00_falcon_dnarganes

February 5, 2024

1 Machine Learning for Drug Discovery Research Literature Mining

1.1 By @DavidNarganes-Carlon

We are interested in applications of machine learning to mine the research literature for additional types of entities/classes/answers relevant to drug discovery not already available to Open Targets which include but are not limited to variants, biomarkers, tissues/cell types, adverse events, and assay conditions.

1.2 Tasks

In this, you will need to develop an ML pipeline that involves:

1. Data Collection
2. Data Preprocessing (Visualization)
3. ML Model Development
4. Evaluation
5. Complexity of Models in Production

1.3 Things to consider:

- Please use free Google Colab or Kaggle if you require GPU access.
- This task should not take more than 6 hours to solve.
- Achieving more than 60% F-score/accuracy on the dataset is *not a requirement*, and you are assessed based on the design choices and data science protocols you follow in solving this task.
- Once completed, please share your notebook as a zip file as <*firstname_lastname.zi*>“, including outputs and detailed comments.
- Feedback will be given on your notebook.

2 FalconFrames Environment Setup

This guide simplifies setting up the Python environment and installing necessary libraries for the test.

2.1 Prerequisites

Ensure Python 3.7 or later is installed.

2.2 Environment Setup

1. **Create a Python Virtual Environment:**

   Create a virtual environment named `falconframes_env` in your home directory:

   ```
   python -m venv ~/falconframes_env
   ```

2. **Activate the Virtual Environment:**

   Activate the virtual environment:

   ```
   source ~/falconframes_env/bin/activate
   ```

2.3 Installing Dependencies

Install required libraries within the virtual environment:

```
pip install notebook matplotlib lxml ipywidgets
```

2.4 Jupyter Notebook Extensions

Enable necessary Jupyter Notebook extensions:

```
jupyter nbextension enable --py widgetsnbextension
```

2.5 Installing SciSpacy

Install SciSpacy and its dependencies:

```
pip install scispacy
pip install https://s3-us-west-2.amazonaws.com/ai2-s2-scispacy/releases/v0.5.3/en_core_sci_sm-
```

2.6 Usage

Start using Jupyter Notebook for the FalconFrames project after installation. Activate the virtual environment when working on the project.

2.7 Deactivating the Environment

Deactivate the virtual environment when finished:

```
deactivate
```

3 1. Data Collection

This Python script collects scientific article annotations from Europe PMC. To run the script, execute:

```
!python3 data_collection.py
```

Key Features:

- Utilizes `aiohttp`, `pandas`, and `spacy` libraries.
- Loads the "en_core_sci_sm" Spacy model.

- Asynchronously fetches data by annotation type.
- Processes annotations and creates a Pandas DataFrame.
- Data collection parameters are specified in the `main` function.
- Results are saved as a CSV file.

Run the script to gather scientific article annotations efficiently.

```
[ ]: ! python3 data_collection.py
```

4 2. Data Preprocessing

In the data preprocessing phase, a set of functions in the `utils.py` module significantly streamlined the workflow. These functions collectively addressed the challenge of obtaining complete sentences or paragraphs to enrich the dataset.

Firstly, the `fetch_xml(pmcid)` function asynchronously interacted with the Europe PMC Article API to retrieve full-text XML content using the PMCID as a reference. This content was essential as it contained complete articles with relevant annotations.

The `get_relevant_paragraphs(pmcid, partial_sentences)` function, meanwhile, performed web scraping to parse the XML content. It efficiently extracted paragraphs containing the required annotations, narrowing down the dataset.

Another key function, `segment_sentences_spacy(text)`, utilized SpaCy to break down paragraphs into individual sentences. This step facilitated subsequent analysis by splitting content into manageable units.

Additionally, `get_full_text_xml_paragraphs(pmcid, partial_sentences)` combined these functions to retrieve and refine XML content. It isolated relevant paragraphs and segmented them into sentences.

To address class imbalance, the `balance_ner_samples(df)` function ensured a more equitable distribution of samples across NER classes, enhancing dataset representativeness.

Lastly, the `process_articles(pmcids, annotations_df)` function orchestrated the data preprocessing pipeline. It operated in batches, fetching and processing content at the sentence level, and efficiently extracting necessary annotations.

In summary, the functions in the `utils.py` module improved the efficiency of data preprocessing. They resolved the challenge of obtaining complete sentence or paragraph context for the dataset, simplifying code organization and enhancing the workflow. `data_preprocessing.py` served as the core script utilizing these functions.

I do not like to have big pipelines in a Jupyter notebook so I will just have this to point to the right python scripts to run:

```
[ ]: ! python3 src/ebi02201/data_preprocessing.py
```

5 3. ML Learning Model Development

There is liberty in using any model you would like e.g., Classification or Question and Answering. However, the dataset here provides an NER task for ML training.

In the `train.py` script, I'm building and training a machine learning model for Named Entity Recognition (NER). This script does several things:

1. **Data Preprocessing**: It starts by loading a dataset from a CSV file using Pandas. This dataset contains sentences and their corresponding NER annotations. I made sure there are no missing annotations or spans for https://europepmc.org/article/MED/22536497 with the code `check_missing_annotations.py` in the `src/ebi02201` folder.

2. **Data Splitting**: The script divides the dataset into three parts: training (70%), validation (15%), and test (15%) to prevent overfitting. Ideally, it should have been done at the article level to avoid testing sentences from the same articles. Due to time constraints (limited to six hours), we couldn't perform a more extensive validation with diverse language styles or species in the test set. The code provided handles the splitting correctly and designates the five PMC IDs with the least sentences as 'test'.

3. **Hyperparameter Tuning**: It performs hyperparameter tuning by exploring different combinations of hyperparameters using a predefined grid search. These hyperparameters include the choice of a pre-trained language model, learning rate, dropout rate, loss type, batch size, and maximum sequence length. It creates a list of hyperparameter combinations and iterates over them.

4. **Model Initialization**: For each combination of hyperparameters, it initializes an instance of the `EntityModel` class defined in the `myModel.py` file. This class constructs the neural network architecture for NER using the Transformers library. It allows customization of model architecture, dropout rates, and loss functions. See the grid search file.

5. **Tokenization and DataLoader Setup**: The script tokenizes the input texts and generates labels for entity recognition. It uses the `AutoTokenizer` from the Transformers library to tokenize the text. For each dataset split (train, val, test), it sets up a DataLoader to efficiently load and iterate through the data during training.

6. **Model Training**: The model is trained using the training dataset. It uses the AdamW optimizer and trains for a specified number of epochs. During training, it calculates and logs the training loss and performs intermediate validation to monitor the model's performance.

7. **Model Evaluation**: Following the training phase, the evaluate function measures the model on both the validation and test datasets. This evaluation encompasses a set of metrics, including true positives (TP), false positives (FP), false negatives (FN), true negatives (TN), precision, recall, F1 score, and Matthews correlation coefficient (MCC) for each individual entity class. These metrics assess the model's performance.

8. **Model Saving**: If the model performs well, it saves the trained model to a file with a descriptive name that includes hyperparameters, timestamp, and all relevant hyperparameters. I should have done it with TensorBoard or MLFlow or DVC Studio to report the models and do some MLOps to organise them.

9. **Iterative Hyperparameter Tuning**: This process repeats for each combination of hyperparameters defined in the grid search, allowing for a comprehensive exploration of model

configurations.

Overall, the `train.py` script is a vital component of the NER model development pipeline. It automates the process of training and evaluating multiple models with different hyperparameters, making it easier to find the best-performing model for the NER task.

Regarding the `myModel.py` file, it contains custom classes used in the NER model development:

1. **EntityModel**: This class defines the neural network architecture for NER. It takes input parameters such as the pre-trained language model, number of labels, dropout rate, and loss function type. It sets up the model, including the label classifier layer, and handles the forward pass to generate class probabilities. It also supports various loss functions, including binary cross-entropy, focal loss, and smoothed focal loss.
2. **FocalLoss**: A custom loss class derived from PyTorch's `nn.Module`. It implements the focal loss function, which is a modification of the binary cross-entropy loss. Focal loss is designed to handle class imbalance and focus on hard-to-classify examples by introducing hyperparameters like alpha and gamma. I wrote this because I thought the model was performing badly but there are some entities missanotated or missing!
3. **SmoothFocalLoss**: Another custom loss class that extends `FocalLoss` by adding label smoothing. Label smoothing is a regularization technique that improves model generalization.

These classes in `myModel.py` provide the foundation for building, training, and evaluating the NER model in the `train.py` script. They encapsulate the model's architecture and loss functions, offering flexibility and customization options for different NER tasks and datasets.

[ ]:
```
!mlflow server \
    --backend-store-uri sqlite:///mlflow.db \
    --default-artifact-root ./mlruns \
    --host 0.0.0.0 \
    --port 5002

!python3 src/ebi02201/train.py
```

6 Exploring Fine-Tuned NER Model Predictions

This code explores predictions made by a pre-trained Named Entity Recognition (NER) model, referred to as `EntityModel`, fine-tuned for identifying entities like ['Gene Mutations', 'Cell', 'Cell Line', 'Organ Tissue'] within text data.

Steps:

1. **Data Preparation:**
    - Load a dataset containing text and corresponding NER labels.
    - Convert NER labels from text to Python objects.
2. **Model Initialization:**
    - Load and initialize the pre-trained NER model for inference. The model specializes in recognizing specified entities within text.
3. **Tokenization and Label Extraction:**
    - Tokenize sentences using the model's tokenizer.

    - Extract actual NER labels from the dataset.
4. **Visualizing Model Predictions:**
    - Primary goal: Visualize model predictions.
    - Visualizations include:
        - **Predicted Probabilities:** Confidence scores for token-level named entity classification.
        - **Ground Truth Labels:** Actual NER labels.
    - These visuals assess the model's accuracy in identifying named entities.
5. **Exploring Sample Data:**
    - Visualizations are generated for a selected dataset subset.
    - The aim is to evaluate model performance and identify areas for potential improvement.

Overall, this code serves to qualitatively evaluate the fine-tuned NER model's performance by visually comparing its predictions with ground truth labels. It aids in assessing accuracy and provides insights for refining named entity recognition tasks.

[ ]:
```
import pandas as pd
import torch
from transformers import AutoTokenizer
from torch.utils.data import DataLoader
from myModel import EntityModel
from train import get_tokens_and_labels, CustomDataset
from plotting import visualize_probabilities_and_labels

# Define the dataset splits
annotation_types = ['Gene Mutations', 'Cell', 'Cell Line', 'Organ Tissue']
label_to_id = {v: k for k, v in enumerate(annotation_types)}

# Load the dataset
file_path = '../../data/raw/biggest_test.csv'
final_df = pd.read_csv(file_path, encoding='utf-8')
final_df['ner'] = final_df['ner'].apply(eval)

# Path to the pre-trained model
model_path = "../../models/
 ↪entity_model_distilbert-base-uncased_20240204_231544_lr5e-05_dropout0.
 ↪1_batch32_seed123.pth"

# Initialize the model
model_name = "distilbert-base-uncased"
model = EntityModel(model_name, num_labels=4, dropout_rate=0)
model.load_state_dict(torch.load(model_path))
model.eval()
print(model)

# Load the tokenizer
max_seq_length = 128  # Specify your desired sequence length
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
# Tokenize sentences and extract labels
tokens_and_labels = get_tokens_and_labels(final_df['sentence'].tolist(),␣
 ↪final_df['ner'].values, tokenizer, max_seq_length, annotation_types,␣
 ↪label_to_id)

# Create a custom dataset and data loader
custom_dataset = CustomDataset(tokens_and_labels)
data_loader = DataLoader(custom_dataset, batch_size=32)

# Extract probabilities for a sample batch
with torch.no_grad():
    for batch in data_loader:
        probabilities, _ = model(batch)
        break

# Sample data for visualization
num_tokens_to_plot = 64  # You can adjust this value
for idx in range(8):
    visualize_probabilities_and_labels(probabilities, batch['labels'], idx,␣
 ↪annotation_types, tokenizer, max_seq_length, final_df)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.bias',
'vocab_transform.bias', 'vocab_layer_norm.bias', 'vocab_layer_norm.weight',
'vocab_transform.weight', 'vocab_projector.weight']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
EntityModel(
  (model): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
  )
  (model_dropout): Dropout(p=0, inplace=False)
  (label_classifier): Linear(in_features=768, out_features=4, bias=True)
  (loss_fct): BCEWithLogitsLoss()
)
```

```
Truncation was not explicitly activated but `max_length` is provided a specific
value, please use `truncation=True` to explicitly truncate examples to max
length. Defaulting to 'longest_first' truncation strategy. If you encode pairs
of sequences (GLUE-style) with the tokenizer you can select this strategy more
precisely by providing a specific strategy to `truncation`.
```

These plots depict token-wise probabilities assigned by my `EntityModel` to different classes for 8 sentences.

In one sentence, the model confidently identifies “oligodendrocytes” as “Cell” with high probability, matching the ground truth label. There is a minor spike in “Organ Tissue” and “Cell Line” probabilities for the token “neurons,” which is not labeled as such and maybe it should as it could be labeled as a tissue or cells. The same happends for “microglia” which is not bad!

In anotther sentence, “astrocytes” is correctly identified as “Cell” with high probability, aligning with the ground truth. The model does not misclassify many tokens with high confidence as any other class. It has a high precision as you will see below in the F1 scores and the MCC scores.

7 4. Evaluation

[ ]:
```
import pandas as pd
import plotly.express as px
from gridsearch import GRID_SEARCH

# Load the dataset
df = pd.read_csv('../runs/hparam_tuning_mlflow.csv')
print(df.columns)
```

```
# Define the hyperparameter column
hparam_column = 'DROPOUT_RATE'

# Define the F1 score columns for different classes
f1_columns = ['test_Class_2_F1', 'test_Class_3_F1', 'test_Class_4_F1']

# Define class names
class_names = ['Gene Mutations', 'Cell', 'Cell Line', 'Organ Tissue']

# Iterate through F1 score columns and create box plots
for i, f1_column in enumerate(f1_columns):
    class_name = class_names[i+1]

    # Create a box plot for F1 score based on max_seq_length, learning_rate,␣
 ↪and loss_type
    fig = px.box(df, x='MAX_SEQ_LENGTH', y=f1_column, color='LOSS_TYPE',
                 facet_col='LEARNING_RATE', points='all',
                 labels={'MAX_SEQ_LENGTH': 'Max Seq Length', f1_column: 'F1␣
 ↪Score'},
                 title=f"Box Plots of F1 Score for {class_name} Class")

    # Show the plot
    fig.show()

# Show the grid search
print(GRID_SEARCH)
```

```
Index(['Start Time', 'Duration', 'Run ID', 'Name', 'Source Type',
       'Source Name', 'User', 'Status', 'BATCH_SIZE', 'DROPOUT_RATE',
       'LEARNING_RATE', 'LOSS_TYPE', 'MAX_SEQ_LENGTH', 'MODEL_NAME',
       'test_Class_1_F1', 'test_Class_1_FN', 'test_Class_1_FP',
       'test_Class_1_MCC', 'test_Class_1_Precision', 'test_Class_1_Recall',
       'test_Class_1_TN', 'test_Class_1_TP', 'test_Class_2_F1',
       'test_Class_2_FN', 'test_Class_2_FP', 'test_Class_2_MCC',
       'test_Class_2_Precision', 'test_Class_2_Recall', 'test_Class_2_TN',
       'test_Class_2_TP', 'test_Class_3_F1', 'test_Class_3_FN',
       'test_Class_3_FP', 'test_Class_3_MCC', 'test_Class_3_Precision',
       'test_Class_3_Recall', 'test_Class_3_TN', 'test_Class_3_TP',
       'test_Class_4_F1', 'test_Class_4_FN', 'test_Class_4_FP',
       'test_Class_4_MCC', 'test_Class_4_Precision', 'test_Class_4_Recall',
       'test_Class_4_TN', 'test_Class_4_TP', 'val_Class_1_F1',
       'val_Class_1_FN', 'val_Class_1_FP', 'val_Class_1_MCC',
       'val_Class_1_Precision', 'val_Class_1_Recall', 'val_Class_1_TN',
       'val_Class_1_TP', 'val_Class_2_F1', 'val_Class_2_FN', 'val_Class_2_FP',
       'val_Class_2_MCC', 'val_Class_2_Precision', 'val_Class_2_Recall',
       'val_Class_2_TN', 'val_Class_2_TP', 'val_Class_3_F1', 'val_Class_3_FN',
       'val_Class_3_FP', 'val_Class_3_MCC', 'val_Class_3_Precision',
       'val_Class_3_Recall', 'val_Class_3_TN', 'val_Class_3_TP',
       'val_Class_4_F1', 'val_Class_4_FN', 'val_Class_4_FP', 'val_Class_4_MCC',
       'val_Class_4_Precision', 'val_Class_4_Recall', 'val_Class_4_TN',
       'val_Class_4_TP'],
      dtype='object')
```

```
{'MODEL_NAME': ['distilbert-base-uncased', 'dmis-lab/TinyPubMedBERT-v1.0',
'dmis-lab/biobert-base-cased-v1.2'], 'LEARNING_RATE': [2e-05, 5e-05],
'DROPOUT_RATE': [0, 0.1], 'LOSS_TYPE': ['bce', 'focal', 'smooth_focal'],
'BATCH_SIZE': [16, 32], 'MAX_SEQ_LENGTH': [128, 256]}
```

Based on the box plots provided in above images…

7.0.1 Cell Line Class

- For a maximum sequence length of 64, a learning rate of 5e-05 appears to perform better across all loss types, with 'focal' loss type having a slightly higher median F1 score.
- Increasing the maximum sequence length to 128 shows a general decrease in performance for the learning rate of 5e-05, with 'bce' loss type now showing the highest median F1 score.
- With a learning rate of 2e-05, the maximum sequence length of 64 shows comparable performance across all loss types, but 'smooth_focal' has a slightly higher median F1 score.
- When the maximum sequence length is increased to 128 for the learning rate of 2e-05, 'focal' loss type leads to the highest median F1 score.

7.0.2 Organ Tissue Class

- At a learning rate of 5e-05, 'focal' loss type has the highest median F1 score for both sequence lengths, with 64 being slightly better than 128.
- For a learning rate of 2e-05, 'bce' and 'smooth_focal' have similar median F1 scores for a sequence length of 64, but 'bce' has a higher median for the sequence length of 128.

7.0.3 Best Overall Hyperparameter Combination

- For the "Cell Line" class, the combination of a 5e-05 learning rate, 64 maximum sequence length, and 'focal' loss type is optimal.
- For the "Organ Tissue" class, the combination of a 5e-05 learning rate, 64 maximum sequence length, and 'focal' loss type also yields the best results.

It should be noted that the spread and outliers may affect the robustness of these results. The choice of the "best" hyperparameter combination may depend on whether the priority is higher median F1 score or consistency (tighter interquartile range).

8 5. Complexity

In my work, I'm dealing with an entity recognition model known for its power but also its large size and high computational requirements. This can be a problem when deploying the model in resource-constrained production environments. To tackle this complexity, I'm exploring model compression techniques, with a focus on pruning.

Model compression means making a model smaller without sacrificing its performance. It involves various methods tailored to specific needs. One common approach is pruning, which involves removing less important model weights or neurons. This can be done by either removing individual weights or entire neurons or layers.

In my case, I have a model called `loaded_model`, which is designed for entity recognition with `num_labels=4`. This model consists of several components, including embedding layers, a multi-layer BERT encoder, a pooling layer, and a linear classifier.

8.1 5.1 Pruning

To simplify the model through pruning, I'll use PyTorch's `torch.nn.utils.prune` library. This versatile tool allows me to identify and remove less critical weights or neurons, effectively compressing the model. It's crucial to fine-tune the pruning to balance model size reduction with preserving performance.

Additionally, apart from pruning, I may explore other compression techniques like quantization or knowledge distillation, depending on the specific resource constraints of my deployment environment. These methods offer more ways to optimize the model's efficiency while maintaining its effectiveness.

[ ]:
```
import torch
import torch.nn as nn
import torch.nn.utils.prune as prune
from myModel import EntityModel

# Path to the pre-trained model
model_path = "../../models/
 ↪entity_model_distilbert-base-uncased_20240204_221018_lr2e-05_dropout0_batch16_seed123.
 ↪pth"

path_parts = model_path.split("_")
model_name = path_parts[1]
model_name = 'distilbert-base-uncased'

# Initialize the model
loaded_model = EntityModel(model_name=model_name, num_labels=4, dropout_rate=0)
loaded_model.load_state_dict(torch.load(model_path))
loaded_model.eval()
print(loaded_model)

# Define pruning parameters
pruning_method = 'l1_unstructured'   # There are other methods
pruning_rate = 0.5  # Adjust the pruning rate as needed

# Prune the loaded model
for name, module in loaded_model.named_modules():
    if isinstance(module, nn.Linear):
        prune.ln_structured(module, name="weight", amount=pruning_rate, n=1,␣
↪dim=0)

# Save the pruned model's state dictionary
pruned_model_state_dict_path = model_path.replace('.pth', '-pruned.pth')
torch.save(loaded_model.state_dict(), pruned_model_state_dict_path)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias', 'vocab_transform.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
EntityModel(
  (model): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): Linear(in_features=768, out_features=768, bias=True)
            (k_lin): Linear(in_features=768, out_features=768, bias=True)
            (v_lin): Linear(in_features=768, out_features=768, bias=True)
            (out_lin): Linear(in_features=768, out_features=768, bias=True)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): Linear(in_features=768, out_features=3072, bias=True)
            (lin2): Linear(in_features=3072, out_features=768, bias=True)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
  )
  (model_dropout): Dropout(p=0, inplace=False)
  (label_classifier): Linear(in_features=768, out_features=4, bias=True)
  (loss_fct): BCEWithLogitsLoss()
)
```

8.2 5.2 Quantization

Quantization is a technique used to decrease the level of precision of a model's parameters. For example, it can convert parameters from being represented as 32-bit floating-point numbers (float32) to 8-bit integers (int8). The primary purpose of quantization is to reduce the model's size and make it more memory-efficient, which can lead to faster inference times, particularly on hardware that is optimized for low-precision arithmetic operations.

```
[ ]: import torch
     import torch.nn as nn
     from myModel import EntityModel
     import copy
     # Set the quantization engine
     torch.backends.quantized.engine = 'qnnpack'  # Use for ARM M1 Mac

     def quantize_model(model):
         quantized_model = copy.deepcopy(model)
         for name, module in quantized_model.named_modules():
             if isinstance(module, nn.Linear):
                 # Quantize the weights
                 module.weight.data = torch.quantize_per_tensor(module.weight.data,␣
       ↪scale=0.1, zero_point=0, dtype=torch.qint8)
                 if module.bias is not None:
                     # Quantize the biases
                     module.bias.data = torch.quantize_per_tensor(module.bias.data,␣
       ↪scale=0.1, zero_point=0, dtype=torch.qint32)
         return quantized_model

     # Path to the pre-trained model
     model_path = "../../models/
       ↪entity_model_distilbert-base-uncased_20240204_221018_lr2e-05_dropout0_batch16_seed123.
       ↪pth"
     path_parts = model_path.split("_")
     model_name = path_parts[2]
     print(model_name)

     # Initialize the model
     loaded_model = EntityModel(model_name=model_name, num_labels=4, dropout_rate=0)
     loaded_model.load_state_dict(torch.load(model_path))
     loaded_model.eval()
```

```python
# Clone the model for quantization
# quantized_model = quantize_model(loaded_model)

# Apply dynamic quantization
quantized_model = torch.quantization.quantize_dynamic(
    loaded_model,
    {nn.Linear},  # Specify which types of layers to dynamically quantize
    dtype=torch.qint8
)

# Save the quantized model's state dictionary with a specific name reflecting␣
 ↪its quantized state
quantized_model_state_dict_path = model_path.replace('.pth', '-quantized.pth')
torch.save(quantized_model.state_dict(), quantized_model_state_dict_path)

print('QUANTISED MODEL with torch.qint8', quantized_model)
```

```
distilbert-base-uncased
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias', 'vocab_transform.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
QUANTISED MODEL EntityModel(
  (model): DistilBertModel(
    (embeddings): Embeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (transformer): Transformer(
      (layer): ModuleList(
        (0-5): 6 x TransformerBlock(
          (attention): MultiHeadSelfAttention(
            (dropout): Dropout(p=0.1, inplace=False)
            (q_lin): DynamicQuantizedLinear(in_features=768, out_features=768,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
            (k_lin): DynamicQuantizedLinear(in_features=768, out_features=768,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
```

```
            (v_lin): DynamicQuantizedLinear(in_features=768, out_features=768,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
            (out_lin): DynamicQuantizedLinear(in_features=768, out_features=768,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
          )
          (sa_layer_norm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (ffn): FFN(
            (dropout): Dropout(p=0.1, inplace=False)
            (lin1): DynamicQuantizedLinear(in_features=768, out_features=3072,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
            (lin2): DynamicQuantizedLinear(in_features=3072, out_features=768,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
            (activation): GELUActivation()
          )
          (output_layer_norm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
        )
      )
    )
  )
  (model_dropout): Dropout(p=0, inplace=False)
  (label_classifier): DynamicQuantizedLinear(in_features=768, out_features=4,
dtype=torch.qint8, qscheme=torch.per_tensor_affine)
  (loss_fct): BCEWithLogitsLoss()
)
```

8.3 5.3 Knowledge Distillation

Knowledge distillation is a technique used in machine learning where a smaller and more computationally efficient model, often referred to as the "student," is trained to replicate the behavior and predictions of a larger and pre-trained model known as the "teacher." The primary goal of knowledge distillation is to transfer the knowledge and expertise encapsulated in the teacher model to the student model.

In this process, the student model learns to approximate not only the final predictions of the teacher model (often referred to as "hard labels") but also the more nuanced output distributions generated by the teacher model (referred to as "soft targets"). These soft targets contain more detailed and fine-grained information per training example, providing a richer source of guidance for the student model.

By mimicking the teacher model's behavior and learning from its soft targets, the student model can achieve similar performance levels as the teacher model while being more lightweight and suitable for deployment in resource-constrained environments. Knowledge distillation is especially valuable when the teacher model is computationally expensive, and there is a need for a smaller model that can approximate its capabilities efficiently.

[ ]:
```python
import torch
from torch.nn import KLDivLoss, CrossEntropyLoss
import torch.optim as optim
```

```python
from transformers import AutoTokenizer
from torch.utils.data import DataLoader
from train import get_tokens_and_labels, CustomDataset
import pandas as pd
from train import get_tokens_and_labels
from myModel import EntityModel

# Define annotation classes
annotation_types = ['Gene Mutations', 'Cell', 'Cell Line', 'Organ Tissue']
label_to_id = {v: k for k, v in enumerate(annotation_types)}

# Initialize Teacher Model
teacher_model_path = "../../models/
 ↪entity_model_distilbert-base-uncased_20240204_221018_lr2e-05_dropout0_batch16_seed123.
 ↪pth"
teacher_model = EntityModel(model_name="distilbert-base-uncased", num_labels=4,␣
 ↪dropout_rate=0)
teacher_model.load_state_dict(torch.load(teacher_model_path))
teacher_model.eval()

# Initialize Student Model
student_model_name = "prajjwal1/bert-mini"
student_model = EntityModel(model_name=student_model_name, num_labels=4,␣
 ↪dropout_rate=0)

# Load and preprocess dataset
file_path = '../../data/raw/biggest_test.csv'
final_df = pd.read_csv(file_path, encoding='utf-8')
final_df['ner'] = final_df['ner'].apply(eval)

# Load the tokenizer
max_seq_length = 128  # Specify your desired sequence length
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Tokenize sentences and extract labels
tokens_and_labels = get_tokens_and_labels(final_df['sentence'].tolist(),␣
 ↪final_df['ner'].values, tokenizer, max_seq_length, annotation_types,␣
 ↪label_to_id)

# Create a custom dataset and data loader
custom_dataset = CustomDataset(tokens_and_labels)
data_loader = DataLoader(custom_dataset, batch_size=32)

# Knowledge Distillation Training
distillation_loss_fn = KLDivLoss(reduction='batchmean')
classification_loss_fn = CrossEntropyLoss()
optimizer = optim.Adam(student_model.parameters(), lr=5e-5)
```

```
# Distillation Training Loop
num_epochs = 5
for epoch in range(num_epochs):
    for batch in data_loader:
        # Forward pass of teacher with input
        with torch.no_grad():
            teacher_outputs, _ = teacher_model(batch)

        # Forward pass of student
        student_outputs, student_loss = student_model(batch)

        # Calculate distillation loss
        distillation_loss = distillation_loss_fn(student_outputs,␣
 ↪teacher_outputs.detach())

        # Total loss
        loss = distillation_loss + student_loss

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs} completed.")

# Save the distilled student model
student_model_path = teacher_model_path.replace('.pth', '-student.pth')
torch.save(student_model.state_dict(), student_model_path)
```

```
Some weights of the model checkpoint at distilbert-base-uncased were not used
when initializing DistilBertModel: ['vocab_projector.weight',
'vocab_projector.bias', 'vocab_layer_norm.bias', 'vocab_transform.weight',
'vocab_transform.bias', 'vocab_layer_norm.weight']
- This IS expected if you are initializing DistilBertModel from the checkpoint
of a model trained on another task or with another architecture (e.g.
initializing a BertForSequenceClassification model from a BertForPreTraining
model).
- This IS NOT expected if you are initializing DistilBertModel from the
checkpoint of a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
Some weights of the model checkpoint at prajjwal1/bert-mini were not used when
initializing BertModel: ['cls.seq_relationship.bias',
'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.bias',
'cls.predictions.decoder.weight', 'cls.seq_relationship.weight',
'cls.predictions.bias', 'cls.predictions.transform.dense.bias',
'cls.predictions.transform.LayerNorm.weight',
'cls.predictions.transform.LayerNorm.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a
model trained on another task or with another architecture (e.g. initializing a
BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of
a model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[20], line 55
     52     teacher_outputs, _ = teacher_model(batch)
     54 # Forward pass of student
---> 55 student_outputs, student_loss = student_model(batch)
     57 # Calculate distillation loss
     58 distillation_loss = distillation_loss_fn(student_outputs,␣
 ↪teacher_outputs.detach())

File ~/miniforge3/envs/gpt/lib/python3.9/site-packages/torch/nn/modules/module.
 ↪py:1518, in Module._wrapped_call_impl(self, *args, **kwargs)
   1516     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

File ~/miniforge3/envs/gpt/lib/python3.9/site-packages/torch/nn/modules/module.
 ↪py:1527, in Module._call_impl(self, *args, **kwargs)
   1522 # If we don't have any hooks, we want to skip the rest of the logic in
   1523 # this function, and just call forward.
   1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1525         or _global_backward_pre_hooks or _global_backward_hooks
   1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
   1529 try:
   1530     result = None

File ~/repos/exscientia/ebi02201/src/ebi02201/myModel.py:72, in EntityModel.
 ↪forward(self, batch)
     70 x = self.model(input_ids=batch['input_ids'],␣
 ↪attention_mask=batch['attention_mask'])[0]
     71 x = self.model_dropout(x)
---> 72 logits = self.label_classifier(x)
     74 probabilities = torch.sigmoid(logits)
     76 if 'labels' in batch:
```

```
File ~/miniforge3/envs/gpt/lib/python3.9/site-packages/torch/nn/modules/module.
 ↪py:1518, in Module._wrapped_call_impl(self, *args, **kwargs)
   1516     return self._compiled_call_impl(*args, **kwargs)  # type:␣
 ↪ignore[misc]
   1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

File ~/miniforge3/envs/gpt/lib/python3.9/site-packages/torch/nn/modules/module.
 ↪py:1527, in Module._call_impl(self, *args, **kwargs)
   1522 # If we don't have any hooks, we want to skip the rest of the logic in
   1523 # this function, and just call forward.
   1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
 ↪_forward_hooks or self._forward_pre_hooks
   1525         or _global_backward_pre_hooks or _global_backward_hooks
   1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
   1529 try:
   1530     result = None

File ~/miniforge3/envs/gpt/lib/python3.9/site-packages/torch/nn/modules/linear.
 ↪py:114, in Linear.forward(self, input)
    113 def forward(self, input: Tensor) -> Tensor:
--> 114     return F.linear(input, self.weight, self.bias)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (4096x256 and 768x4)
```

```
[ ]:
```