Long Nguyen
Ming-Shih Wang
Defne Coban

Project 2

Introduction

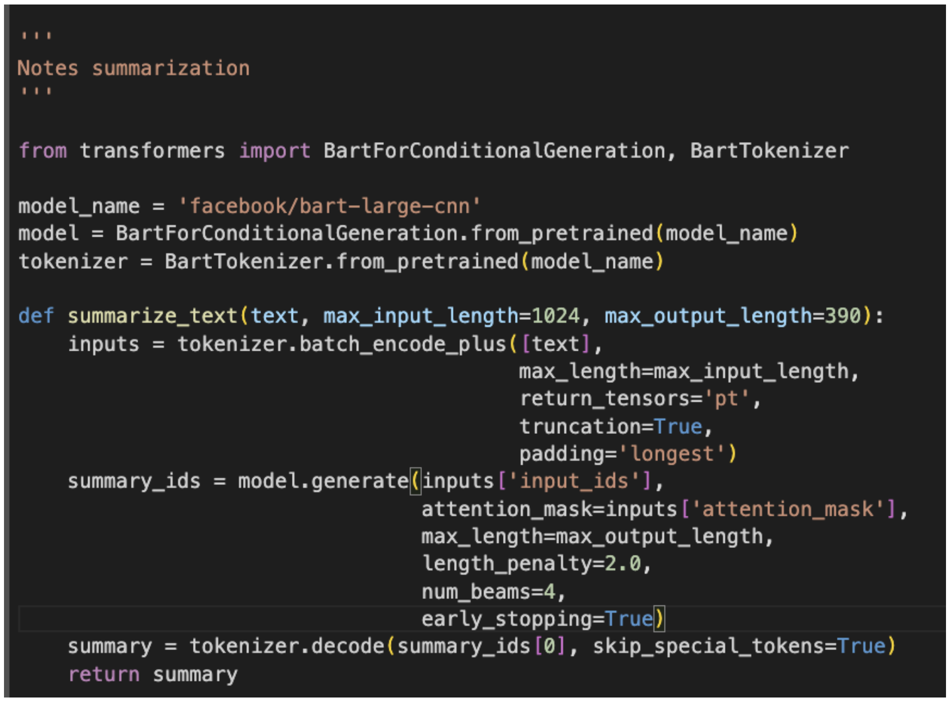
This project aims to fine-tune an LLM to generate accurate ARDS (acute respiratory distress syndrome) detection based on clinical notes. Although we spent the majority of our time summarizing and using data manipulation for data imbalances, we ran out of time and resorted to a simpler, less accurate method.

Contributions

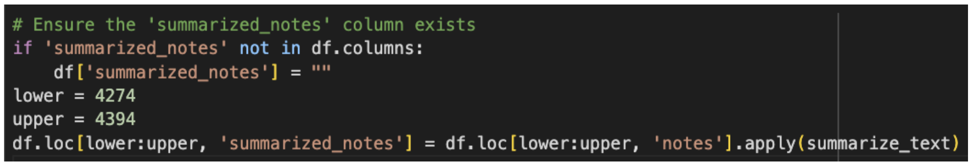
Long Nguyen: Summarizations, Data Manipulation, Fine-Tuning, Training, Report
Ming-Shih Wang: Summarizations, Fine-Tuning, Training, Report
Defne Coban: Summarizations, Code-Explanation, Training, Output Generation, Report

Intended Process

1. **Summarization**
   Initially, we summarized and cleaned the doctor's notes with BART to under 400 tokens. Each note would take 30 seconds to summarize, taking us 66 hours of continuous summarizing. Given the time constraint on Google Collab, we decided to train the data on 1-hour segments, meaning 120 notes for each partition. We did this for all 8100 samples, which was time-consuming.



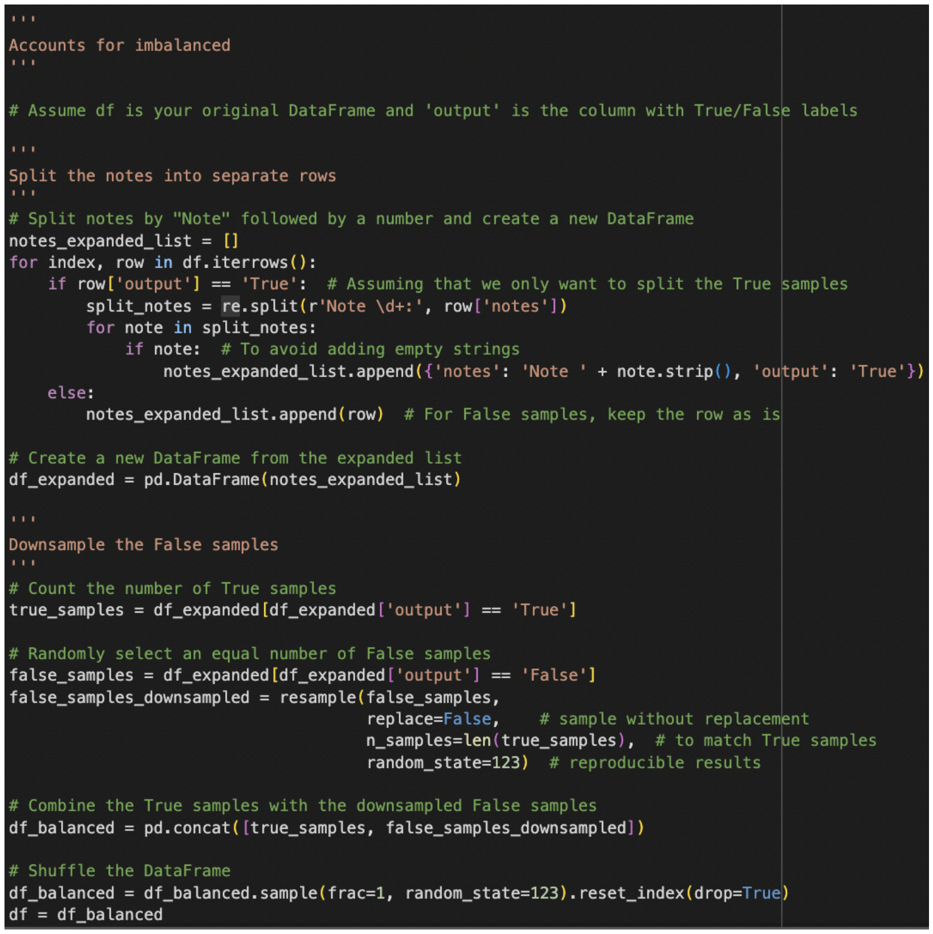
```
'''
Notes summarization
'''

from transformers import BartForConditionalGeneration, BartTokenizer

model_name = 'facebook/bart-large-cnn'
model = BartForConditionalGeneration.from_pretrained(model_name)
tokenizer = BartTokenizer.from_pretrained(model_name)

def summarize_text(text, max_input_length=1024, max_output_length=390):
    inputs = tokenizer.batch_encode_plus([text],
                                         max_length=max_input_length,
                                         return_tensors='pt',
                                         truncation=True,
                                         padding='longest')
    summary_ids = model.generate(inputs['input_ids'],
                                 attention_mask=inputs['attention_mask'],
                                 max_length=max_output_length,
                                 length_penalty=2.0,
                                 num_beams=4,
                                 early_stopping=True)
    summary = tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary
```



```
# Ensure the 'summarized_notes' column exists
if 'summarized_notes' not in df.columns:
    df['summarized_notes'] = ""
lower = 4274
upper = 4394
df.loc[lower:upper, 'summarized_notes'] = df.loc[lower:upper, 'notes'].apply(summarize_text)
```

2. Resolve Data Imbalance

After generating the summarizations from the training set, we realized that the data is heavily imbalanced with 180 true values and 8000 false values. Our solution was to still utilize the summarized notes correlated to the False values but split the original notes that correlated to True values into subsections for a more balanced dataset. Within each medical note, there are many subnets. To populate the True samples, we split the one “clumped up” medical note of the True into however many sub notes were in there. For instance, if it had 25 sub notes, the one True sample would become 25 samples. After this process, we had 3505 of each True and False sample



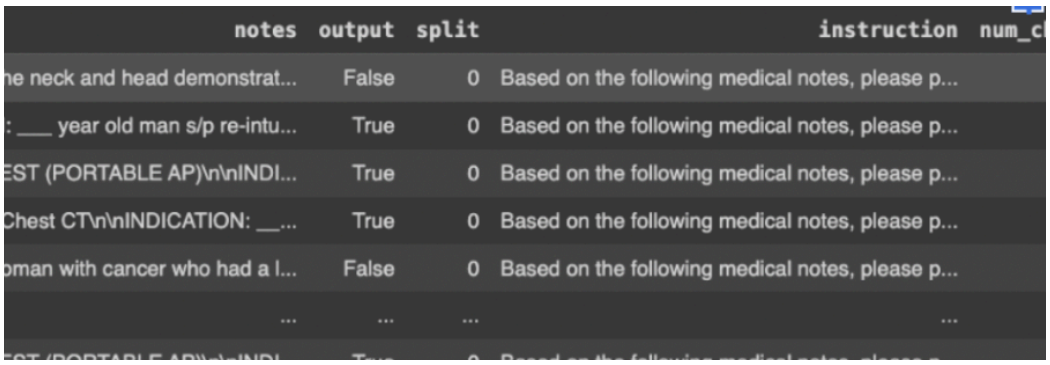
```
'''
Accounts for imbalanced
'''

# Assume df is your original DataFrame and 'output' is the column with True/False labels

'''
Split the notes into separate rows
'''
# Split notes by "Note" followed by a number and create a new DataFrame
notes_expanded_list = []
for index, row in df.iterrows():
    if row['output'] == 'True':  # Assuming that we only want to split the True samples
        split_notes = re.split(r'Note \d+:', row['notes'])
        for note in split_notes:
            if note:  # To avoid adding empty strings
                notes_expanded_list.append({'notes': 'Note ' + note.strip(), 'output': 'True'})
    else:
        notes_expanded_list.append(row)  # For False samples, keep the row as is

# Create a new DataFrame from the expanded list
df_expanded = pd.DataFrame(notes_expanded_list)

'''
Downsample the False samples
'''
# Count the number of True samples
true_samples = df_expanded[df_expanded['output'] == 'True']

# Randomly select an equal number of False samples
false_samples = df_expanded[df_expanded['output'] == 'False']
false_samples_downsampled = resample(false_samples,
                                     replace=False,    # sample without replacement
                                     n_samples=len(true_samples),  # to match True samples
                                     random_state=123)  # reproducible results

# Combine the True samples with the downsampled False samples
df_balanced = pd.concat([true_samples, false_samples_downsampled])

# Shuffle the DataFrame
df_balanced = df_balanced.sample(frac=1, random_state=123).reset_index(drop=True)
df = df_balanced
```

Long Nguyen
Ming-Shih Wang
Defne Coban

Then we randomized the notes to ensure the following qualities.

Preventing Model Bias:
If all true examples are followed by all false examples (or vice versa), the model might learn the sequence in which the data is presented as a feature, leading to biased learning. Randomizing ensures that the model does not pick up on the order of the data as a pattern.

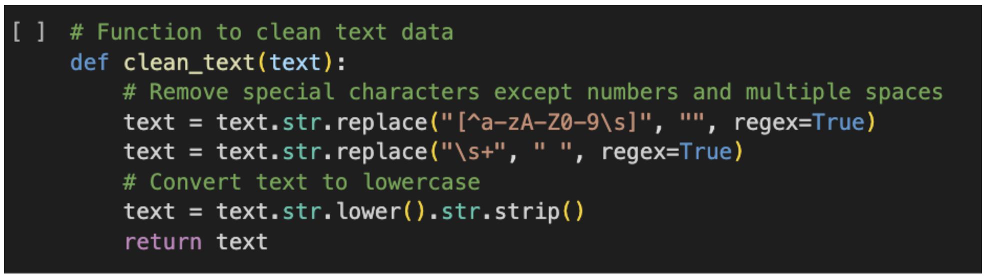
Training and Validation Split:
When splitting the dataset into training and validation sets, having a randomized dataset ensures that both sets are representative of the overall distribution of the data. If the data is not randomized, there's a risk that the training set might contain only one type of label (all true or all falses), which would not train the model effectively for the other label.



notes	output	split	instruction	num_c
ne neck and head demonstrat...	False	0	Based on the following medical notes, please p...	
: ___ year old man s/p re-intu...	True	0	Based on the following medical notes, please p...	
EST (PORTABLE AP)\n\nINDI...	True	0	Based on the following medical notes, please p...	
Chest CT\n\nINDICATION: ___...	True	0	Based on the following medical notes, please p...	
oman with cancer who had a l...	False	0	Based on the following medical notes, please p...	
...	...	...	...	

3. Cleaning the text-data

We implemented a function, clean_text, to remove special characters, numbers, and extraneous spaces from the text. This function also converted the text to lowercase and stripped leading and trailing spaces for uniformity. We applied this cleaning process to both the 'notes' and 'instruction' columns.

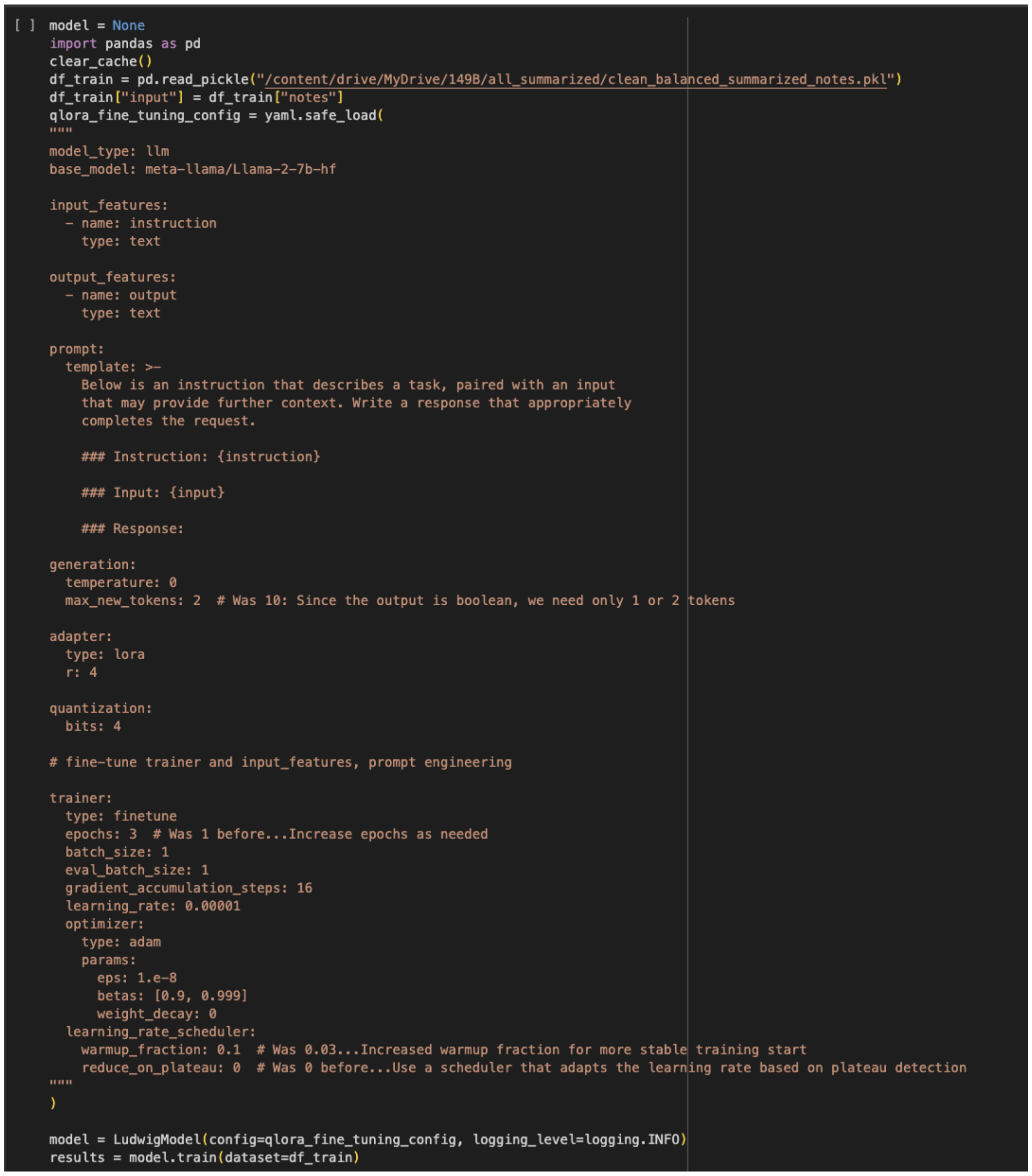


```
[ ]  # Function to clean text data
     def clean_text(text):
         # Remove special characters except numbers and multiple spaces
         text = text.str.replace("[^a-zA-Z0-9\s]", "", regex=True)
         text = text.str.replace("\s+", " ", regex=True)
         # Convert text to lowercase
         text = text.str.lower().str.strip()
         return text
```

Long Nguyen
Ming-Shih Wang
Defne Coban

4. Training the model

In addition, we fine-tuned the model before training in the YAML configuration, such as the epochs, number of new tokens, and the warmup function. Now that we have our training data prepared, our GPU unfortunately has reached its capacity. We attempted to train the model locally with VScode, however, we didn't realize that the CUDA accelerator is not compatible with Applic Silicon. With MPS (Metal Performance Shaders), Apple Silicon's accelerator, we weren't able to utilize Ludwig's quantization technique to reduce the model size, resulting in a massive training time that wasn't possible for our given time frame.



```
[ ] model = None
    import pandas as pd
    clear_cache()
    df_train = pd.read_pickle("/content/drive/MyDrive/149B/all_summarized/clean_balanced_summarized_notes.pkl")
    df_train["input"] = df_train["notes"]
    qlora_fine_tuning_config = yaml.safe_load(
    """
    model_type: llm
    base_model: meta-llama/Llama-2-7b-hf

    input_features:
      - name: instruction
        type: text

    output_features:
      - name: output
        type: text

    prompt:
      template: >-
        Below is an instruction that describes a task, paired with an input
        that may provide further context. Write a response that appropriately
        completes the request.

        ### Instruction: {instruction}

        ### Input: {input}

        ### Response:

    generation:
      temperature: 0
      max_new_tokens: 2  # Was 10: Since the output is boolean, we need only 1 or 2 tokens

    adapter:
      type: lora
      r: 4

    quantization:
      bits: 4

    # fine-tune trainer and input_features, prompt engineering

    trainer:
      type: finetune
      epochs: 3  # Was 1 before...Increase epochs as needed
      batch_size: 1
      eval_batch_size: 1
      gradient_accumulation_steps: 16
      learning_rate: 0.00001
      optimizer:
        type: adam
        params:
          eps: 1.e-8
          betas: [0.9, 0.999]
          weight_decay: 0
      learning_rate_scheduler:
        warmup_fraction: 0.1  # Was 0.03...Increased warmup fraction for more stable training start
        reduce_on_plateau: 0  # Was 0 before...Use a scheduler that adapts the learning rate based on plateau detection
    """
    )

    model = LudwigModel(config=qlora_fine_tuning_config, logging_level=logging.INFO)
    results = model.train(dataset=df_train)
```

Long Nguyen
Ming-Shih Wang
Defne Coban

Submitted Process

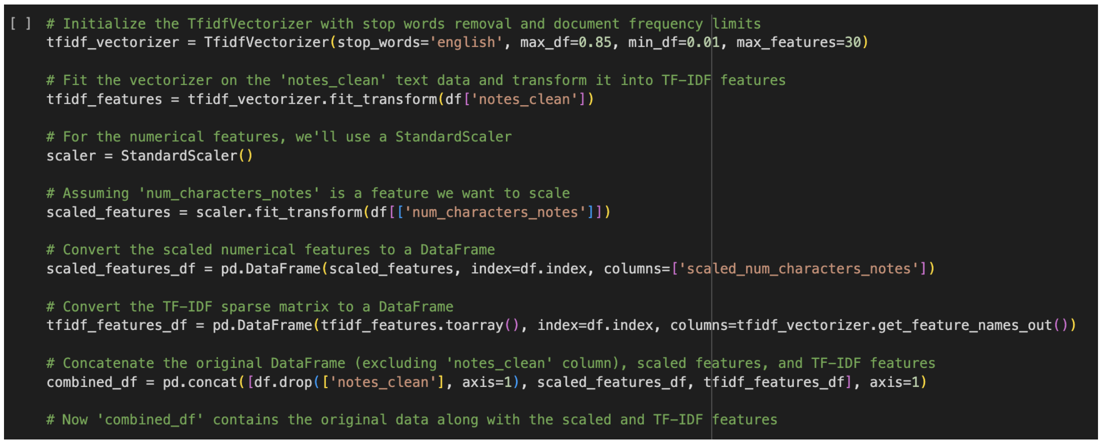
For the training data, we employed TfidfVectorizer, configured to remove English common words and stop words and limit document frequency. This transformed the cleaned notes into TF-I.

1. **Data Cleaning and Truncation:**

   TF-IDF Vectorization: For textual data, we employed TfidfVectorizer, configured to remove English common words and stop words and limit document frequency. This transformed the cleaned notes into TF-IDF features.

   Numerical Feature Scaling: We used StandardScaler to normalize numerical features, such as 'num_characters_notes'. This scaling ensured that our numerical data would not skew the model due to differing scales.

   Data Integration: The processed textual and numerical features were combined into a single DataFrame, alongside the original data, to create a comprehensive dataset for training.



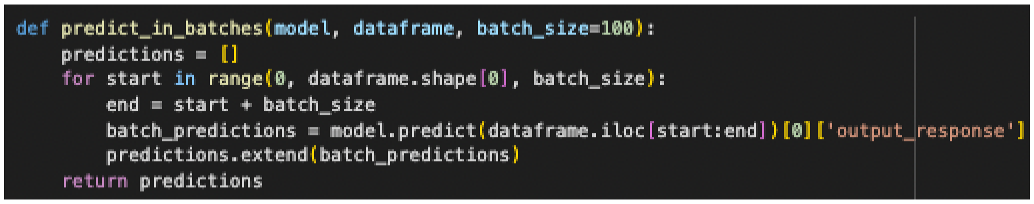
```
[ ]  # Initialize the TfidfVectorizer with stop words removal and document frequency limits
     tfidf_vectorizer = TfidfVectorizer(stop_words='english', max_df=0.85, min_df=0.01, max_features=30)

     # Fit the vectorizer on the 'notes_clean' text data and transform it into TF-IDF features
     tfidf_features = tfidf_vectorizer.fit_transform(df['notes_clean'])

     # For the numerical features, we'll use a StandardScaler
     scaler = StandardScaler()

     # Assuming 'num_characters_notes' is a feature we want to scale
     scaled_features = scaler.fit_transform(df[['num_characters_notes']])

     # Convert the scaled numerical features to a DataFrame
     scaled_features_df = pd.DataFrame(scaled_features, index=df.index, columns=['scaled_num_characters_notes'])

     # Convert the TF-IDF sparse matrix to a DataFrame
     tfidf_features_df = pd.DataFrame(tfidf_features.toarray(), index=df.index, columns=tfidf_vectorizer.get_feature_names_out())

     # Concatenate the original DataFrame (excluding 'notes_clean' column), scaled features, and TF-IDF features
     combined_df = pd.concat([df.drop(['notes_clean'], axis=1), scaled_features_df, tfidf_features_df], axis=1)

     # Now 'combined_df' contains the original data along with the scaled and TF-IDF features
```

2. **Predictions and Batch Processing**

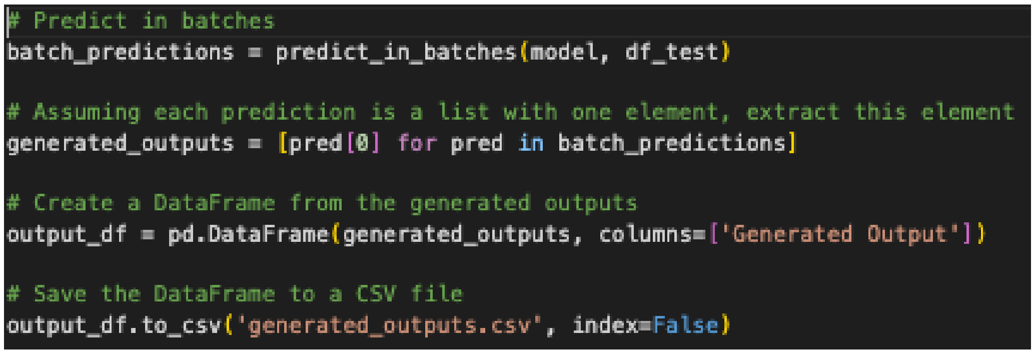
   In generating predictions from the test set, we employed a batch-processing methodology to optimize our computational resources. Recognizing the challenges posed by the large volume of data, we developed a function named predict_in_batches. This approach prevented system overload. Here, we first conducted a preliminary time estimation to figure out how long it would take for our predictions to be complete. The execution of the batch predictions was carried out using our custom function. The output from these predictions was systematically transformed into a structured format conducive to analysis. In the final stage of this process, the predictions were compiled into a data frame. This data was

Long Nguyen
Ming-Shih Wang
Defne Coban

then exported to a CSV file serving as our project's conclusive output. This file encapsulated our findings and was poised for evaluation based on the predefined criteria of the project.



```
def predict_in_batches(model, dataframe, batch_size=100):
    predictions = []
    for start in range(0, dataframe.shape[0], batch_size):
        end = start + batch_size
        batch_predictions = model.predict(dataframe.iloc[start:end])[0]['output_response']
        predictions.extend(batch_predictions)
    return predictions
```



```
# Predict in batches
batch_predictions = predict_in_batches(model, df_test)

# Assuming each prediction is a list with one element, extract this element
generated_outputs = [pred[0] for pred in batch_predictions]

# Create a DataFrame from the generated outputs
output_df = pd.DataFrame(generated_outputs, columns=['Generated Output'])

# Save the DataFrame to a CSV file
output_df.to_csv('generated_outputs.csv', index=False)
```