Probing Language Model Representations

In this notebook, you will explore how much information language models have about linguistic structure even when they have not been explicitly trained to predict it. You will use the encoder language model BERT.

This is a kind of experiment called "probing", where we use internal representations from a language model to predict certain information we have but the language model does not. In particular, we will use a named entity recognition (NER) task, `BIO` tags on each word for the classes person, location, organization, and miscellaneous. The base BERT model did not see any of these labels in training—although BERT has often been fine-tuned on token labeling tasks. For more on token classification for named entity recognition, and for some of the code we use here, see this huggingface tutorial.

Work through the notebook and complete the cells marked TODO to set up and run these experiments.

We start by installing the huggingface `transformers` and related libraries.

```
!pip install transformers datasets evaluate seqeval
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.12/dist-packages (4.57.1)
Requirement already satisfied: datasets in /usr/local/lib/python3.12/dist-packages (4.0.0)
Collecting evaluate
  Downloading evaluate-0.4.6-py3-none-any.whl.metadata (9.5 kB)
Collecting seqeval
  Downloading seqeval-1.2.2.tar.gz (43 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 43.6/43.6 kB 4.8 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from transformers) (3.20.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.34.0 in /usr/local/lib/python3.12/dist-packages (from transformers
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.12/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (25.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.12/dist-packages (from transformers) (6.0.3)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.12/dist-packages (from transformers) (2024.11.
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from transformers) (2.32.4)
Requirement already satisfied: tokenizers<=0.23.0,>=0.22.0 in /usr/local/lib/python3.12/dist-packages (from transformers)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.12/dist-packages (from transformers) (0.6.2)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.12/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: pyarrow>=15.0.0 in /usr/local/lib/python3.12/dist-packages (from datasets) (18.1.0)
Requirement already satisfied: dill<0.3.9,>=0.3.0 in /usr/local/lib/python3.12/dist-packages (from datasets) (0.3.8)
Requirement already satisfied: pandas in /usr/local/lib/python3.12/dist-packages (from datasets) (2.2.2)
Requirement already satisfied: xxhash in /usr/local/lib/python3.12/dist-packages (from datasets) (3.6.0)
Requirement already satisfied: multiprocess<0.70.17 in /usr/local/lib/python3.12/dist-packages (from datasets) (0.70.16)
Requirement already satisfied: fsspec<=2025.3.0,>=2023.1.0 in /usr/local/lib/python3.12/dist-packages (from fsspec[http]<
Requirement already satisfied: scikit-learn>=0.21.3 in /usr/local/lib/python3.12/dist-packages (from seqeval) (1.6.1)
Requirement already satisfied: aiohttp!=4.0.0a0,!=4.0.0a1 in /usr/local/lib/python3.12/dist-packages (from fsspec[http]<=
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-hu
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-hub<1.0,
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->transf
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->transformers) (3.1
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->transformers
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->transformers
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn>=0.21.3->seqeva
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn>=0.21.3->seqev
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.12/dist-packages (from scikit-learn>=0.21.3
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.12/dist-packages (from pandas->datasets)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.12/dist-packages (from pandas->datasets) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.12/dist-packages (from pandas->datasets) (2025.2)
Requirement already satisfied: aiohappyeyeballs>=2.5.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,
Requirement already satisfied: aiosignal>=1.4.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0.0
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0.0a1-
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0.
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0.0
Requirement already satisfied: yarl<2.0,>=1.17.0 in /usr/local/lib/python3.12/dist-packages (from aiohttp!=4.0.0a0,!=4.0.
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.12/dist-packages (from python-dateutil>=2.8.2->pandas->
Downloading evaluate-0.4.6-py3-none-any.whl (84 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.1/84.1 kB 9.2 MB/s eta 0:00:00
Building wheels for collected packages: seqeval
  Building wheel for seqeval (setup.py) ... done
  Created wheel for seqeval: filename=seqeval-1.2.2-py3-none-any.whl size=16162 sha256=0f9d5b672142102fca9d05ceb28609999c
  Stored in directory: /root/.cache/pip/wheels/5f/b8/73/0b2c1a76b701a677653dd79ece07cfabd7457989dbfbdcd8d7
Successfully built seqeval
Installing collected packages: seqeval, evaluate
Successfully installed evaluate-0.4.6 seqeval-1.2.2
```

In case you want them later, we'll load the sklearn functions you used for training logistic regression in assignment 2.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_validate, LeaveOneOut, KFold
import numpy as np
```

Then, we'll use the huggingface `datasets` library to download the CoNLL (Conference on Natural Language Learning) 2003 data for named-entity recognition.

```
from datasets import load_dataset
conll2003 = load_dataset("hgissbkh/conll2003-en")
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens),
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
README.md: 100%  776/776 [00:00<00:00, 13.3kB/s]
data/train-00000-of-00001.parquet: 100%  838k/838k [00:01<00:00, 568kB/s]
data/validation-00000-of-00001.parquet: 100%  212k/212k [00:00<00:00, 52.1kB/s]
data/test-00000-of-00001.parquet: 100%  192k/192k [00:00<00:00, 351kB/s]
Generating train split: 100%  14042/14042 [00:00<00:00, 8977.52 examples/s]
Generating validation split: 100%  3252/3252 [00:00<00:00, 62009.30 examples/s]
Generating test split: 100%  3454/3454 [00:00<00:00, 68387.76 examples/s]
```

To keep things simple, we'll work with a sample of 1000 sentences.

```
sample = conll2003['train'].select(range(1000))
```

Each record contains a list of word tokens and a list of NER labels. For efficiency, the labels have been turned into integers, which makes them hard to interpret.

```
sample[0]
```

```
{'words': ['EU',
  'rejects',
  'German',
  'call',
  'to',
  'boycott',
  'British',
  'lamb',
  '.'],
 'ner': [4, 0, 8, 0, 0, 0, 8, 0, 0]}
```

Fortunately, the dataset object also contains information to map these integers back to readable strings. We can see tags such as `B-PER` (the beginning token of a personal name), `I-PER` (the following tokens inside a personal name, if any), and `O` (a token outside any named entities). We create two dictionaries `id2label` and `label2id` to make mapping between integers and labels easier.

```
labels = sample.features['ner'].feature.names
id2label = {i: label for i, label in enumerate(labels)}
label2id = {label: i for i, label in enumerate(labels)}
print(labels)
print(id2label)
```

```
['O', 'B-PER', 'I-PER', 'B-ORG', 'I-ORG', 'B-LOC', 'I-LOC', 'B-MISC', 'I-MISC']
{0: 'O', 1: 'B-PER', 2: 'I-PER', 3: 'B-ORG', 4: 'I-ORG', 5: 'B-LOC', 6: 'I-LOC', 7: 'B-MISC', 8: 'I-MISC'}
```

For a language model to interpret our data properly, we need to tokenize it in the same way as its training data. We download the tokenizer for the `bert-base-cased` model from huggingface.

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

```
tokenizer_config.json: 100%  49.0/49.0 [00:00<00:00, 2.02kB/s]
config.json: 100%  570/570 [00:00<00:00, 41.2kB/s]
vocab.txt: 100%  213k/213k [00:00<00:00, 495kB/s]
tokenizer.json: 100%  436k/436k [00:00<00:00, 2.03MB/s]
```

Let's see what happens when we run the tokenizer on a single sentence. We tell it that our sentence has already been split into words, in this case by the creators of the CoNLL 2003 NER dataset. BERT, like many language models, used **subword tokenization** to keep the size of its vocabulary manageable. The tokenizer turns n words into $m \geq n$ tokens, represented as a list of integer token identifiers. We use the method `convert_ids_to_tokens` to turn these integers back into a string representation.

```
example = sample[10]
tokenized_input = tokenizer(example['words'], is_split_into_words=True)
tokens = tokenizer.convert_ids_to_tokens(tokenized_input['input_ids'])
tokens
```

```
['[CLS]',
 'Spanish',
 'Farm',
 'Minister',
 'Loyola',
 'de',
 'Pa',
 '##la',
 '##cio',
 'had',
 'earlier',
 'accused',
 'Fi',
 '##sch',
 '##ler',
 'at',
 'an',
 'EU',
 'farm',
 'ministers',
 "'",
 'meeting',
 'of',
 'causing',
 'un',
 '##ju',
 '##st',
 '##ified',
 'alarm',
 'through',
 "'",
 'dangerous',
 'general',
 '##isation',
 '.',
 "'",
 '[SEP]']
```

Notice how the name `Palacio` has been split into three subword tokens: `Pa`, `##la`, and `##cio`. The prepended `##` indicates that this token is *not* the start of a word. But the NER annotations we have are at the word level. We thus need to do some work to map the sequence of NER labels, linked to words, to the usually longer sequence of subword tokens. This is a common task when you have data that wasn't created for a particular language model's classification. We adapt a function from the huggingface tutorial to map the NER labels onto the subword tokens. We assign the label -100 to tokens not at the beginning of a word, as well as to the sentinel `[CLS]` and `[SEP]` tokens at the beginning and end of the sentence.

```
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples['words'], truncation=True, is_split_into_words=True)

    labels = []
    for i, label in enumerate(examples['ner']):
        word_ids = tokenized_inputs.word_ids(batch_index=i)  # Map tokens to their respective word.
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:  # Set the special tokens to -100.
            if word_idx is None:
                label_ids.append(-100)
```

```
            elif word_idx != previous_word_idx:  # Only label the first token of a given word.
                label_ids.append(label[word_idx])
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx
        labels.append(label_ids)

    tokenized_inputs['labels'] = labels
    return tokenized_inputs
```

We apply this function to the whole dataset.

```
tokenized_sample = sample.map(tokenize_and_align_labels, batched=True)
tokenized_sample.set_format(type='torch', columns=['input_ids', 'token_type_ids', 'attention_mask', 'labels'])
```

```
Map: 100%  1000/1000 [00:00<00:00, 4470.07 examples/s]
```

Each record in the tokenized sample now has numeric IDs for each token, an attention mask (always 1 in this encoding task), and token-level labels.

```
tokenized_sample[0]
```

```
{'input_ids': tensor([  101,  7270, 22961,  1528,  1840,  1106, 21423,  1418,  2495, 12913,
          119,   102]),
 'token_type_ids': tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]),
 'attention_mask': tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]),
 'labels': tensor([-100,    4,    0,    8,    0,    0,    0,    8,    0, -100,    0, -100])}
```

Now let's load the BERT model itself. We use the version that was trained on data that hadn't been case-folded, since upper-case words might be useful features for NER in English.

```
from transformers import BertModel

model = BertModel.from_pretrained("bert-base-cased", output_hidden_states=True)
```

```
model.safetensors: 100%  436M/436M [00:06<00:00, 71.4MB/s]
```

We run inference on the first sentence in our sample, passing the model the list of token identifiers (coerced into a tensor with a single batch dimension) and the attention mask, which is all 1s for this simple encoding task.

```
import torch
with torch.no_grad():
  outputs = model(input_ids=tokenized_sample[0]['input_ids'].unsqueeze(0), attention_mask=tokenized_sample[0]['attention
  hidden_states = outputs.hidden_states
```

The `hidden_states` object we just created is a tuple with 13 items, one for each layer of the BERT model. The initial token embedding is layer 0 and the output is layer 12. Each layer contains embeddings for each token—here, there are 12—each of which is a vector of length 768.

```
print(len(hidden_states))
print(hidden_states[0].shape)
```

```
13
torch.Size([1, 12, 768])
```

We now define a function to take a dataset of tokens, run it through BERT to produce embeddings at all 13 layers, and to produce features for predicting NER labels from token embeddings. This function uses two explicit nested loops, which is not the fastest way to do things in pytorch, but more clearly expresses what is being computed. It takes about a minute to run on colab. (This assignment isn't meant to be a pytorch tutorial, but if you know pytorch, or are learning it, feel free to speed up this code by batching the examples together.)

```
def compute_layer_representation(data, model, tokenizer):
  rep = []
  lab = []
  for example in data:
    with torch.no_grad():
```

```
        outputs = model(input_ids=example['input_ids'].unsqueeze(0), attention_mask=example['attention_mask'].unsqueeze(0)
        tokens = tokenizer.convert_ids_to_tokens(example['input_ids'])
        hidden_states = outputs.hidden_states
        for i in range(len(example['labels'])):
          if example['labels'][i] != -100:
            lab.append(int(example['labels'][i]))
            rep.append([hidden_states[layer][0][i].numpy() for layer in range(len(hidden_states))])
            #rep.append(hidden_states[layer][0][i].numpy())
    return [np.array(rep), np.array(lab)]
```

We compute embeddings for all layers for the full dataset. Note that the first dimension is now *words* rather then sentences. This means that we can probe the information that each word's embedding has about named entities (or anything else).

```
X, y = compute_layer_representation(tokenized_sample, model, tokenizer)
```

We can select information about the bottom (word embedding) layer, which gives as a matrix of words by embedding dimensions.

```
X[:,0,:].shape
```

```
(12057, 768)
```

TODO: Your first task is to probe the information that these emedding layers have about named entities. Train one linear model for each of the 13 layers of BERT to predict the label of each word in `y` using the embeddings in `X`. Print the accuracy of this model for each of the 13 layers of BERT. By accuracy, we simply mean the proportion of words that have been assigned the correct tag. (Although NER is often evaluated at the level of the entity, which may span one or more words, we will keep things simple here.)

You may use the sklearn code for training logistic regression models that you ran in assignment 2. You may also train these classifiers using pytorch. In any case, perform 10-fold cross validation and return the average accuracy over all ten folds.

```
# TODO: Train linear models to predict the NER labels using embeddings from each layer of BERT.
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score,KFold
from sklearn.metrics import accuracy_score

kf_val = KFold(n_splits= 10,shuffle = True,random_state = 42)
accuracy_layer = []

for layer in range (X.shape[1]):
  X_layer = X[:,layer,:]
  clf_val = LogisticRegression(max_iter=500,solver='liblinear')
  scores = cross_val_score(clf_val,X_layer,y,cv=kf_val,scoring = 'accuracy')
  acc_mean = np.mean(scores)
  accuracy_layer.append(acc_mean)
  print(f'Layer {layer:2d}: Accuracy : {acc_mean:4f}')
```

```
Layer  0: Accuracy : 0.925190
Layer  1: Accuracy : 0.953223
Layer  2: Accuracy : 0.960687
Layer  3: Accuracy : 0.960024
Layer  4: Accuracy : 0.966824
Layer  5: Accuracy : 0.970723
Layer  6: Accuracy : 0.972962
Layer  7: Accuracy : 0.973625
Layer  8: Accuracy : 0.971966
Layer  9: Accuracy : 0.972381
Layer 10: Accuracy : 0.974371
Layer 11: Accuracy : 0.972878
Layer 12: Accuracy : 0.970888
```

The above code loops through all the 13 layers of BERT ,logistic regression is used for training for each and then print the mean accuracy among all the folds.

TODO: How good are these accuracy levels? Since the `O` tag is very common, you can do quite well by always predicting `O`. Compute the baseline accuracy, i.e., the accuracy you would get on the sample data if you always predicted `O`.

```
# TODO: Compute and print the baseline accuracy of always predicting O.
id_O = label2id['O']
accuracy_baseline = np.mean(y == id_O)
print(f"Baseline accuracy always predicting 'O': {accuracy_baseline:.4f}")
```

```
Baseline accuracy always predicting 'O': 0.7650
```

TODO: Now try another probing experiment for capitalized words, a simple feature that, in English, is correlated with named entities. For each word in the sample data, create a feature that indicates whether that word's first character is a capital letter. Then train logistic regression models for each layer of BERT to see how well they predict capitalization. Perform 10-fold cross-validation as above. Note any differences you see with the NER probes.

In addition, compute the baseline accuracy, i.e., the accuracy of always predicting that a word is not capitalized.

```
# TODO: Train linear models to predict capitalization.
# Compute and print the accuracy of these models for each layer of BERT.
# Compute and print the baseline accuracy.

words = [tokenizer.decode(tokenized_sample[i]['input_ids']) for i in range (len(tokenized_sample))]
all_words = [w for s in sample ['words'] for w in s ]
capitalization_labels = np.array([w[0].isupper() for w in all_words]).astype(int)

cap_acc_layer=[]
for layer in range (X.shape[1]):
  clf = LogisticRegression(max_iter=500,solver = 'liblinear')
  scores = cross_val_score(clf,X[:,layer,:],capitalization_labels,cv=kf_val,scoring = 'accuracy')
  mean_acc = np.mean(scores)
  cap_acc_layer.append(mean_acc)
  print(f"Layer {layer:2d} :  Accuracy : {mean_acc:.4f}")


baseline_cap = max(np.mean(capitalization_labels), 1 - np.mean(capitalization_labels))
print(f"\nBaseline accuracy: {baseline_cap:.4f}")
```

```
Layer  0 :  Accuracy : 1.0000
Layer  1 :  Accuracy : 1.0000
Layer  2 :  Accuracy : 1.0000
Layer  3 :  Accuracy : 1.0000
Layer  4 :  Accuracy : 0.9999
Layer  5 :  Accuracy : 0.9998
Layer  6 :  Accuracy : 0.9998
Layer  7 :  Accuracy : 0.9992
Layer  8 :  Accuracy : 0.9991
Layer  9 :  Accuracy : 0.9983
Layer 10 :  Accuracy : 0.9981
Layer 11 :  Accuracy : 0.9974
Layer 12 :  Accuracy : 0.9976

Baseline accuracy (always predicting majority class): 0.6985
```

```
Start coding or generate with AI.
```