." To calculate perplexity for multiple sentences efficiently, you can modify the function to accept a list of sentences and process them in batches:

python

**def** calculate\_batch\_perplexity(sentences, model, tokenizer, device='cuda', batch\_size=8):

perplexities = []

**for** i **in** range(0, len(sentences), batch\_size):

batch = sentences[i:i+batch\_size]

inputs = tokenizer(batch, return\_tensors='pt', padding=True, truncation=True).to(device)

input\_ids = inputs.input\_ids

**with** torch.no\_grad():

outputs = model(input\_ids, labels=input\_ids)

loss = outputs.loss

perplexity = math.exp(loss.item())

perplexities.extend([perplexity] \* len(batch))

**return** perplexities

*# Example usage for multiple sentences*

sentences = [

"The quick brown fox jumps over the lazy dog.",

"I love programming in Python.",

"Natural language processing is fascinating."

]

perplexities = calculate\_batch\_perplexity(sentences, model, tokenizer)

**for** sentence, perplexity **in** zip(sentences, perplexities):

**print**(f"Sentence: {sentence}")

**print**(f"Perplexity: {perplexity:.2f}")

**print**()

This batch processing approach will be more efficient when dealing with a large number of sentences.