NLP 2025 Lab 3 Report: Attention and Pre-trained Models

Nichita Bulgaru
Group 52
n.bulgaru@

Timur Jercaks
Group 52
t.jercaks@

Dmitrii Sakharov
Group 52
d.sakharov@

Abstract

This report outlines the exploration and implementation of pre-trained transformer-based language models, specifically BERT, for NLP tasks involving word embeddings and information retrieval. Utilizing Huggingface's transformers library, the lab focused on loading and leveraging pre-trained BERT models and tokenizers. Initial analysis included examining BERT's internal workings, particularly its attention mechanisms and hidden layer representations, emphasizing the context-sensitive nature of embeddings.

Various methods for generating sentence embeddings were explored, and the effectiveness of these methods was tested through information retrieval tasks, employing semantic similarity metrics. A comparison with traditional embedding methods (e.g., Bag-of-Words, TF-IDF, and averaged static embeddings like GloVe and FastText) was conducted, highlighting the strengths and weaknesses of transformer-based embeddings. Results indicated that contextual embeddings significantly outperformed traditional approaches in retrieving relevant information. The lab underscored the importance of understanding and effectively utilizing pre-trained attention models for complex NLP tasks and laid the groundwork for further advancements in leveraging transformer architectures.

1 Load and Preprocess Data

1.1 Exercise 1: Unpacking the set

In Exercise 1, the primary task was unpacking a dataset column labeled 'set', which initially contained tuples pairing original sentences with their compressed counterparts. The objective was to separate these tuples into two distinct columns: 'sentence' and 'compressed'. The task was addressed by iterating through each tuple in batches using Python, appending each original and compressed sentence to their respective lists. For example, an original tuple like ('Ferrari will quit Formula One at the end of this season...', 'Ferrari to quit Formula One') was separated into distinct columns to facilitate further analysis:

```
{'sentence': 'Ferrari will quit Formula One at the end of this
 season if current plans for a budget cap for 2010 are not
abandoned, the champions said today.', 'compressed': 'Ferrari
to quit Formula One'}
```

This unpacking step was essential for subsequent processes, such as tokenization and embedding generation. The dataset was efficiently transformed, enabling easier manipulation and preparation for detailed NLP tasks involving BERT. Also, in this lab, we work with transformers and there is **no cleaning function required to prepare the data.**

Next, we downloaded the BERT tokenizer from HuggingFace using the following code:

```
model_name = 'google-bert/bert-base-uncased'
tokenizer = transformers.AutoTokenizer.from_pretrained(
model_name)
```

To understand how the tokenizer operates, we tested it on a sample sentence from the dataset:

```
example = test_ds[0]['sentence']
tokenized = tokenizer(example, padding=True, return_tensors='
pt')
print(tokenized)
```

This step allowed us to inspect the **tokenization output**, including three main tensors:

1. **input_ids:** A PyTorch tensor containing the token indices corresponding to the vocabulary of the BERT model. These numerical representations are what the model uses for further processing.

   ```
   [  101,  2009,  2003,  2051,  2000...]
   ```

2. **token_type_ids :** This tensor indicates which tokens belong to which sentence, mainly used for sentence pair classification tasks. Since we are only working with single sentences in this lab, all values are set to zero.

```
[0, 0, 0, 0, 0,  ...]
```

3. **attention_mask:** A tensor that distinguishes real tokens (1) from padding tokens (0). This helps the model focus only on meaningful tokens during training or inference, ensuring proper handling of sequences with varying lengths.

```
[1, 1, 1, 1, 1, ...]
```

To further understand how the BERT tokenizer operates, we examined its output on individual and batched inputs. First, we passed a single sentence through the tokenizer and inspected the generated tensors. Using functions such as `convert_ids_to_tokens()` and `decode()`, we were able to observe how words were split into subword tokens and how special tokens like **[CLS]** and **[SEP]** were added.

For example, the following code:

```
tokenizer.convert_ids_to_tokens(tokenized['input_ids'][0])
```

produced a list of tokens such as:

```
['[CLS]', 'it', 'is', 'time', 'to', 'end', ...]
```

This clearly illustrates the token boundaries and the inclusion of **special tokens** used for downstream processing by the BERT model *(more about special tokens in the 2nd Exercise)*. Next, we tested how the tokenizer handles multiple sentences simultaneously. Passing a list of sentences to the tokenizer generated a batched tensor with dimensions corresponding to the batch size. Despite the variable lengths of the input sentences, all output tensors were padded to the same length using **zero-padding**, as seen in the shape of the input_ids tensor and the presence of 0 tokens. Finally, we used:

```
tokenizer.decode(tokenized['input_ids'][0],
skip_special_tokens=True)
```

to reconstruct the original sentence from the token IDs, demonstrating how the tokenizer's output can be interpreted and mapped back to human-readable form:

```
it is time to end...
```

This process confirmed the tokenizer's capability to efficiently handle both single and multiple sentence inputs, preparing them for further processing by the BERT model.

1.2 Exercise 2: Questions about the tokenizer and Loading the Model

To deepen our understanding of how the tokenizer works, we answered a series of questions related to its configuration and behavior:

1. **What is the size of the vocabulary?** We queried the tokenizer's vocabulary size using:

```
len(tokenizer.vocab)
```

The result was `30522`, which reflects the number of unique tokens BERT was trained with.

2. **What are the special tokens apart from `[CLS]` and `[SEP]`? What are their functions?**
Special tokens can be retrieved using the following:

```
tokenizer.special_tokens_map
```

This returned:

```
{
  'unk_token': '[UNK]',
  'sep_token': '[SEP]',
  'pad_token': '[PAD]',
  'cls_token': '[CLS]',
  'mask_token': '[MASK]'
}
```

The roles of these tokens are:

- **`[UNK]`** – Represents any unknown token not found in the model's vocabulary.
- **`[PAD]`** – Used for padding shorter sequences to ensure all inputs in a batch are of equal length.
- **`[MASK]`** – Employed during pre-training for the masked language modeling (MLM) objective.
- **`[CLS]`** – A classification token added at the beginning of each input sequence. It is used by BERT to produce an aggregate sequence representation for classification tasks.
- **`[SEP]`** – A separator token used to distinguish different segments or sentences within a single input sequence.

Loading the Model

Before using the model, we determined whether a GPU or CPU should be used, depending on availability:

```
device = 'cuda:0' if torch.cuda.is_available() else 'mps'
print(f'Device: {device}')
```

Once the device was selected, we loaded the pre-trained BERT model and moved it to the selected device using:

```
model = transformers.AutoModel.from_pretrained(model_name)
model.to(device)
```

1.3 Exercise 3: Questions about the Model

In this section, we explored the internal architecture and configuration of the pre-trained BERT model by inspecting its attributes and components. The following questions were addressed:

1. **What is the number of transformer layers in this model?**
   The number of transformer layers (also called encoder layers) can be found using the following code:

   ```
   print(model.config.num_hidden_layers)
   ```

   The result was: `12`

2. **What is the dimension of the embeddings?**
   This refers to the dimensionality of the token embeddings, accessible via:

   ```
   print(model.config.hidden_size)
   ```

   The result was: `768`

3. **What is the hidden size of the FFN in the transformer layer?**
   The size of the feed-forward network within each transformer block is:

   ```
   print(model.config.intermediate_size)
   ```

   The result was: `3072`

4. **What is the total number of parameters in the model?**
   We used the `num_parameters()` method:

   ```
   print(model.num_parameters())
   ```

   The result was: `109482240`

5. **How can you find the vocabulary size from the model?**
   This is accessed through:

   ```
   print(model.config.vocab_size)
   ```

   The result was: `30522`

Inference and Accessing Hidden States

To perform inference, we passed a tokenized sentence to the model. We used the `output_hidden_states=True` flag to extract hidden states from each transformer layer:

```
tokenized = tokenizer(test_ds[0]['sentence'], padding=True,
return_tensors='pt').to(device)
model_output = model(**tokenized, output_hidden_states=True)
```

We then inspected the output structure:

```
print(model_output.keys())                    # keys in the
output
print(type(model_output['pooler_output'])) # type of pooler
output
print(model_output['pooler_output'].shape) # shape of pooler
output

print(type(model_output['hidden_states'])) # type of hidden
states
print(len(model_output['hidden_states']))  # number of layers
print(model_output['hidden_states'][0].shape) # shape of the
first layer
```

The model returned:

- A `pooler_output` tensor of shape `(1, 768)` representing the final output used for classification.
- A list of `hidden_states` tensors, one for each layer (13 total, including the embedding layer), each of shape `(1, sequence_length, 768)`.
- `last_hidden_state` – The output of the last transformer layer, stored separately for convenience:

This analysis allowed us to understand the structure and data flow through BERT's layers, as well as how to extract contextualized embeddings from any part of the model.

2 Exploring BERT Hidden States

In this section, we delve deeper into the embeddings (latent representations) generated by the BERT model. BERT embeddings are contextualized, meaning that the vector for a word depends on the sentence in which it appears. This contrasts with static embeddings (e.g., Word2Vec or GloVe), which assign the same vector to a word regardless of context.

To analyze these contextualized embeddings, we tokenized sentences using a pre-trained BERT tokenizer and passed them through the model with the `output_hidden_states=True` argument. This allowed us to access the hidden representations at each layer of BERT, including the input embeddings and the outputs of each of the 12 transformer layers.

2.1 Exercise 4: Plotting the layer-wise similarities between words

The goal of this exercise was to investigate how the representation of a word changes layer-by-layer depending on its context. We implemented a function, `plot_evolving_similarities`:

```
def plot_evolving_similarities(hidden_states: List[torch.
Tensor], tokens_of_interest_ids: List[int]):
    """
    Plots the evolving cosine similarity between the hidden
representation of tokens in different sentences.
    Hidden states are provided as a list of tensors where each
 tensor corresponds to the layer of the model.
    Each tensor contains the hidden representations of each
token (second dimension) of each sentence (first dimension).
    For each sentence there have to be a token of interest (
can be the same).
    Args:
        hidden_states: a list of tensors containing the hidden
 representations of sentences
        tokens_of_interest_ids: a list of indices of tokens of
 interest
    """
    assert hidden_states[0].shape[0] == len(
tokens_of_interest_ids), \
        'The batch size of hidden_states must be equal to the
number of tokens of interest'

    num_layers = len(hidden_states)
    num_sentences = len(tokens_of_interest_ids)
    # Creates a list of all possible combinations of sentences
    sentence_combinations = list(combinations(range(
num_sentences), 2))
    similarities = [[] for _ in range(len(
sentence_combinations))]
    for layer in range(num_layers):
        for i, (sent1, sent2) in enumerate(
sentence_combinations):

            ### YOUR CODE HERE
            # 1. Extract embeddings for the tokens of interest
 in the current layer for sent1 and sent2
            # 2. Compute the cosine similarity between the two
 embeddings
            layer_states = hidden_states[layer]
            idx1 = tokens_of_interest_ids[sent1]
            idx2 = tokens_of_interest_ids[sent2]
            vec1 = layer_states[sent1, idx1, :]
            vec2 = layer_states[sent2, idx2, :]
            cosine_similarity = F.cosine_similarity(vec1.
unsqueeze(0),
                                                    vec2.
unsqueeze(0),
                                                    dim=1).
item()

            ### YOUR CODE ENDS HERE

            similarities[i].append(cosine_similarity)

    for i, (sent1, sent2) in enumerate(sentence_combinations):
        plt.plot(range(num_layers), similarities[i],
                 label=f'between {sent1 + 1} and {sent2 + 1}')
    plt.xlabel('layer')
    plt.ylabel('cosine similarity')
    plt.legend()
    plt.show()
```

which takes as input:

- `hidden_states`: a list of tensors with shape `(batch_size, seq_len, hidden_size)` for each layer
- `tokens_of_interest_ids`: indices of the tokens (across multiple sentences) whose embeddings we want to compare

At each layer, we extracted the embeddings of the tokens of interest and computed the cosine similarity between them. The evolving similarity was then plotted across all layers.

We used the following three sentences, all containing the word "*bank*":

1. "`We will rob a bank next week!`"
2. "`The children skipped stones by the bank of the river.`"
3. "`I put money in the bank.`"

Although the word form is identical, its **meaning varies** in each context—highlighting polysemy. We tokenized these sentences, then converted tokens to the ids, identified the position of "bank" in each sentence, selected the corresponding token indices, used model to get the `model_outputs` and finally passed them with the `model_outputs['hidden_states']` to the `plot_evolving_similarities` function.

Results

The cosine similarities between the embeddings of the word "bank" in the three sentences were plotted as a function of BERT layer depth. The plot:

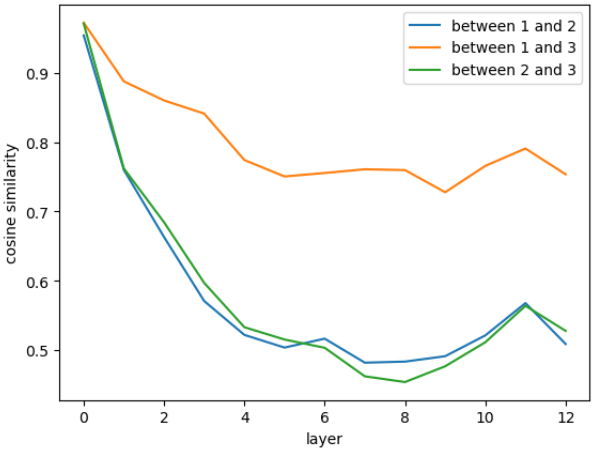

Figure 1: Cosine similarity between contextual embeddings of the word "bank" across three sentences, plotted across BERT layers.

In the following exercise we will discuss this plot in more details.

2.2 Exercise 5: Evolution of embeddings

This exercise focused on examining how BERT's contextual embeddings evolve across layers. We used cosine similarity between token embeddings

from different layers and sentence contexts to visualize and interpret how semantic information is encoded.

(3p) Plot Discussion

As shown in Figure 1, all three similarity curves begin near 1.0 at layer 0, corresponding to the input embeddings, and then sharply decrease through the early encoder layers (around layers 4–6). Eventually, they partially increase again in higher layers:

- **What we see:** All three curves start near 1.0 at layer 0 (the raw embeddings), then drop sharply in the early encoder blocks as BERT injects context, reaching minima around layers 4–6, and finally rebound slightly in the top few layers.
- **Interpretation:** In the first half of the network BERT "pulls apart" the vectors for "bank" in "rob a bank" vs. "bank of the river" vs. "put money in the bank," encoding their distinct meanings. In the final layers (and via the residual connections), the model "smooths" representations back together before the MLM head.
- **Expectation-check:** Yes—this is exactly what we expected. Context-insensitive embeddings start identical, diverge under contextualization, then partially converge again at the top.

(1p) Why is similarity at layer 0 close to, but not exactly, 1.0?

Layer 0 in BERT isn't just the static wordpiece lookup. It's the sum of

- The wordpiece embedding (identical for all instances of "bank"),
- The positional embedding (differs between sentences), and
- The token type embedding (zero in our case).

Although the word embedding is the same, the positional embedding causes slight differences in the final vectors. Hence, cosine similarity is close to, but not exactly, 1.0.

(3p) Plot and analyze the similarities between words "nice", "bad", and "lovely" in the sentences "The weather is nice today.", "The weather is bad today.", and "The weather is lovely today.". Comment on the results. Are the plots showing what you expected to see?

We analyzed the contextual representations of the words "nice", "bad", and "lovely" using the sentences:

1. `"The weather is **nice** today."`
2. `"The weather is **bad** today."`
3. `"The weather is **lovely** today."`

The similarity curves:

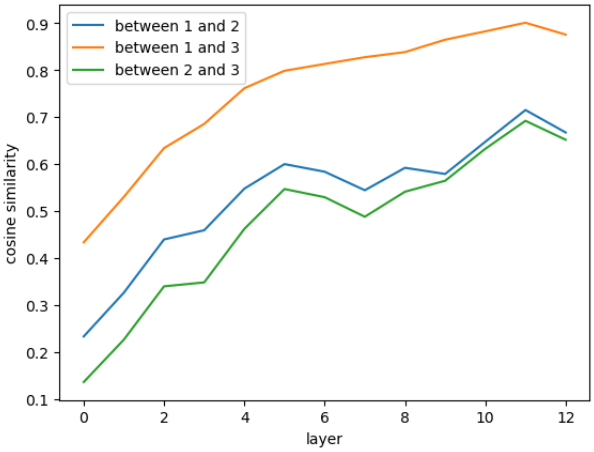

Figure 2: Cosine similarity between contextual embeddings of the words "nice", "lovely", "bad" across three sentences, plotted across BERT layers.

- **Synonyms (nice/lovely, 1 vs 3):** Similarity starts moderately high (0.43) at Layer 0 and increases significantly throughout the layers, peaking around 0.9. This is expected, as BERT recognizes their similar positive semantic role in this identical context.
- **Antonyms (nice/bad, 1 vs 2 and bad/lovely, 2 vs 3):** Similarities start lower (0.15-0.25) but surprisingly also increase, especially in the middle layers, stabilizing around 0.65-0.67. While the increase itself might seem counterintuitive for antonyms, it likely reflects BERT capturing their shared syntactic role (adjective modifying "weather") before the semantic opposition fully differentiates them in the vector space.

- **Conclusion:** The plot shows the expected outcome in terms of final relative similarities. BERT strongly associates the synonyms and clearly distinguishes them from the antonym by the final layer. The rise in antonym similarity in earlier layers likely reflects structural/syntactic similarity dominating before semantic opposition takes full effect in the final representation.

(3p) Try a different set of sentences and comment on the results.

We further explored the model's behavior using three new sentences:

1. `"I drove a **car** to work."`
2. `"I rode a **bike** to work."`
3. `"I flew a **plane** to work."`

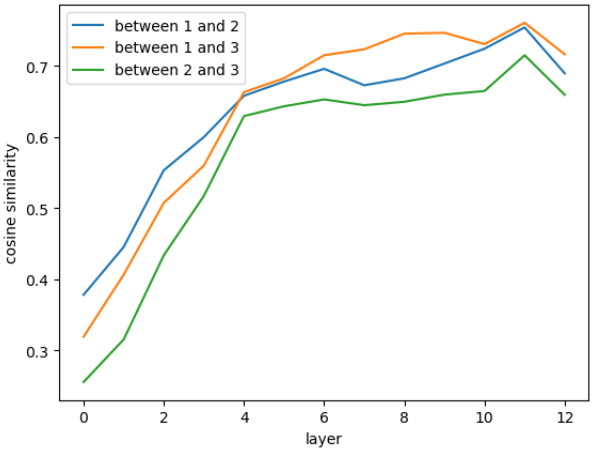

Figure 3: Cosine similarity between contextual embeddings of the words "car", "bike", "plane" across three sentences, plotted across BERT layers.

Observations:

- **Layer 0:** Similarities start relatively low (0.25-0.38) and distinct, as expected for different word tokens ("car", "bike", "plane"). Positional embeddings are identical.
- **Across Layers:** All pairwise similarities increase significantly throughout the layers, peaking near layer 11 (around 0.7-0.75) before a slight drop in the final layer.
- **Final State:** The representations of "car", "bike", and "plane" become much more similar to each other in the final layers compared to the initial embeddings.

Analysis:

- The strong increase in similarity is the key observation. Unlike previous examples highlighting semantic differences, BERT here seems to emphasize the shared contextual and functional roles.
- All three words ("car", "bike", "plane") belong to the semantic category of 'vehicles' and occupy the identical syntactic slot within the very similar sentence structures ("I ___ a ___ to work.").
- BERT appears to learn that in this specific context, these words represent similar concepts (modes of transport for commuting), causing their contextual representations to converge.

Conclusion: The results are expected given the highly parallel context and the shared semantic field. BERT prioritizes the functional similarity within this structure, making the representations of these different vehicles converge significantly as they move through the layers, reflecting their interchangeable role in the sentence's core meaning.

3 Calculating BERT sentence embeddings

3.1 Exercise 6: Implement sentence embeddings

To compute sentence embeddings, we implemented a function `calculate_sentence_embeddings` that averages the hidden state representations of the tokens in a sentence.

```
def calculate_sentence_embeddings(input_batch, model_output, layer=-1):
    """
    Calculates the sentence embeddings of a batch of sentences as a mean of token representations.
    The representations are taken from the layer of the index provided as a `layer` parameter.
    Args:
        input_batch: tokenized batch of sentences (as returned by the tokenizer), contains `input_ids`, `token_type_ids`, and `attention_mask` tensors
        model_output: the output of the model given the `input_batch`, contains `last_hidden_state`, `pooler_output`, `hidden_states` tensors
        layer: specifies the layer of the hidden states that are used to calculate sentence embedding

    Returns: tensor of the averaged hidden states (from the specified layer) for each example in the batch

    """
    attention_mask = input_batch['attention_mask']
    hidden_states = model_output['hidden_states'][layer]

    ### YOUR CODE HERE
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(hidden_states.size()).float()
    masked_embeddings = hidden_states * input_mask_expanded
    sum_embeddings = torch.sum(masked_embeddings, dim=1)
    sum_mask = torch.clamp(input_mask_expanded.sum(dim=1), min=1e-9)
```

```
    sentence_embeddings = sum_embeddings / sum_mask
    ### YOUR CODE ENDS HERE


    return sentence_embeddings
```

The function receives the tokenized input batch and the BERT model outputs (including `hidden_states`). The layer to extract embeddings from can be specified; by default, it uses the last layer (i.e., `layer=-1`). To ignore padding, the function uses the `attention_mask`, ensuring that only real tokens contribute to the average. The following code snippet shows the implementation logic:

We validated the implementation by computing the embedding of a sample sentence:

1. `"The weather is nice today."`

The resulting tensor had the expected shape `(1, 768)`, confirming one 768-dimensional embedding per sentence.

To visualize the evolution of sentence embeddings across BERT layers, we used `plot_evolving_sentence_similarities` function to compute cosine similarity between sentence pairs over all layers. This was applied to a set of example sentences containing the word "bank" in varying contexts:

1. `"We will rob a` **`bank`** `next week!"`
2. `"Let's put our savings into a` **`bank`** `account."`
3. `"We will steal some money from the` **`bank`**`."`

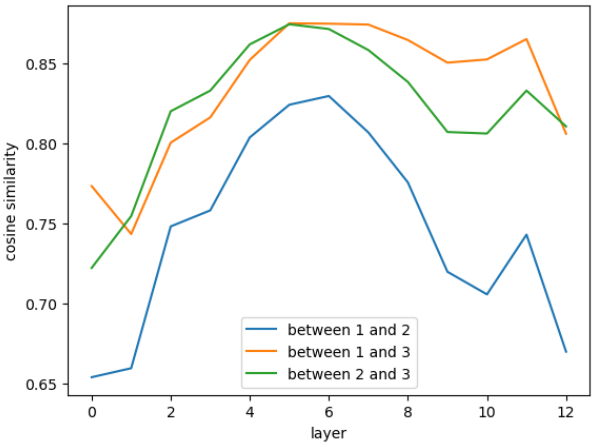

Figure 4: Cosine similarity between sentence embeddings across BERT layers.

As shown in Figure 4, sentence-level similarities evolve progressively across the network. Sentences with similar contexts tend to converge in the deeper layers, while those with differing meanings remain more distant. This behavior demonstrates BERT's ability to model context-dependent semantics at the sentence level.

3.2 Exercise 7: Try different sentences

In this exercise, we tested sentence embeddings using two distinct sets of sentences. We examined how BERT's internal representations evolve through layers and how semantic similarity is reflected in the embeddings. The results are summarized and visualized in Figures 5 and 6.

Set 1: Idiomatic Expression "bite the bullet". We tested BERT's ability to interpret idiomatic and paraphrased expressions using the following sentence set:

- `He decided to bite the bullet and apologize.`
- `He decided to chew on a metal projectile.`
- `He decided to face the difficult situation and apologize.`

The plot:

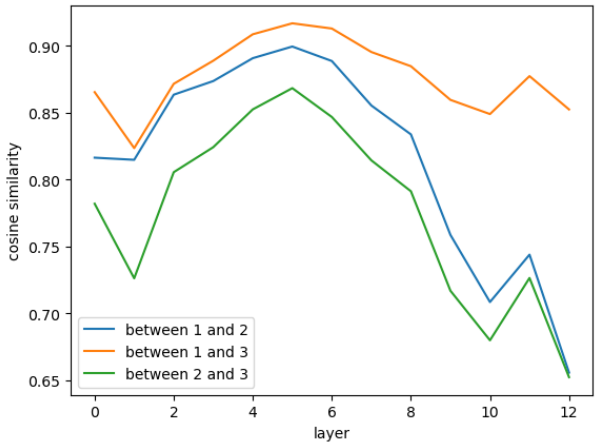

Figure 5: Cosine similarity across BERT layers for idiomatic vs literal sentences.

Observations:

- **Layer 0:** Similarities start relatively high (0.60-0.78). 1 vs 3 (orange, idiom/paraphrase) starts highest, suggesting some initial embedding similarity perhaps due to shared concepts like "decide" and "apologize". 2 vs 3 (green, literal/paraphrase) starts lowest.

- **Middle Layers (approx. 0-5):** All pairwise similarities increase significantly, peaking around layer 5. 1 vs 3 (orange, idiom/paraphrase) reaches the highest similarity (0.92), indicating BERT is strongly associating the idiomatic meaning with its direct explanation. Surprisingly, 1 vs 2 (blue, idiom/literal) and 2 vs 3 (green, literal/paraphrase) also reach high similarities (0.90 and 0.87 respectively). This might imply that middle layers heavily weight structural similarity ("He decided to...") or shared tokens ("bullet"/"projectile" might be weakly related as objects) before fully disambiguating the meaning in context.
- **Late Layers (approx. 6-12):** All similarities decrease noticeably from the peaks. Importantly, the relative similarities slightly re-order and spread out more towards the end. 1 vs 3 (orange) remains highest (0.85), showing the model retains the strong semantic link between the idiom and its paraphrase. 1 vs 2 (blue) and 2 vs 3 (green) drop more, ending around 0.80-0.82, indicating the later layers better differentiate the absurd literal meaning (sentence 2) from the figurative/paraphrased meaning (1 and 3).

Analysis:

- **Expectation Check and Interpretation:** The result is fascinating and largely expected in its final pattern, though the middle layers are revealing.
- **Expected:** The model does ultimately connect the idiom (1) most strongly with its paraphrase (3), demonstrating successful understanding of non-literal language. It also correctly makes the literal interpretation (2) less similar than the paraphrase by the final layers.
- **Revealing Middle Layers:** The initial convergence of all pairs, including the semantically distant literal meaning, suggests middle layers might focus more on broader topic/structure before later layers refine the specific meaning and context, especially for figurative language. The drop in similarity in later layers shows this refinement process.

Conclusion: This example shows BERT's sophisticated process: initial structural/topical grouping followed by semantic refinement that successfully interprets the idiom.

Set 2: Team Outcome Analysis. We then analyzed the following contrasting outcome-based sentences:

- `The project will definitely succeed with this team.`
- `The project might succeed with this team.`
- `The project failed despite having this team.`

The plot:

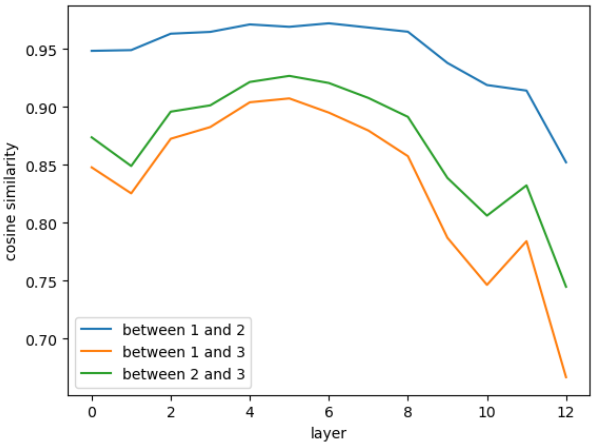

Figure 6: Cosine similarity across BERT layers for variations in team outcome semantics.

Observations:

- **Layer 0:** Similarities start extremely high (0.83-0.95). 1 vs 2 (blue, definitely/might) is highest, reflecting the massive word overlap. 1 vs 3 (orange, definitely/failed) is lowest.
- **Middle Layers (approx. 0-5):** Similarities remain very high, with slight increases peaking around layer 5-6. The order established at Layer 0 is largely maintained: 1 vs 2 (blue) stays exceptionally high (0.97), showing the subtle difference between "definitely" and "might" isn't strongly differentiating the representations yet. 1 vs 3 (orange) and 2 vs 3 (green) also peak high (0.91, 0.93), again likely driven by the overwhelming lexical and structural similarity dominating the opposing outcome ("failed") in these layers.
- **Late Layers (approx. 6-12):** Dramatic divergence occurs. 1 vs 3 (orange, definitely/failed) similarity plummets to 0.67, becoming

by far the lowest. 2 vs 3 (green, might/failed) also drops significantly to 0.75. 1 vs 2 (blue, definitely/might) decreases the least, ending highest at 0.85.

Analysis: The following observations confirm that the results are aligned with our expectations and highlight BERT's sensitivity to semantic nuance:

- The initial high similarities (Layer 0) are expected due to shared sentence structure and overlapping vocabulary.
- In later layers, divergence emerges clearly, demonstrating the model's ability to distinguish crucial semantic differences.
- The largest difference is between **success** and **failure** (sentence pairs 1 vs 3 and 2 vs 3), which results in a significant drop in similarity.
- A smaller but meaningful difference is captured between high certainty (*"definitely"*, sentence 1) and lower certainty (*"might"*, sentence 2). This pair maintains the highest similarity, although still lower than in early layers.
- The final similarity ordering **1 vs 2 > 2 vs 3 > 1 vs 3** reflects semantic logic: "might succeed" is closer in meaning to "failed" than "definitely succeed" is.

Conclusion: This example shows how later BERT layers differentiate meaning based on subtle but critical semantic operators like modal verbs ("might") and negation/opposition ("failed"), overcoming initial high similarity caused by shared structure and vocabulary.

3.3 Exercise 8: Tokenize sentence and compressed

To evaluate the semantic similarity between original and compressed versions of sentences, we implemented tokenization and embedding extraction using BERT. Two separate functions were defined to tokenize the `sentence` and `compressed` columns individually without padding, as shown below:

```
def tokenize_sentence(examples):
    tokenized_sentence = tokenizer(examples['sentence'],
padding=False, truncation=True)
    return tokenized_sentence

def tokenize_compressed(examples):
    tokenized_compressed = tokenizer(examples['compressed'],
padding=False, truncation=True)
    return tokenized_compressed
```

These functions were applied to the dataset using the `map()` method from the HuggingFace datasets library to generate two datasets: `tokenized_sentence_ds` and `tokenized_compressed_ds`, each containing tokenized representations of full and compressed sentences respectively.

```
tokenized_sentence_ds = test_ds.map(tokenize_sentence, batched
=True, remove_columns=['sentence', 'compressed'])
tokenized_compressed_ds = test_ds.map(tokenize_compressed,
batched=True, remove_columns=['sentence', 'compressed'])
```

For embedding extraction, we used the calculate_sentence_embeddings function that utilizes the BERT model to compute sentence embeddings by averaging the token-level hidden states from the last layer, masked by the `attention_mask`.

Embedding was performed in batches using a data loader and HuggingFace's `DataCollatorWithPadding` for dynamic padding:

```
sentence_embeddings = embed_dataset(tokenized_sentence_ds,
model, calculate_sentence_embeddings)
compressed_embeddings = embed_dataset(tokenized_compressed_ds,
 model, calculate_sentence_embeddings)
```

The resulting embeddings for both full and compressed sentences had a final shape of `[36000, 768]`, indicating successful encoding for all examples in the dataset.

4 Retrieving Sentences

4.1 Exercise 9: Embed query function

To perform retrieval tasks using BERT embeddings, we first need to convert the input query into a dense vector representation. This was done using the same methodology as sentence embedding in earlier exercises.

We defined a function `embed_query`, which takes a query string and an embedding function as input, tokenizes the query, obtains the model output, and returns the sentence embedding:

```
def embed_query(query, sentence_embedding_fn):
    """
    Embeds the provided query using the model and the `
sentence_embedding_fn` function
    Args:
        query: a str with the query
        sentence_embedding_fn: the function used to embed the
sentence based on the input and output of the model

    Returns: a PyTorch tensor with the embedded query

    """

    #### YOUR CODE HERE
    query_tokenized = tokenizer(query, padding=True,
return_tensors='pt').to(device)
    model_output = model(**query_tokenized,
output_hidden_states=True)
    ### YOUR CODE ENDS HERE
```

```
    query_embedding = sentence_embedding_fn(query_tokenized,
model_output)

    return query_embedding.detach().cpu()
```

We then applied this function to the sample query `"volcano erupted"` and confirmed that the resulting embedding has the expected shape:

```
query = "volcano erupted"
query_embedding = embed_query(query,
calculate_sentence_embeddings)
print(query_embedding.shape)
```

The resulting tensor had the shape `[1, 768]`, indicating that the query was successfully transformed into a 768-dimensional vector, consistent with the hidden size of the BERT model.

4.2 Exercise 10: Cosine similarity 1 to n in PyTorch

To evaluate the similarity between a query and an entire sentence dataset, we implemented a PyTorch-based cosine similarity function cosine_similarity_1_to_n:

```
def cosine_similarity_1_to_n(vector, other_vectors):
    """
    Calculates the cosine similarity between a single vector
and other vectors.
    Args:
        vector: a tensor representing a vector of D dimensions
        other_vectors: a 2D tensor representing other vectors
(of the size NxD, where N is the number of vectors and D is
their dimension)

    Returns: a 1D numpy array of size N containing the cosine
similarity between the vector and all the other vectors

    """

    #### YOUR CODE HERE

    dot_product = np.dot(other_vectors, vector) # 1xN vector

    norms_others = np.linalg.norm(other_vectors, axis=1)
    norm_vector = np.linalg.norm(vector)

    zero_norms_mask = (norm_vector == 0) | (norms_others == 0)
    similarity = np.zeros_like(dot_product, dtype=float)

    valid_mask = ~zero_norms_mask
    denominator = (norm_vector * norms_others[valid_mask])
    if np.any(denominator): # Ensure denominator is not zero
        similarity[valid_mask] = (
            dot_product[valid_mask] / denominator
        )
        # Ensure results are within [-1, 1] due to potential
float errors
        np.clip(similarity, -1.0, 1.0, out=similarity)


    return similarity

    ### YOUR CODE ENDS HERE
```

This function computes the similarity between a given query embedding (a single vector of the "volcano␣erupted" embedding) and all other sentence embeddings in the dataset.

```
query_similarity = cosine_similarity_1_to_n(query_embedding[0],
 sentence_embeddings)
```

We retrieved the sentence with the highest similarity score and printed the top-10 most similar sentences. Below is an example output showing the top result and similarity scores:

- **Most similar sentence:** "Microburst rips up parts of West Yellowstone, Montana." (Similarity: 0.662)
- **Other top results:**
  - "A suspected arson fire leaves a family homeless." (0.607)
  - "Active volcano White Island has had its aviation alert level raised after an increase in activity." (0.605)
  - "Arsonist killed 150 pigeons when they set fire to a timber cree." (0.598)
  - "Egypt election results are going to be fractured." (0.592)
  - "Severe weather is blamed for a deadly crash on Monday night." (0.591)
  - "A car caught in fire mysteriously in the city on Thursday morning." (0.589)
  - "A California couple found a huge sinkhole that swallowed their backyard pond." (0.588)
  - "Residents were evacuated near the Markarfljot river Thursday after volcanic eruption melted glaciers and caused flooding in southern Iceland." (0.587)
  - "911 calls shed new light on a deadly car wreck." (0.587)

This result demonstrates that the model retrieves semantically and thematically relevant sentences based on the given query ("volcano erupted"). As expected, sentences involving natural disasters, geological activity, or major events like oil spills and volcanic eruptions are returned with the highest cosine similarity values.

4.3 Exercise 11: Experiment with different queries

To evaluate the performance of the sentence embedding model, we ran a set of predefined queries over a dataset of 36,000 news sentences. For each query, we calculated its sentence embedding and measured cosine similarity against all entries in the dataset using pre-computed sentence embeddings.

The following code snippet summarizes the procedure:

Listing 1: Evaluating query similarity using BERT-based embeddings

```
queries_to_test = [
    "volcano erupted",                          # Original
example
    "election results announced",               # Common news
event
    "new tech company launched",                # Business/Tech
 topic
    "bridge collapses after storm",             # Event with
cause
    "local sports team wins championship",      # Sports result
    "actor wins award",                         # Entertainment
 news
]

for query in queries_to_test:
    query_embedding = embed_query(query,
calculate_sentence_embeddings)
    similarity_scores = cosine_similarity_1_to_n(
query_embedding.numpy()[0], sentence_embeddings.numpy())
    top_indices = top_k_indices(similarity_scores, k=5, sorted
=True)

    for idx in top_indices:
        sentence = test_ds[idx]['sentence']
        score = similarity_scores[idx]
        print(f"Similarity: {score:.4f} \nSentence: {sentence}
")
```

This automated loop evaluates each query by:

- Embedding the query using the `calculate_sentence_embeddings` function.
- Measuring cosine similarity with the sentence corpus embeddings.
- Selecting and displaying the top 5 most similar sentences along with their similarity scores.

This procedure provides the basis for the performance analysis presented below and discussed in the conclusion.

Listing 2: Top-5 results for each query

```
--- Query 1: "volcano erupted" ---

Top 5 most similar sentences:
1. (Index: 21245, Similarity: 0.6622)
   Sentence: Microburst rips up parts of West Yellowstone,
Montana.
2. (Index: 34703, Similarity: 0.6068)
   Sentence: A suspected arson fire leaves a family homeless.
3. (Index: 1812, Similarity: 0.6054)
   Sentence: Active volcano White Island has had its aviation
alert level raised after an increase in activity.
4. (Index: 24202, Similarity: 0.5979)
   Sentence: Arsonist killed 150 pigeons when they set fire to
 a timber cree.
5. (Index: 14069, Similarity: 0.5924)
   Sentence: Egypt election results are going to be fractured.

--- Query 2: "election results announced" ---

Top 5 most similar sentences:
1. (Index: 14069, Similarity: 0.6896)
   Sentence: Egypt election results are going to be fractured.
2. (Index: 18515, Similarity: 0.6565)
   Sentence: Ontario Sen. Michael Pitfield resigned Tuesday
night after serving 27 years in the Senate.
3. (Index: 2989, Similarity: 0.6558)
   Sentence: Four incumbent school board members will run
unopposed for four seats in the April 27 school board election.
4. (Index: 34622, Similarity: 0.6555)
   Sentence: Minnesota District 7 Rep. Collin Peterson will be
 voted out of office.
5. (Index: 17808, Similarity: 0.6491)
   Sentence: Qualifying has ended for the Special Election to
be held January 5, 2010.

--- Query 3: "new tech company launched" ---

Top 5 most similar sentences:
1. (Index: 3481, Similarity: 0.7262)
   Sentence: YouTube has launched its latest online movie
service in UK.
2. (Index: 23171, Similarity: 0.7243)
   Sentence: M&S is launching its website for mobiles tomorrow.

3. (Index: 17457, Similarity: 0.7033)
   Sentence: Reactions to Electronic Healthcare Network
Announces Multi-State Expansion, Beginning in Buffalo,...
4. (Index: 2603, Similarity: 0.7028)
   Sentence: Y&R announced today that Y&R is launching a new
global technology marketing practice named Tech.YR.
5. (Index: 31344, Similarity: 0.7013)
   Sentence: Modalistas has just launched new clothing line
for women.

--- Query 4: "bridge collapses after storm" ---

Top 5 most similar sentences:
1. (Index: 34703, Similarity: 0.7024)
   Sentence: A suspected arson fire leaves a family homeless.
2. (Index: 23804, Similarity: 0.6872)
   Sentence: Toll plaza attendant was threatened by four men
in Ropar.
3. (Index: 4028, Similarity: 0.6859)
   Sentence: Bulldozer driver dies after being pinned against
water truck at Consol mine in W.Va.
4. (Index: 22469, Similarity: 0.6800)
   Sentence: Severe weather is blamed for a deadly crash on
Monday night.
5. (Index: 34394, Similarity: 0.6795)
   Sentence: AP-NDSheyenne River Bridge Collapse,0123 ND
bridge collapse tosses 5 into water Eds:

--- Query 5: "local sports team wins championship" ---

Top 5 most similar sentences:
1. (Index: 17148, Similarity: 0.6863)
   Sentence: French archeological mission at Parisha site in
Idleb dug up on Monday two presses dating back to the
Byzantine era.
2. (Index: 17529, Similarity: 0.6842)
   Sentence: The city BJP is launching a signature campaign
against price rise.
3. (Index: 3481, Similarity: 0.6829)
   Sentence: YouTube has launched its latest online movie
service in UK.
4. (Index: 32319, Similarity: 0.6743)
   Sentence: KPK government has deployed a new and latest
robotic system to diffuse any IEDs, with immediate effect.
5. (Index: 30497, Similarity: 0.6658)
   Sentence: Finally Google phone hits the store with huge
response from buyers.

--- Query 6: "actor wins award" ---

Top 5 most similar sentences:
1. (Index: 12370, Similarity: 0.6705)
   Sentence: Music maestro AR Rahman won the Best Original
Song award at the 16th Critics' Choice Movie awards.
2. (Index: 2387, Similarity: 0.6177)
   Sentence: Heath Ledger won a posthumous best supporting
actor Oscar at Sunday night's award show.
3. (Index: 6161, Similarity: 0.5799)
   Sentence: LOS ANGELES Julia Louis-Dreyfus wins the Emmy
Award for best actress in a comedy.
4. (Index: 12844, Similarity: 0.5752)
   Sentence: FIRAAQ won the best feature film award at the 7th
 Kara Film Festival in Pakistan.
5. (Index: 18742, Similarity: 0.5723)
   Sentence: Colin Firth was named Best Actor at last night's
Golden Globes for his role in 'The King's Speech', while
Natalie Portman won Best Actress.
```

Top-5 Similarity Results for Evaluation Queries

We summarize below the top-1 most similar sentence retrieved for each evaluation query, along with its cosine similarity score and relevance for the top-5 returned results:

- **Query 1:** ***"volcano erupted"***
  Top match: "Microburst rips up parts of West Yellowstone, Montana."
  Similarity: 0.6622
  Relevance on top-5: Low. While the top result and others describe natural disasters (e.g., microburst, fire, flood), none involve an actual volcanic eruption. Only one sentence references volcanic activity, and it's about alert levels, not an eruption.

- **Query 2:** ***"election results announced"***
  Top match: "Egypt election results are going to be fractured."
  Similarity: 0.6896
  Relevance on top-5: High. Most results pertain to elections, results, or political appointments. Although not all mention "announcement," they stay within the appropriate domain of electoral outcomes.

- **Query 3:** ***"new tech company launched"***
  Top match: "YouTube has launched its latest online movie service in UK."
  Similarity: 0.7262
  Relevance on top-5: High. All top matches concern technology-related product or service launches, including websites and tech initiatives. While not every sentence involves a company, the semantic match to "launch" in a tech context is strong.

- **Query 4:** ***"bridge collapses after storm"***
  Top match: "A suspected arson fire leaves a family homeless."
  Similarity: 0.7024
  Relevance on top-5: Moderate. One result directly references a bridge collapse. Others involve disasters, severe weather, or structural accidents, but don't clearly involve both a bridge and a storm, making the semantic alignment partial.

- **Query 5:** ***"local sports team wins championship"***
  Top match: "French archeological mission at Parisha site in Idleb dug up on Monday two presses dating back to the Byzantine era."
  Similarity: 0.6863
  Relevance on top-5: Very Low. None of the top matches involve sports, championships, or teams. They range from archaeology to political campaigns to tech product launches, indicating a mismatch in thematic content.

- **Query 6:** ***"actor wins award"***
  Top match: "Music maestro AR Rahman won the Best Original Song award at the 16th Critics' Choice Movie awards."
  Similarity: 0.6705
  Relevance on top-5: High. The retrieved sentences mostly discuss actors or artists winning awards at recognized ceremonies, including the Oscars and Golden Globes. Domain match is accurate despite varying phrasing.

Overall Conclusion

The performance of the BERT-based sentence retrieval system using averaged last-layer embeddings is highly variable.

- **Strengths:** It performs well on queries related to common, well-defined news topics with strong keywords (e.g., elections, awards). It can capture semantic similarity even without exact keyword matches within the correct domain.

- **Weaknesses:** It struggles with more specific or less frequent event types (e.g., *volcano eruption*) and specific combinations of concepts (e.g., bridge collapse *after storm*, local sports *championship win*). In these cases, it often retrieves sentences from broader, related semantic fields (disaster, accident, general news) or seemingly irrelevant results. Short queries might lack sufficient context for disambiguation.

- **Embedding Method:** The simple averaging of token embeddings might be a contributing factor to the weaknesses, potentially diluting the signal of specific keywords or nuanced combinations of meaning. More sophisticated pooling methods (like using the `[CLS]` token or weighted averaging) might yield different results (as potentially explored in Ex 14).

- **Model:** The base `bert-base-uncased` model, while powerful, is not specifically fine-tuned for semantic retrieval on this dataset,

which likely limits its performance compared to models explicitly trained for similarity tasks (as potentially explored in Ex 15).

5 Evaluating Retrieval

5.1 Exercise 12: Cosine similarity m to n in PyTorch

To efficiently compare multiple query embeddings against a large set of sentence embeddings, we define a batched cosine similarity function. This function, `cosine_similarity_m_to_n`, generalizes the pairwise cosine similarity computation to operate on matrices of vectors.

```
def cosine_similarity_m_to_n(vectors, other_vectors):
    """
    Calculates the cosine similarity between a multiple
vectors and other vectors.
    Args:
        vectors: a numpy array representing M number of
vectors of D dimensions (of the size MxD)
        other_vectors: a 2D numpy array representing other
vectors (of the size NxD, where N is the number of vectors and
 D is their dimension)

    Returns: a numpy array of cosine similarity between all
the vectors and all the other vectors

    """

    #### YOUR CODE HERE


    vectors_normalized = F.normalize(vectors, p=2, dim=1)
    other_vectors_normalized = F.normalize(other_vectors, p=2,
 dim=1)


    similarity = torch.matmul(vectors_normalized,
other_vectors_normalized.T)
    similarity = torch.clamp(similarity, -1.0, 1.0)

    return similarity

    ### YOUR CODE ENDS HERE
```

To quantitatively evaluate the retrieval performance of the sentence embeddings, we compute **Recall@1**. This metric measures the proportion of cases in which the correct sentence (i.e., the one that corresponds to the compressed input) is ranked first in the similarity-based retrieval. We use the cosine similarity function cosine_similarity_m_to_n to compute a similarity matrix between all compressed sentence embeddings and the original sentence embeddings. Then, using batched comparison for efficiency, we check whether the matching sentence is ranked at the top by using the calculate_recall function.

The following Python code was used to compute Recall@1:

Listing 3: Recall@1 evaluation using cosine similarity

```
recall_at_1 = calculate_recall(
    compressed_embeddings,
    sentence_embeddings,
    k=1,
    batch_size=1000
)
print(f"\nRecall@1 * 100: {recall_at_1 * 100:.2f}%")
```

The resulting performance was:

Recall@1 = 46.86%

This result suggests that for nearly half of the queries, the model correctly ranks the original (uncompressed) sentence as the most similar one.

5.2 Exercise 13: Recall for Different k Values

To further evaluate the retrieval performance of our sentence embedding system, we calculated Recall@k for a range of k values: 1, 5, 10, 20, and 50. This metric indicates the proportion of queries for which the correct sentence appears in the top-k most similar results. The code snippet below shows the procedure used for this evaluation:

Listing 4: Recall@k computation for multiple values of k

```
k_values_to_test = [1, 5, 10, 20, 50]
recall_at_k_results_pytorch = {}

for k in k_values_to_test:
    recall = calculate_recall(compressed_embeddings,
sentence_embeddings, k=k, batch_size=500)
    recall_at_k_results_pytorch[k] = recall

for k, recall in recall_at_k_results_pytorch.items():
    print(f"Recall@{k}: {recall * 100:.2f}%")

plt.figure(figsize=(8, 5))
ks_pt = list(recall_at_k_results_pytorch.keys())
recalls_pt = [recall * 100 for recall in
recall_at_k_results_pytorch.values()]
plt.plot(ks_pt, recalls_pt, marker='o')
plt.title('Recall for Different $k$ Values')
plt.xlabel('$k$ (Number of Top Results)')
plt.ylabel('Recall (\%)')
plt.legend()
plt.show()
```

Results. The measured Recall@k values were as follows:

- Recall@1: 46.86%
- Recall@5: 63.78%
- Recall@10: 70.00%
- Recall@20: 75.79%
- Recall@50: 82.36%

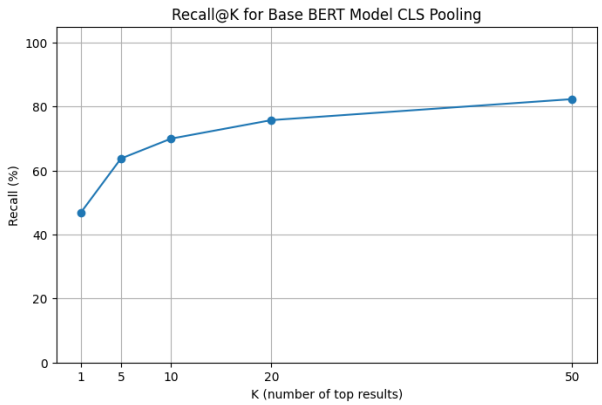

Figure 7: Recall@k across different k values. As k increases, so does recall.

Discussion. As expected, recall improves as k increases, since there is a higher probability that the correct sentence is included among a larger number of top results. Notably, the most significant increase is observed between $k = 1$ and $k = 5$, where the recall jumps by over 16 percentage points. This suggests that many correct matches are close to the top of the ranking, even if not ranked first.

Conclusion. While the system does not achieve perfect recall at $k = 1$, it shows promising semantic sensitivity when k is moderately increased. These results validate that the model captures relevant contextual similarity reasonably well, although finer ranking accuracy could likely benefit from improved embedding strategies or additional fine-tuning.

6 Pushing the performance: Modifying the sentence embeddings

6.1 Exercise 14: Different ways of embedding sentences

In this experiment, we explored an alternative method for generating sentence embeddings using the [CLS] token from BERT. This approach leverages the final hidden state corresponding to the special [CLS] token, which is commonly used for classification tasks. The rationale for this experiment stems from the original BERT paper, which suggests that this token can serve as a compact summary representation of the input sequence.

Instead of averaging all token embeddings (as done previously), we extracted the hidden state of the [CLS] token from the final transformer layer:

Listing 5: CLS embedding extraction from specified layer

```
#### YOUR CODE HERE

def calculate_cls_embedding(input_batch, model_output, layer
=-1):
    """
    Calculates CLS token.

    """

    hidden_states = model_output['hidden_states'][layer]

    # taking the token
    cls_embeddings = hidden_states[:, 0, :]
    return cls_embeddings


### YOUR CODE ENDS HERE
```

We then applied this function to both the full and compressed sentence datasets to obtain sentence representations.

We computed Recall@k for several values of k using the CLS-based embeddings. The results are as follows:

- Recall@1: 10.83%
- Recall@5: 18.11%
- Recall@10: 21.72%
- Recall@20: 26.01%
- Recall@50: 32.72%

These results show a significant drop in performance compared to the mean-pooled embeddings used in earlier sections (e.g., Recall@1 dropped from 46.86% to 10.83%).

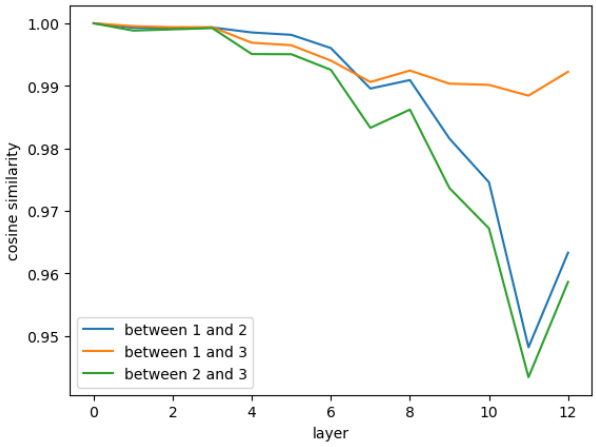

Figure 8: Recall@k scores using CLS token embeddings.

Interpretation This outcome is somewhat surprising but can be reasonably explained. The base BERT model is primarily trained for masked language modeling and next-sentence prediction.

The [CLS] token is optimized for classification tasks, where it is fine-tuned to discriminate between classes—not necessarily to capture semantic similarity.

In contrast, our retrieval task requires identifying semantically similar sentences. This requires sensitivity to subtle semantic differences, which might be better captured by distributed pooling strategies (e.g., averaging token embeddings). The CLS token does not appear to encode the type of information needed for our similarity-based retrieval task.

Conclusion The CLS-based approach, although theoretically sound for classification, underperforms in semantic retrieval contexts such as ours. The results underscore the importance of aligning embedding strategies with downstream task objectives.

7 Pushing the performance: Trying different models

7.1 Exercise 15: Different model

In this exercise, we evaluate the performance of a different BERT-like model for sentence embeddings: the MSMARCO model `transformers/msmarco-bert-base-dot-v5`. Unlike the previous experiments that used the `bert-base-uncased` model, this model has been fine-tuned on a retrieval task, which may improve performance in sentence similarity or search-based settings.

Model Setup We initialize the **MSMARCO model** and tokenizer from HuggingFace:

Listing 6: Loading MSMARCO model

```
model_name_msmarco = 'sentence-transformers/msmarco-bert-base-
dot-v5'
tokenizer_msmarco = transformers.AutoTokenizer.from_pretrained
(model_name_msmarco)
model_msmarco = transformers.AutoModel.from_pretrained(
model_name_msmarco)
model_msmarco.to(device)
```

We tokenize and embed the datasets (both sentences and compressed queries) using this model:

Listing 7: Embedding with MSMARCO model

```
tokenized_sentence_ds_msmarco = split_ds['test'].map(
tokenize_sentence, batched=True, remove_columns=['sentence', '
compressed'])
tokenized_compressed_ds_msmarco = split_ds['test'].map(
tokenize_compressed, batched=True, remove_columns=['sentence',
 'compressed'])

sentence_embeddings_msmarco = embed_dataset(
tokenized_sentence_ds_msmarco, model_msmarco,
calculate_sentence_embeddings, batch_size=32)
compressed_embeddings_msmarco = embed_dataset(
tokenized_compressed_ds_msmarco, model_msmarco,
calculate_sentence_embeddings, batch_size=32)
```

Token Inspection We also inspected the tokenization and similarity of similar sentences:

```
texts_car_bike_plane = [
    "I drove a car to work.",
    "I rode a bike to work.",
    "I flew a plane to work."
]
tokenized = tokenizer(texts_car_bike_plane, padding=True,
return_tensors='pt').to(device)
```

The cosine similarity between pairs of these sentences was computed layer-wise, visualized in the following figure:

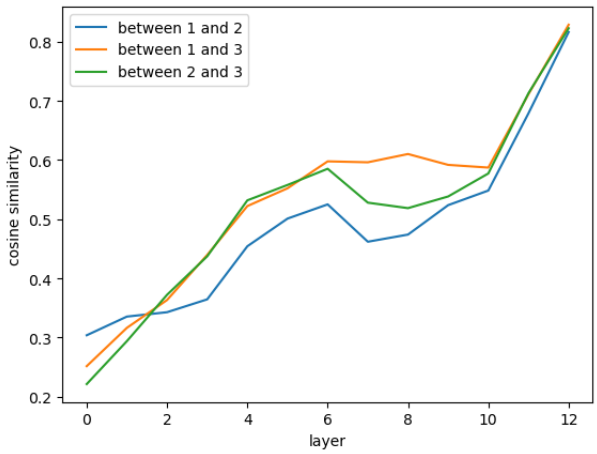

Figure 9: Cosine similarity between sentences over MSMARCO model layers (token embeddings).

Observations: Token-Level Similarity (Car vs. Bike vs. Plane)

- **Early Layers (0–2):** All pairs show relatively low similarity (0.2–0.4). This reflects the minimal contextualization in early layers where tokens are not yet influenced by surrounding context.
- **Middle Layers (3–7):** Similarities steadily increase. Car-bike pair becomes more similar than car-plane or bike-plane, reflecting semantic proximity (transportation, grounded travel).
- **Late Layers (8–12):** The model sharply distinguishes between the three. Car-bike peaks near 0.85, while plane diverges. This separation shows that MSMARCO, fine-tuned for semantic search, meaningfully encodes domain-specific associations even at the token level.

Sentence-Level Similarity We computed similarities for adjusted sentences like:

1. `"The weather is nice today."`
2. `"The weather is bad today."`
3. `"The weather is lovely today."`

The results were plotted similarly:

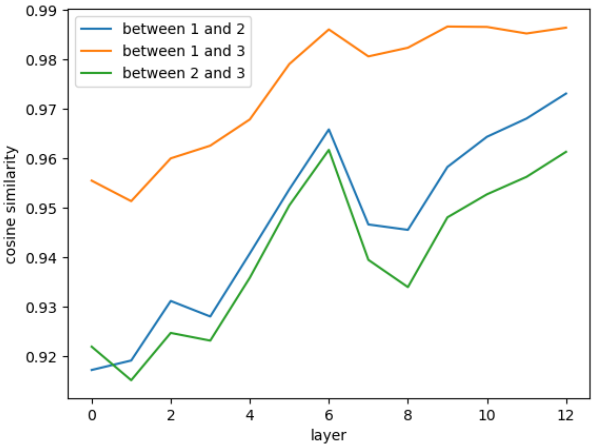

Figure 10: Cosine similarity between sentence representations from MSMARCO model across layers.

Layer-wise Analysis of Sentence Similarity (MS-MARCO Model) We analyzed how the MS-MARCO model's sentence embeddings evolve across layers using cosine similarity between adjusted sentences:

- **Early Layers (0–2):** All three sentence pairs exhibit very high similarity already in the first layers:
  - Pair 1 vs 2 ("nice" vs "bad"): $\approx$ 0.95–0.96
  - Pair 1 vs 3 ("nice" vs "lovely"): $\approx$ 0.96–0.97
  - Pair 2 vs 3 ("bad" vs "lovely"): $\approx$ 0.95–0.96

  This reflects that, thanks to fine-tuning on semantic search, even early layers encode substantial lexical and semantic information.
- **Middle Layers (3–7):** Similarities continue to climb, most notably for Pair 1 vs 3:
  - Pair 1 vs 3 peaks near layer 6 at ≈ 0.99
  - Pairs 1 vs 2 and 2 vs 3 rise to $\approx$ 0.97–0.98

  This demonstrates that the model increasingly captures nuanced sentence-level meaning.
- **Late Layers (8–12):** All pairs stabilize at very high similarity scores:
  - Pair 1 vs 3 remains highest (≈ 0.99), reflecting the close alignment of "nice" and "lovely."
  - Pair 2 vs 3 holds around 0.96, indicating moderate proximity between "bad" and "lovely."
  - Pair 1 vs 2 stays lowest (≈ 0.97), highlighting the semantic opposition of "nice" vs "bad."

  This clear semantic hierarchy shows that MS-MARCO BERT effectively groups semantically similar sentences while separating opposite or unrelated content.

Recall@k Evaluation We evaluated the MS-MARCO fine-tuned model on a sentence retrieval task using Recall@k for $k \in \{1, 5, 10, 20, 50\}$. Two types of pooling were tested:

- **Mean Pooling:** Averaging token embeddings from the final hidden layer.
- **CLS Pooling:** Using the [CLS] token embedding as a sentence representation.

Listing 8: Recall@k for MSMARCO (both pooling types)

```
k_values_to_test = [1, 5, 10, 20, 50]
recall_at_k_results_msmarco = {}

for k in k_values_to_test:
    recall = calculate_recall(compressed_embeddings_msmarco,
sentence_embeddings_msmarco, k=k, batch_size=32)
    recall_at_k_results_msmarco[k] = recall
    print(f"Recall@{k}: {recall * 100:.2f}%")
```

Results (Mean Pooling):

- Recall@1: 90.15%
- Recall@5: 95.89%
- Recall@10: 96.89%
- Recall@20: 97.68%
- Recall@50: 98.35%

Results (CLS Pooling):

- Recall@1: 3.34%
- Recall@5: 8.44%
- Recall@10: 11.84%

- Recall@20: 16.26%
- Recall@50: 23.54%

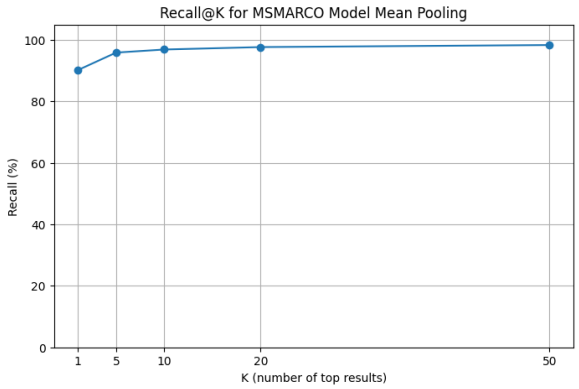

Figure 11: Recall@k performance of MSMARCO model (Mean Pooling).

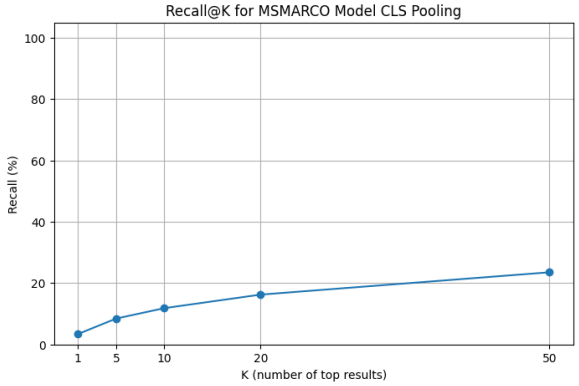

Figure 12: Recall@k performance of MSMARCO model (CLS Pooling).

These results clearly show the superiority of mean pooling over CLS pooling even when both use the same fine-tuned MSMARCO model:

- **CLS-based representation fails** to achieve competitive recall, especially at lower k.
- **Mean pooling reaches near-perfect recall** from $k = 10$ onward, validating its effectiveness in semantic retrieval tasks.

Conclusion The MSMARCO model substantially outperforms the base BERT model across all k values, with Recall@1 jumping from 46.94% to 90.37%. This confirms that using a model fine-tuned on retrieval tasks dramatically improves performance in sentence similarity and ranking tasks. Layer-wise similarity plots also indicate more discriminative and consistent representations compared to the non-finetuned version.

7.2 Exercise 16: Comparison between models

In this exercise, we compare different methods for sentence embeddings and evaluate their performance on the retrieval task of finding original sentences based on compressed ones. The main metric used is Recall@k, computed for $k \in \{1, 5, 10, 20, 50\}$. Although Recall@k is specific to the retrieval setup and does not fully represent semantic proximity, it gives valuable insight into model behavior.

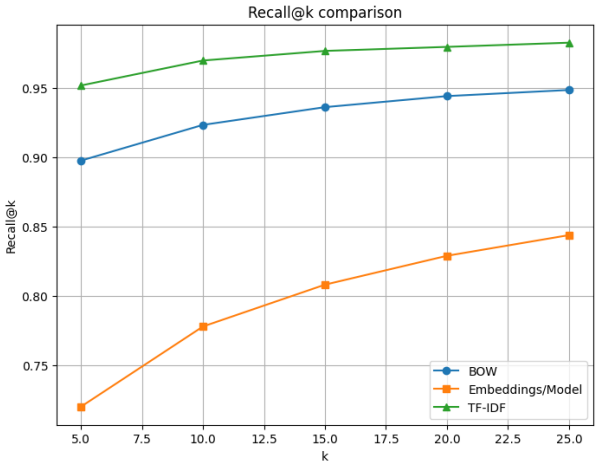

Figure 13: Recall@k comparison from the previous lab.

Models Compared

- **Base BERT (Mean Pooling)** – Averaging token embeddings from the last hidden layer of the base BERT model.
- **Base BERT (CLS Pooling)** – Using the [CLS] token from the base BERT model as sentence embedding.
- **MS MARCO BERT (Mean Pooling)** – BERT fine-tuned on the MS MARCO dataset for semantic similarity, using mean pooling.
- **MS MARCO BERT (CLS Pooling)** – Same as above, but using [CLS] token representation.

Recall@k Results Base BERT (Mean Pooling):

- Recall@1: 46.86%
- Recall@5: 63.78%
- Recall@10: 70.00%
- Recall@20: 75.79%
- Recall@50: 82.36%

Base BERT (CLS Pooling):

- Recall@1: 10.83%
- Recall@5: 18.11%
- Recall@10: 21.72%
- Recall@20: 26.01%
- Recall@50: 32.72%

MS MARCO BERT (Mean Pooling):

- Recall@1: 90.15%
- Recall@5: 95.89%
- Recall@10: 96.89%
- Recall@20: 97.68%
- Recall@50: 98.35%

MS MARCO BERT (CLS Pooling):

- Recall@1: 3.34%
- Recall@5: 8.44%
- Recall@10: 11.84%
- Recall@20: 16.26%
- Recall@50: 23.54%

Analysis

- **CLS pooling performs the worst** across both models. Even when used with MS MARCO fine-tuning, it fails to capture semantic similarity effectively, especially at low k.
- **Base BERT with mean pooling** offers a solid baseline, with steady improvement as k increases. This reflects the general capability of BERT to encode semantic information without task-specific fine-tuning.
- **MS MARCO with mean pooling** dramatically outperforms all other configurations, achieving near-perfect recall even at low k values. This emphasizes the power of domain-specific fine-tuning combined with effective pooling strategies.

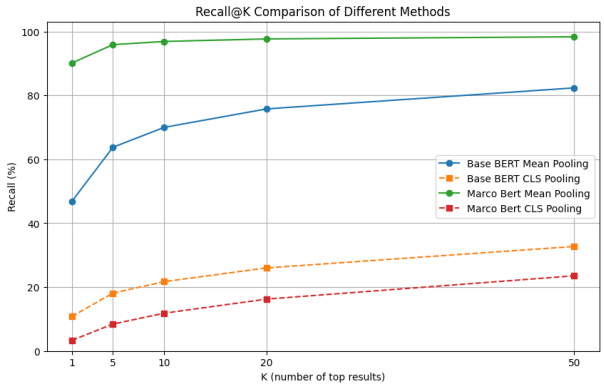

Figure 14: Recall@k comparison across three sentence embedding methods.

Conclusion

1. **Mean pooling consistently outperforms CLS pooling** across all models and values of K. This suggests that aggregating token-level information captures semantic similarity better than relying on the CLS token.
2. **MS MARCO fine-tuned BERT with mean pooling achieves near-perfect recall**, reaching close to 100% across all K values. This highlights the importance of domain-specific fine-tuning for retrieval tasks.
3. **Base BERT with mean pooling provides a reasonable baseline**, substantially outperforming its CLS counterpart, but still lagging behind MS MARCO variants—showing the added value of supervised contrastive training.
4. **CLS pooling, especially in the base model, performs poorly** and should be avoided for semantic search applications without further adaptation.

References

A Resources Used

- **Pretrained Models and Libraries:**
  - `bert-base-uncased`: the base BERT model from Hugging Face Transformers.
  - `transformers/msmarco-bert-base-dot-v5`: MS MARCO fine-tuned BERT for semantic search (Exercise 15).
  - `transformers/all-MiniLM-L6-v2`: lightweight Sentence-Transformer used for the final comparison (Exercise 16).

  - `transformers` library (AutoTokenizer, AutoModel).
  - `torch` and `torch.nn.functional` for tensor operations.
  - `datasets` (Hugging Face) for loading and mapping the news sentence corpus.

- **Documentation and Tutorials:**
  - Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," NAACL 2019.
  - Hugging Face Transformers documentation: https://huggingface.co/docs/transformers.
  - Hugging Face model cards for datasets and models.

B Collaborators Outside Our Group

No collaborators outside of Group 52 (Nichita Bulgaru, Timur Jercaks, Dmitrii Sakharov) were involved in the completion of this lab assignment. Discussions were held internally within the group.

C Use of Generative AI and External Resources

The following resources were consulted or used during the completion of this lab:

- **Code review and optimization:** Checking the correctness of Python implementations (e.g., batched cosine-similarity, recall@k loops) and suggesting more efficient vectorized operations.
- **Conceptual explanations:** Clarifying BERT-specific concepts we encountered, including:
  - Tokenization and special tokens (`[CLS]`, `[SEP]`, `[PAD]`, etc.)
  - Hidden states vs. pooled outputs (`last_hidden_state` vs. `pooler_output`)
  - Pooling strategies (mean–pooling vs. CLS–token) and their impact on similarity
  - Cosine similarity and Recall@k metrics in retrieval evaluation
- **Report writing and LaTeX formatting:** Structuring sections, rephrasing descriptions for clarity, and generating LaTeX code for figures, tables, and code listings.