Fine tuning of BERT model for labeling job resumes (1st screening)

For the first part of the screening of resumes, we wanted to multilabel the target resume with the job categories. From the research we conducted, we agreed on fine tuning a BERT transformer model, which could yield amazing results, without having to train our own Natural Language processing model or LLM from scratch

To be able to use a model like BERT on text inputs, we would need to embed the text into something it could understand. We therefore chose to use the BERT tokenizer, which is particularly effective for this project because it uses WordPiece tokenization, which allows it to handle out-of-vocabulary words and rare terms by breaking them into subword units. This is especially valuable when processing resumes, which often contain technical terminology. By preserving semantic meaning even in fragmented or unfamiliar words, the BERT tokenizer ensures that important contextual information from resumes is retained and properly interpreted by the model during fine-tuning.

```
from transformers import BertTokenizer, BertForSequenceClassification,
Trainer, TrainingArguments
from sklearn.model_selection import train_test_split
import torch
from torch.utils.data import Dataset
import pandas as pd
import numpy as np
from sklearn.metrics import accuracy_score,
precision_recall_fscore_support


# We use the transformers and torch libraries for fine tuning the BERT
model

# Loading the pickled dataset
import pickle
with open('Data/Dataframes/newDF.pkl', 'rb') as f:
    df = pickle.load(f)

# We drop all the columns that are not the resume texts or onehot
encoded for the specific job categories
trainingDF = df.drop(columns=['ID', 'Label', 'TextLen'])

# We load the tokenizer used to train the bert-base-uncased model
# This tokenizer is used to convert the text into tokens that the BERT
model can understand
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Custom PyTorch Dataset class for handling resume data
# This is the class that will handle the embedding of the text and the
```

```
labels
class ResumeDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=512):
        """
        Initializes the dataset with input texts, their corresponding
labels,
        a tokenizer (e.g., BERT tokenizer), and the maximum token
length.

        Args:
            texts: List of resume texts.
            labels: Multi-label targets for each resume.
            tokenizer: Tokenizer to convert text to model input
format.
            max_len: Maximum token length for each input. Its set at
default to 512 tokens.
        """
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __len__(self):
        """
        Returns the number of samples in the dataset.
        """
        return len(self.texts)

    def __getitem__(self, idx):
        """
        Tokenizes and processes a single sample for the model.

        Args:
            idx: Index of the sample.

        Returns:
            Dict: Dictionary with tokenized input tensors and labels.
        """
        # Tokenize the input text with truncation and padding to max
length
        # The truncation is set to True to ensure that the text is cut
off at the max length
        # This is done to ensure that the model does not receive more
than 512 tokens, which is the maximum length for BERT
        encoding = self.tokenizer(
            self.texts[idx],
            truncation=True,
            padding='max_length',
            max_length=self.max_len,
            return_tensors='pt'  # Return PyTorch tensors
```

```
        )

        # Remove batch dimension from returned tensors
        item = {key: val.squeeze(0) for key, val in encoding.items()}

        # Convert label list to a float tensor, which is required for
multi-label classification
        # The labels are converted to a tensor of floats, as the model
expects the labels to be in this format
        # The labels are the one-hot encoded vectors for the specific
job categories
        item['labels'] = torch.tensor(self.labels[idx],
dtype=torch.float)

        return item
```

As we mentioned earlier, we use 512 as the max length of tokens as input for the model when training on the resumes, since that is the maximum length that Bert can handle. If we had more time, we could convert the resumes into subtexts to feed into Bert and therefore still keep all of the contents.

```
# Splitting the dataset into training and testing sets
# We use 90% of the data for training and 10% for validation
# The texts are the resumes and the labels are the onehot encoded job
categories
# We could have assigned more of the dataset for testing, but we did
not have the time test the training of the model with different test
and training sizes

texts = df['Resume'].tolist()
labels = trainingDF.drop(columns=['Resume']).values

X_train, X_val, y_train, y_val = train_test_split(texts, labels,
test_size=0.1)

train_dataset = ResumeDataset(X_train, y_train, tokenizer)
val_dataset = ResumeDataset(X_val, y_val, tokenizer)

# Loading the pre-trained BERT model for sequence classification
# 'bert-base-uncased' refers to the base BERT model with lowercase
(uncased) inputs
# num_labels is set to the number of output classes, which is the
number of job categories
# problem_type is explicitly defined to guide the model to use a
sigmoid activation function.
model = BertForSequenceClassification.from_pretrained(
    "bert-base-uncased",
    num_labels=labels.shape[1],
    problem_type="multi_label_classification"
)
```

```
Some weights of BertForSequenceClassification were not initialized
from the model checkpoint at bert-base-uncased and are newly
initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

Hyperparameters and settings for training the BERT model:

We set the number of epochs to 3, which is a common choice for fine-tuning BERT. We did not have the time to test the model with more epochs. We are however looking at the loss and accuracy of the model after each epoch, so we can see if the model is overfitting or not.

We set the batch size to 8, which is a common choice for fine-tuning BERT. We started with a batch size of 16, but the model was running out of memory. Although we used the AdamW optimizer, which adapts learning rates per parameter, we did not apply a dynamic learning rate schedule during training. Implementing such a scheduler could further optimize the model during training, but we did not have the time to test it. Lastly we use cuda for training the model if it is available, which reduced the training time from 20 hours to 2 hours.

```
import accelerate
print(accelerate.__version__)
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_dir="./logs",
    logging_steps=10
)


trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset
)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
```

```
1.6.0
```

```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
```

```
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768,
bias=True)
              (key): Linear(in_features=768, out_features=768,
bias=True)
              (value): Linear(in_features=768, out_features=768,
bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768,
bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072,
bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
```

```
  (classifier): Linear(in_features=768, out_features=10, bias=True)
)
```

```
trainer.train()
```

```
<IPython.core.display.HTML object>
```

```
TrainOutput(global_step=9771, training_loss=0.03867846989406366,
metrics={'train_runtime': 8241.5805, 'train_samples_per_second':
9.483, 'train_steps_per_second': 1.186, 'total_flos':
2.0565184708583424e+16, 'train_loss': 0.03867846989406366, 'epoch':
3.0})
```

Training output

The model was fine-tuned over 3 epochs using the BERT architecture. Training completed in approximately 2 hours and 17 minutes, with a final average training loss of 0.0387. As shown in the table below, both training and validation loss decreased steadily, indicating that the model was learning effectively without overfitting.

The final validation loss of 0.0181 suggests that the model generalized well to unseen data, making it suitable for multi-label classification of resumes.

```
# To upload the model, we have to save the tokenizer and the model
from transformers import BertTokenizer

# Load the same tokenizer you used during training
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")

# Save it to the directory where your checkpoint is stored
tokenizer.save_pretrained("./results/checkpoint-9771")  # adjust if
needed
```

```
c:\Users\pelle\Work\1semSoft\exam\AIML-Exam\examVenv\Lib\site-
packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please
update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
('./results/checkpoint-9771\\tokenizer_config.json',
 './results/checkpoint-9771\\special_tokens_map.json',
 './results/checkpoint-9771\\vocab.txt',
 './results/checkpoint-9771\\added_tokens.json')
```

```
# To train the model on an nvidia GPU, you need to download the CUDA
toolkit and install the nvidia drivers.


import torch
print(torch.cuda.is_available())
```

```
print(torch.cuda.get_device_name(0) if torch.cuda.is_available() else
"No GPU found")
```

```
True
NVIDIA GeForce RTX 4060
```

```
# We create a function to compute the metrics for the model

def compute_metrics(eval_pred):
    logits, labels = eval_pred

    # Convert logits to predicted class indices
    preds = np.argmax(logits, axis=1)

    # If labels are one-hot, convert to class indices too
    if labels.ndim > 1 and labels.shape[1] > 1:
        labels = np.argmax(labels, axis=1)

    precision, recall, f1, _ = precision_recall_fscore_support(labels,
preds, average='weighted')
    acc = accuracy_score(labels, preds)

    return {
        'accuracy': acc,
        'precision': precision,
        'recall': recall,
        'f1': f1
    }

# We set the compute_metrics function to the trainer
trainer.compute_metrics = compute_metrics

results = trainer.evaluate()
```

```
c:\Users\pelle\Work\1semSoft\exam\AIML-Exam\examVenv\Lib\site-
packages\sklearn\metrics\_classification.py:1565:
UndefinedMetricWarning: Recall is ill-defined and being set to 0.0 in
labels with no true samples. Use `zero_division` parameter to control
this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is",
len(result))
```

```
print(results)
```

```
{'eval_loss': 0.018065307289361954, 'eval_accuracy':
0.8918825561312608, 'eval_precision': 0.922566883602777,
'eval_recall': 0.8918825561312608, 'eval_f1': 0.9027684731323408,
'eval_runtime': 73.8097, 'eval_samples_per_second': 39.222,
'eval_steps_per_second': 4.905, 'epoch': 3.0}
```

- eval_loss': 0.018065307289361954

- eval_accuracy': 0.8918825561312608
- eval_precision': 0.922566883602777
- eval_recall': 0.8918825561312608
- eval_f1': 0.9027684731323408
- eval_runtime': 73.8097
- eval_samples_per_second': 39.222
- eval_steps_per_second': 4.905
- epoch': 3.0

Evaluation Summary (After Final Epoch)

eval_loss: 0.0181 A low loss value on the validation set suggests that the model has learned the task well and generalizes effectively to unseen data. This is a positive indicator and shows that the model did not overfit during training.

eval_accuracy: 89.19% This is a strong result; however, accuracy alone can be misleading in multi-label classification, where multiple correct labels may exist for each sample. This is the metric that calculates the amount of times BERT was 100% correct on the labeling.

eval_precision: 92.26% This high value indicates that when the model predicts a label, it is correct most of the time. This is especially important in screening tasks, where false positives (irrelevant matches) should be minimized.

eval_recall: 89.19% The model is also successful at capturing most of the correct labels, which means it rarely misses relevant job categories. A high recall is crucial to ensure that no important skills or qualifications are overlooked.

eval_f1: 90.28% The F1 score balances both precision and recall. A score above 90% reflects excellent overall performance and confirms that the model handles the trade-off between false positives and false negatives well.

Evaluation Efficiency

Runtime: around 74 seconds

Samples per second: 39.22

Steps per second: 4.91

These metrics indicate that the model evaluates data quickly and efficiently. Processing nearly 40 samples per second shows that the system would be capable of real-time or batch-mode screening in practical HR applications.

Interpretation

Overall, the model performs very well on the task of resume classification:

High F1 score, precision, and recall show the model is both accurate and comprehensive in identifying relevant job categories.

Low loss and consistent performance over epochs suggest good generalization and stable training.

Fast evaluation speed means the system could scale to large datasets in a real-world scenario.

There are no major signs of overfitting or instability, and the performance is strong across multiple metrics. These results indicate that the model is well-suited for first-round automated resume screening in a multi-label setting.