

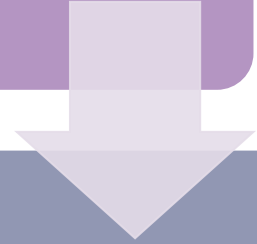
POS tagging project

By: Ahmed Bahaa
49-0354



Agenda

Data Cleaning
and
preprocessing



Milestone 2



Milestone 3

The image features a light gray background with decorative white line art of leaves and branches in the corners. The top-left and top-right corners have clusters of elongated, pointed leaves. The bottom-left and bottom-right corners have clusters of rounded, heart-shaped leaves. A thin horizontal line is centered below the main title.

Data Cleaning

- By inspecting the dataset, I discovered some rows that have the same word repeated many times. This needs to be cleaned by dropping these rows.
- I dropped any row that has the same word repeated more than 4 times.
- No null values were found
- Finally, I removed special characters and non -alphanumeric characters.

```
[ ] # Dropping rows that have the same word repeated more than 4 times
def drop_repeated_words(df, max_repeats=4):

    # Splitting each row's text into words
    words = df['response'].str.split()

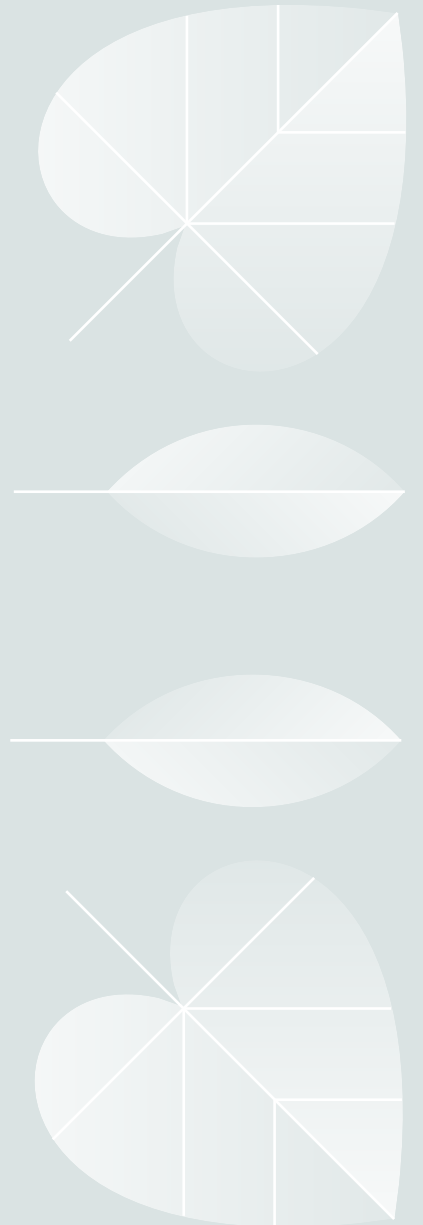
    # Checking for repeated words
    repeated_indices = []
    for i, row_words in enumerate(words):
        word_counts = {}
        for word in row_words:
            word_counts[word] = word_counts.get(word, 0) + 1
            if word_counts[word] > max_repeats:
                repeated_indices.append(i)
                break

    df.drop(repeated_indices, inplace=True)
```

Data preprocessing



-
- Before building the models, data must be preprocessed. I started by tokenization because the model must take the inputs tokenized.
 - After that, I applied lemmatization which returns words to their root form.
 - Next, I removed stop word.
 - Then, I removed punctuation.
 - Finally, I created a method that will return the POS tags for each tokenized word.





Milestone 2

Data Preparation

- I STARTED BY CREATING A VOCABULARY FROM THE WORDS IN MY DATASET. EACH UNIQUE WORD IS ASSIGNED AN INDEX.
- THEN I LOOPED OVER THE TOKENIZED WORDS IN MY DATASET AND CONVERTED TOKENS TO INDICES USING THE TOKENS TO INDICES FUNCTION
- NOW, ALL_INDICES LIST CONTAINS ALL THE WORDS OF THE DATASET REPRESENTED AS THEIR CORRESPONDING INDEX

all_indices list contains the indexes of all tokens in the vocab

```
# Initialize a list to store indices for each row
all_indices = []

# Iterate through each row of tokenized words
for tokens in df['filtered_response']:
    # Convert tokens to indices using the tokens_to_indices function
    indices = tokens_to_indices(tokens, vocab)
    # Append the indices for the current row to the list
    all_indices.append(indices)

# Print the indices for all rows
print(all_indices[0:5])
```

[[4080, 5202, 4488, 1736, 4050, 3551, 7537, 3226, 4502, 7053, 2300], [187, 5923, 6162, 2493, 7815, 4000],

```
def tokens_to_indices(tokens, vocab):
    indices = []
    # Iterate through each token in the tokens
    for token in tokens:
        # Check if the token exists in the vocab
        if token in vocab:
            # Map the token to its corresponding index
            indices.append(vocab[token])
    return indices
```


Data Preparation

- NEXT, I PADDED ALL SEQUENCES TO HAVE THE SAME LENGTH (59).
- THEN I IDENTIFIED MY X_TRAIN WHICH IS THE PADDED SEQUENCES.
- I CREATED A DICTIONARY AND ASSIGNED EACH TAG TO AN INTEGER INDEX.
- THE INTEGER INDICES WILL BE USED FOR MY Y_TRAIN

```
# Function to pad sequences to a fixed length
def pad_sequences(sequences, max_length):
    padded_sequences = []
    # Iterate through each sequence in the list of sequences
    for sequence in sequences:
        # Pad the sequence with zeros if its length is less than the max_length
        padded_sequence = sequence[:max_length] + [0] * max(0, max_length - len(sequence))
        # Append the padded sequence to the list of padded sequences
        padded_sequences.append(padded_sequence)
    return padded_sequences

max_length = 59

# Pad sequences to the maximum length
padded_sequences = pad_sequences(all_indices, max_length)
print(padded_sequences[0:5])
```

```
[ ] # Create a dictionary mapping each unique tag to an integer index
unique_tags = set()

# Iterate through each row in the "ner_results" column
for ner_tags in df['pos_results']:
    # Extract the tags from each row and add them to the set
    for tag in ner_tags:
        unique_tags.add(tag[1])

tag2idx = {tag: idx for idx, tag in enumerate(unique_tags)}

# Print the tag-to-index mapping
print(tag2idx)
```

```
{'INTJ': 0, 'VERB': 1, 'PUNCT': 2, 'PRON': 3, 'ADV': 4, 'SCONJ': 5, 'ADJ': 6, 'X': 7, 'SPACE': 8, 'NUM': 9, 'SYM': 10, 'ADP': 11, 'NOUN': 12, 'DET': 13, 'PROPN': 14, 'PART': 15, 'AUX': 16}
```

Model Training

- A SEQUENTIAL LSTM BASED NEURAL NETWORK MODEL WAS BUILT USING TENSORFLOW/KERAS
- THE MODEL WAS TRAINED FOR 5 EPOCHS WITH A BATCH SIZE OF 128.
- EMBEDDING DIMENSIONALITY OF 200 WAS USED.
- THERE ARE 17 UNIQUE TAGS IN MY DATASET

```
[ ] epochs = 5
```

```
history = model.fit(x_train, y_train_np, epochs=epochs, batch_size=128)
```

```
Epoch 1/5  
1947/1947 [=====] - 561s 285ms/step - loss: 0.0468 - accuracy: 0.9884  
Epoch 2/5  
1947/1947 [=====] - 548s 282ms/step - loss: 0.0075 - accuracy: 0.9975  
Epoch 3/5  
1947/1947 [=====] - 551s 283ms/step - loss: 0.0054 - accuracy: 0.9981  
Epoch 4/5  
1947/1947 [=====] - 549s 282ms/step - loss: 0.0044 - accuracy: 0.9985  
Epoch 5/5  
1947/1947 [=====] - 546s 281ms/step - loss: 0.0037 - accuracy: 0.9987
```

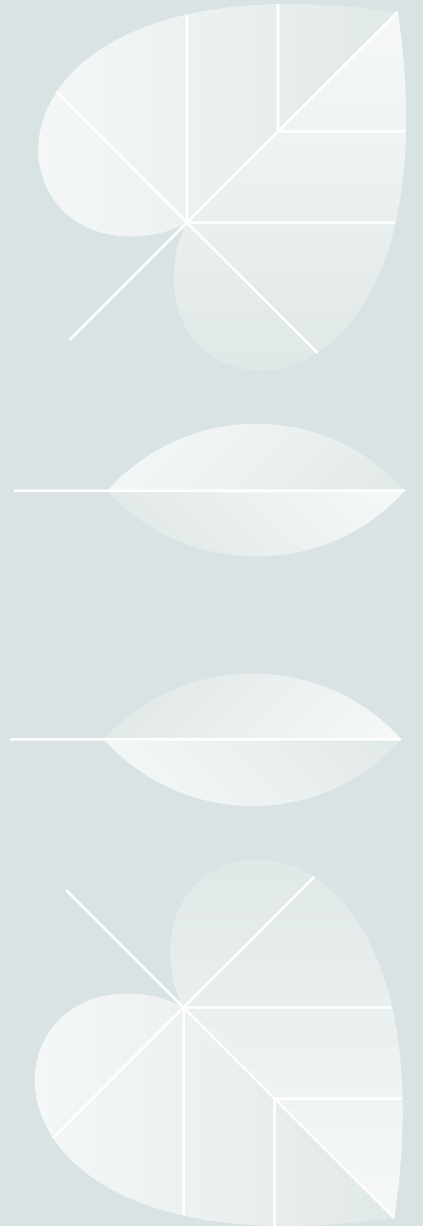
Model: "sequential"

Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 59, 200)	1581600
bidirectional (Bidirectional)	(None, 59, 128)	135680
dense (Dense)	(None, 59, 17)	2193

=====
Total params: 1719473 (6.56 MB)
Trainable params: 1719473 (6.56 MB)
Non-trainable params: 0 (0.00 Byte)

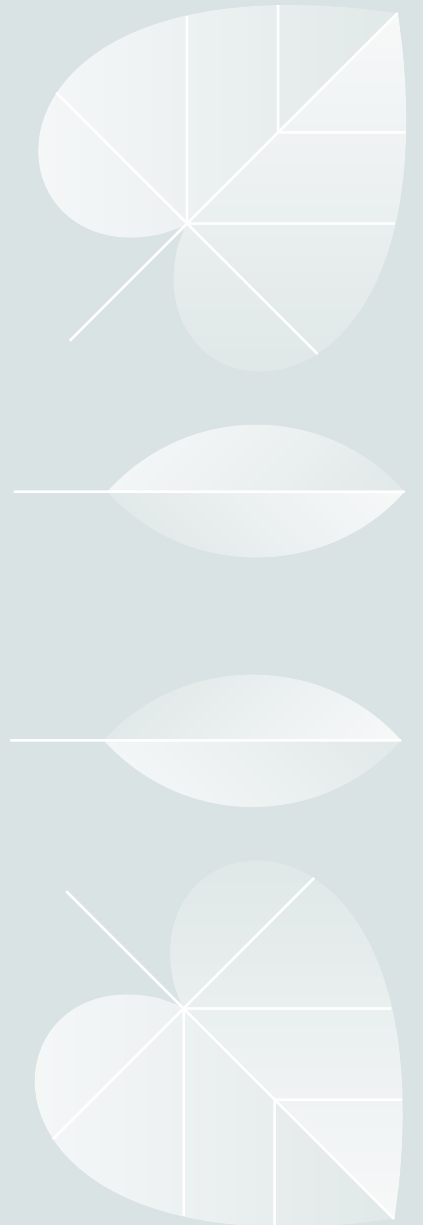
Architecture

- 1) Embedding layer
- Takes 3 parameters: The first parameter is input dimensionality, which represents the length of my vocabulary.
- The second parameter is the output dimensionality, which represents the dimension of the dense embedding vectors.
- The last one is the input length, which represents the length of input sequences (the number of words in each sequence), which is 59 in my project.



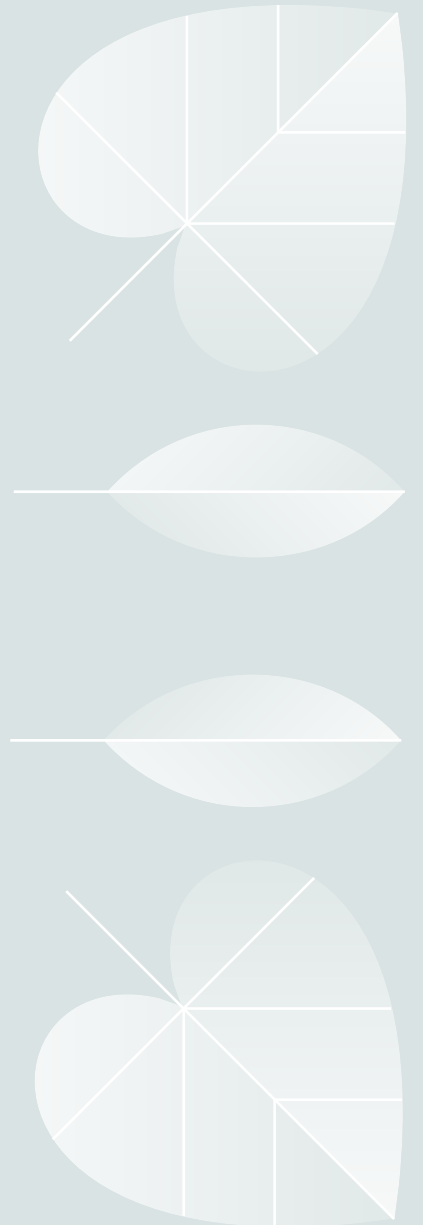
Architecture

- 2) BI-LSTM layer
- processes the sequence data in both forward and backward directions.
- The Bi-LSTM layer takes as a parameter the number of LSTM units (64) which determines the output dimensionality of the LSTM
- A Bidirectional LSTM runs two LSTMs on the input sequence: one from the start to the end (forward LSTM) and another from the end to the start (backward LSTM).T
- The outputs of these two LSTMs are concatenated at each time step, providing the model with information from both past and future contexts for each point in the sequence



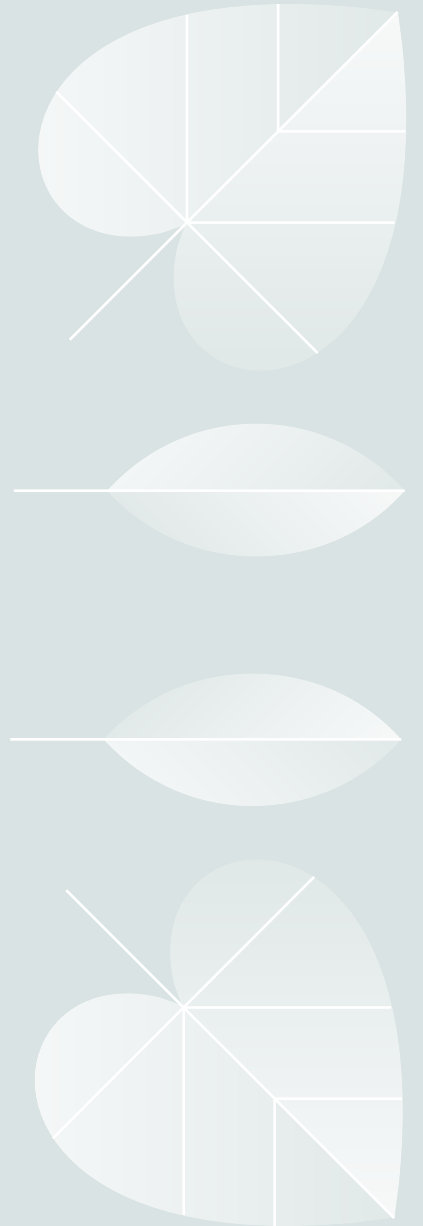
Architecture

- 3) Dense layer
- It takes the output from the LSTM layer and produces a probability distribution over the (tags) for each time step in the sequence.
- The first parameter that this layer takes is the number of tags which is 17 in my dataset.
- The next parameter is the activation function which is softmax. The softmax activation function converts the raw output scores (logits) from the Dense layer into probabilities.
- Finally, i compiled the model using "Adam" optimizer, and used the "sparse categorical crossentropy" loss function.



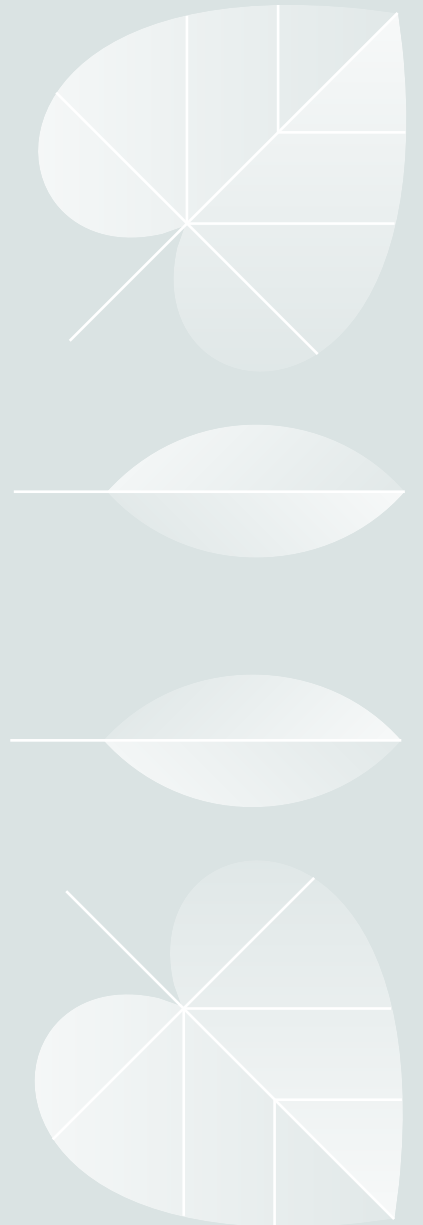
Advantages

- **Contextual understanding:** POS tagging requires understanding the context in which a word appears as the same word can have different parts of speech depending on its usage. The Bidirectional LSTM layer allows the model to consider both the preceding and succeeding words in a sequence, enabling a more comprehensive understanding of context.
- **sequential data handling.** The use of LSTM units helps in managing sequential data effectively. LSTMs are designed to remember long-term dependencies, which means they can keep track of the entire context of a sentence, even if the relevant information is spread out over many words.



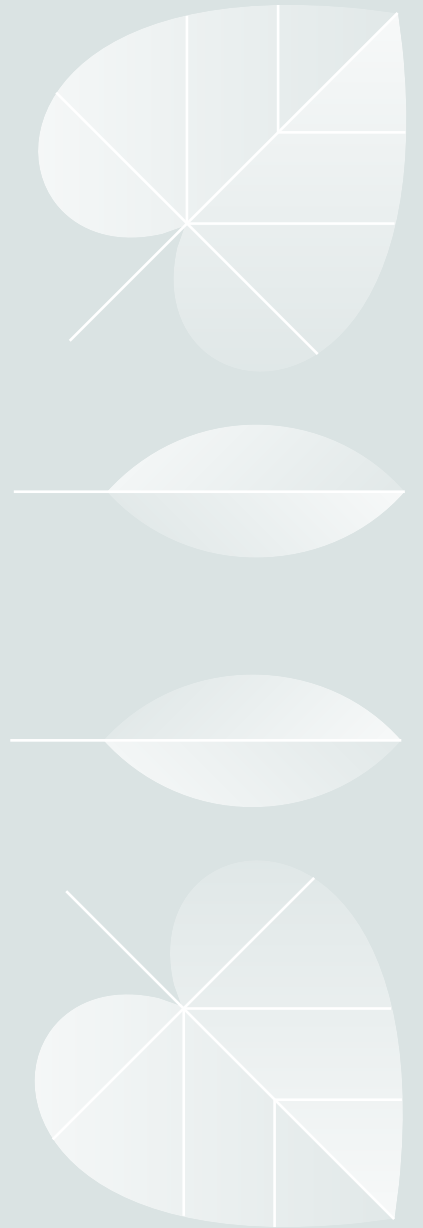
Advantages

- **Simplicity:** The LSTM-based architecture is relatively simple and intuitive, making it easy to understand and implement.
- **Interpretability:** Each layer of the LSTM model represents a clear computational step, allowing for easier interpretation of how the model processes input sequences.
- **Training Speed:** LSTM models can be trained relatively quickly due to their simpler architecture.
- **Performance:** My model had an accuracy of 0.9987.



Limitations

- Bidirectional LSTM layers are computationally expensive, particularly when dealing with large vocabularies and sequences.

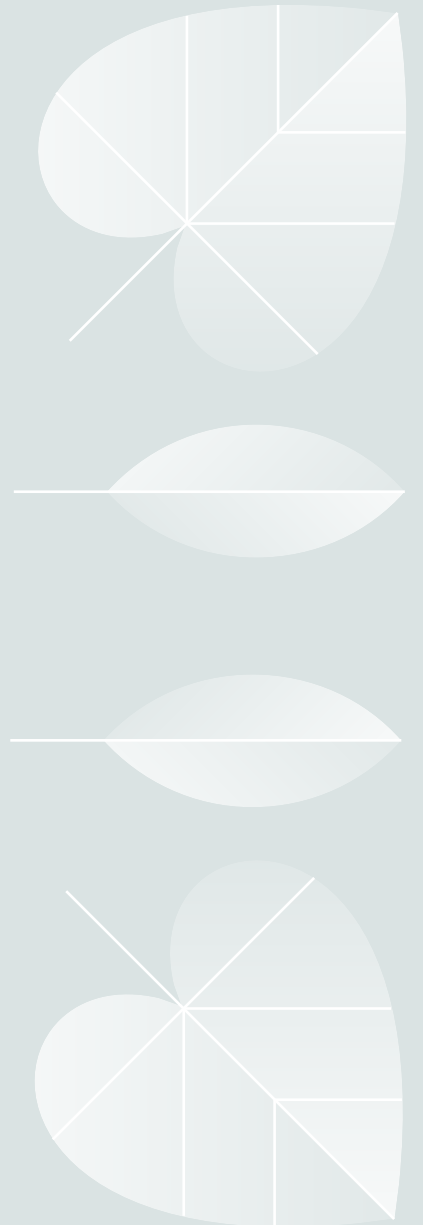


Evaluation results

- Before evaluating the model, I identified my X-Train and y-train. My x-train is the integer representation of tokens (words) of my dataset, while the y-train is the tags.
- I evaluated the model on my test data and the results yielded an accuracy of 0.9989.
- Finally, i compared the results of 10 predicted tags with the actual tags, and the model successfully predicted all the tags correct.

```
[ ] loss, accuracy = model.evaluate(x_test, y_test)
    print(f"Training Loss: {loss:.4f}, Training Accuracy: {accuracy:.4f}")
```

```
➡ 1557/1557 [=====] - 58s 36ms/step - loss: 0.0032 - accuracy: 0.9989
   Training Loss: 0.0032, Training Accuracy: 0.9989
```



Example of actual vs predicted tags

```
The actual tags for prompt number 0 is :  
[ 1 12 12 6 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
The predicted tags for prompt number 0 is :  
[ 1 12 12 6 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
-----  
The actual tags for prompt number 1 is :  
[ 1 6 12 1 6 12 6 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
The predicted tags for prompt number 1 is :  
[ 1 6 12 1 6 12 6 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
-----  
The actual tags for prompt number 2 is :  
[ 6 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```

```
The predicted tags for prompt number 2 is :  
[ 6 12 12 12 12 12 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0]
```





Milestone 3

Defining the model and tokenizer

```
from transformers import BertTokenizer, BertForTokenClassification

# Define the pre-trained TinyBERT model name
model_name = "prajjwal1/bert-tiny"

# Load the tokenizer for the TinyBERT model
tokenizer = BertTokenizer.from_pretrained(model_name)

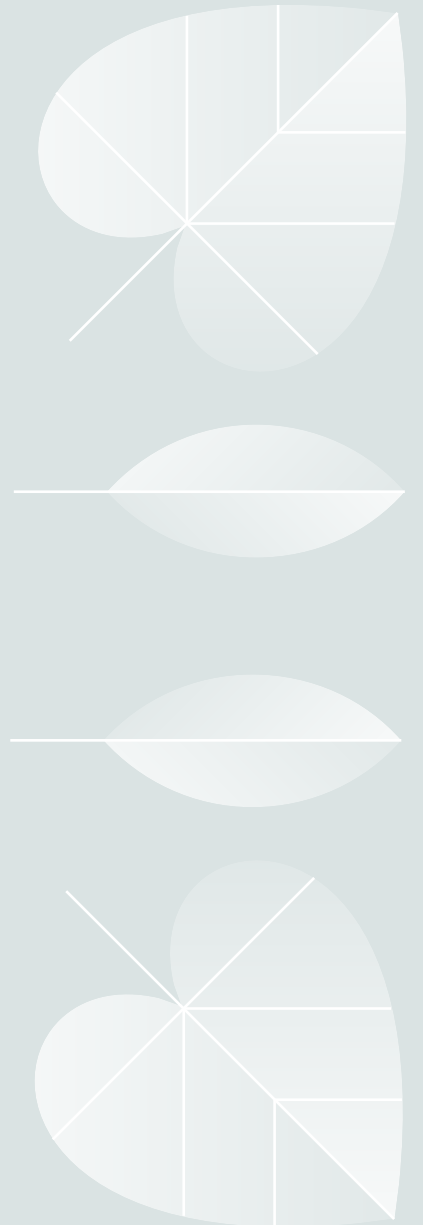
# Load the pre-trained TinyBERT model for token classification
model = BertForTokenClassification.from_pretrained(model_name)
```

Data Preparation

- Tokenized words in my dataset using BERT tokenizer. (tokenizer.tokenize)
- Converted the tokenized words to token IDS using tokenizer.convert_tokens_to_ids.

```
# Initialize an empty list to store tokenized sequences
tokenized_sequences = []

# Iterate over the values in the "filtered_response" column
for sequence in df["filtered_response"]:
    # Convert the value to string
    sequence_str = str(sequence)
    # Tokenize the sequence
    tokens = tokenizer.tokenize(sequence_str)
    # Convert tokens to token IDs
    token_ids = tokenizer.convert_tokens_to_ids(tokens)
    # Append token IDs to the list
    tokenized_sequences.append(token_ids)
```

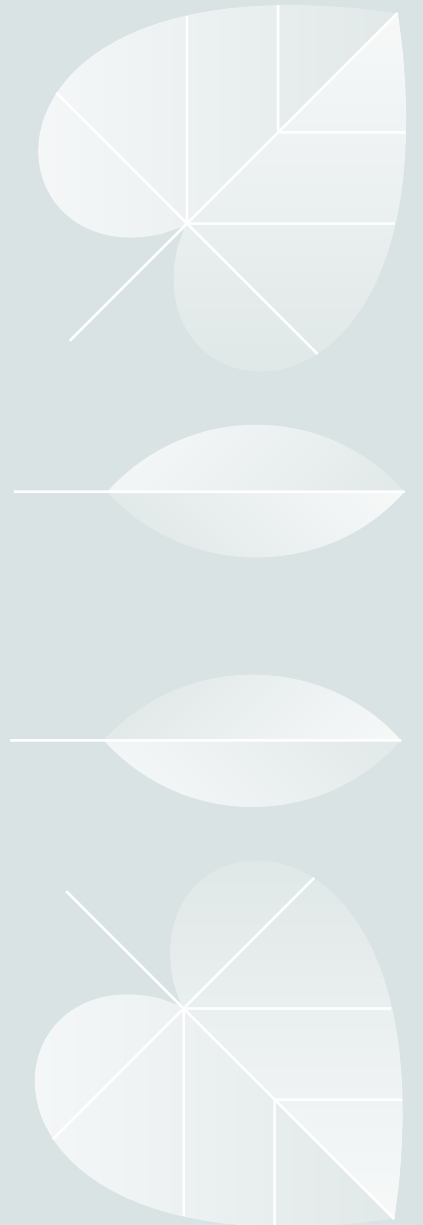


Data preparation

- Padded the tokenized sequences to the max length using `pad_sequences` from `tensorflow.keras.preprocessing.sequence`

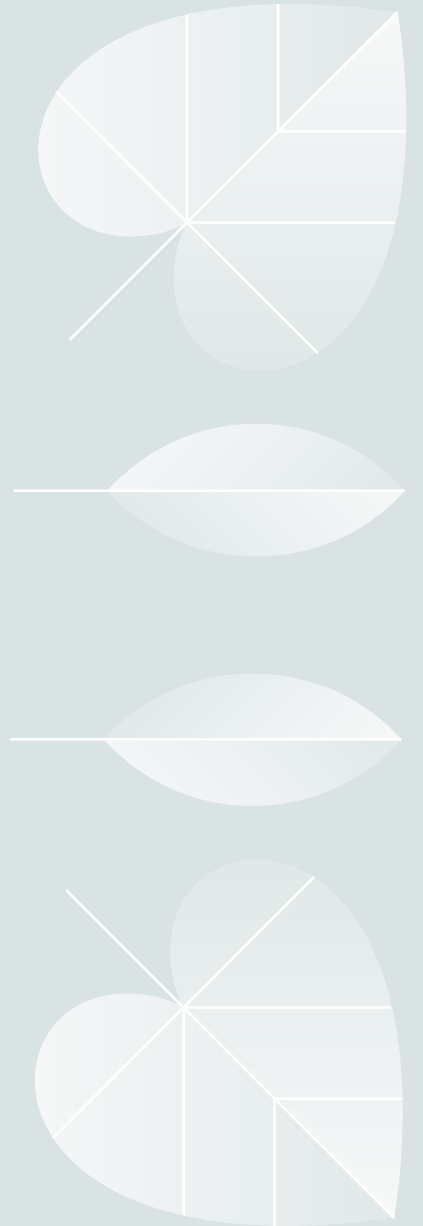
```
[ ] from tensorflow.keras.preprocessing.sequence import pad_sequences

# Pad tokenized sequences to a maximum length
max_length = 59
padded_sequences = pad_sequences(tokenized_sequences, maxlen=max_length, padding='post', truncating='post')
```



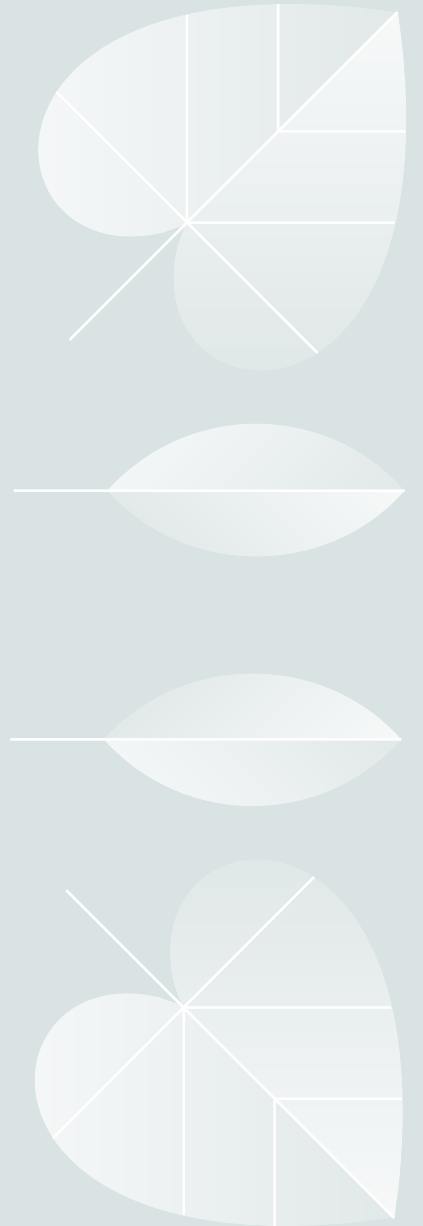
Architecture

- I converted tokenized input sequences and their corresponding POS tags into PyTorch tensors. Then, i created a tensor dataset by combining the input IDs and labels.
- The next step is initializing the data loader object which provides an efficient way to iterate over the dataset in batches, with shuffling for randomness.
- After that, i initialized the pre-trained TinyBERT model for token classification, specifying the number of POS tags.
- I used Adam optimizer to update model parameters during training.



Architecture

- The training loop started with batch processing. Within each epoch, the dataset is processed in batches to manage memory efficiently and to enable faster convergence.
- Before performing backpropagation, the gradients of all model parameters are reset to zero. The next step is forward pass. The input batch is passed through the model to compute the output and loss.
- Next comes backward pass. Gradients are computed through backpropagation based on the loss.
- Then the optimizer updates the model parameters based on the computed gradients.



Model Training

```
import torch
from transformers import BertTokenizer, BertForTokenClassification, AdamW
from torch.utils.data import TensorDataset, DataLoader, RandomSampler
from tqdm import tqdm

# Define hyperparameters
batch_size = 128
num_epochs = 3
learning_rate = 7e-5

# Convert input sequences and labels to PyTorch tensors
input_ids = torch.tensor(padded_sequences)
labels = torch.tensor(padded_y_train)

# Create TensorDataset
dataset = TensorDataset(input_ids, labels)

# Create DataLoader for batching and shuffling
dataloader = DataLoader(dataset, batch_size=batch_size, sampler=RandomSampler(dataset))

# Load pre-trained TinyBERT model
model = BertForTokenClassification.from_pretrained(model_name, num_labels=num_classes)

# Define optimizer
optimizer = AdamW(model.parameters(), lr=learning_rate)

# Fine-tune the model
for epoch in range(num_epochs):
    print(f"Epoch {epoch + 1}/{num_epochs}")
    progress_bar = tqdm(enumerate(dataloader), total=len(dataloader))
    model.train()
    for step, batch in progress_bar:
        # Unpack batch
        input_ids_batch, labels_batch = batch
```

```
model.train()
for step, batch in progress_bar:
    # Unpack batch
    input_ids_batch, labels_batch = batch

    # Zero gradients
    optimizer.zero_grad()

    # Forward pass
    outputs = model(input_ids_batch, labels=labels_batch)
    loss = outputs.loss

    # Backward pass
    loss.backward()

    # Update parameters
    optimizer.step()

    progress_bar.set_description(f"Loss: {loss.item():.4f}")

# Save the fine-tuned model
model.save_pretrained("fine_tuned_tinybert")
```

```
Some weights of BertForTokenClassification were not initialized from the
You should probably TRAIN this model on a down-stream task to be able to
/usr/local/lib/python3.10/dist-packages/transformers/optimization.py:521
warnings.warn(
Epoch 1/3
0% | 0/1947 [00:00<?, ?it/s] We strongly recommend passing in
Loss: 0.0576: 100% | 1947/1947 [18:40<00:00, 1.74it/s]
Epoch 2/3
Loss: 0.0354: 100% | 1947/1947 [18:19<00:00, 1.77it/s]
Epoch 3/3
Loss: 0.0264: 100% | 1947/1947 [18:20<00:00, 1.77it/s]
```

Model Evaluation

- First, I converted the `x_train` and `y_train` to pytorch tensors because the model expects the input in the form of pytorch tensors.
- The model had an accuracy of 0.8561, which is lower than the LSTM model

```
from torch.utils.data import TensorDataset, DataLoader

# Convert x_test and y_test to PyTorch tensors
input_ids_test = torch.tensor(x_test)
labels_test = torch.tensor(y_test)

# Create TensorDataset for test data
dataset_test = TensorDataset(input_ids_test, labels_test)

# Define batch size for test DataLoader
batch_size_test = 128

# Create DataLoader for test data
dataloader_test = DataLoader(dataset_test, batch_size=batch_size_test)
```

```
Evaluation: 100% |██████████| 390/390 [01:12<00:00, 5.40it/s]
Accuracy: 0.8561788342422956
```

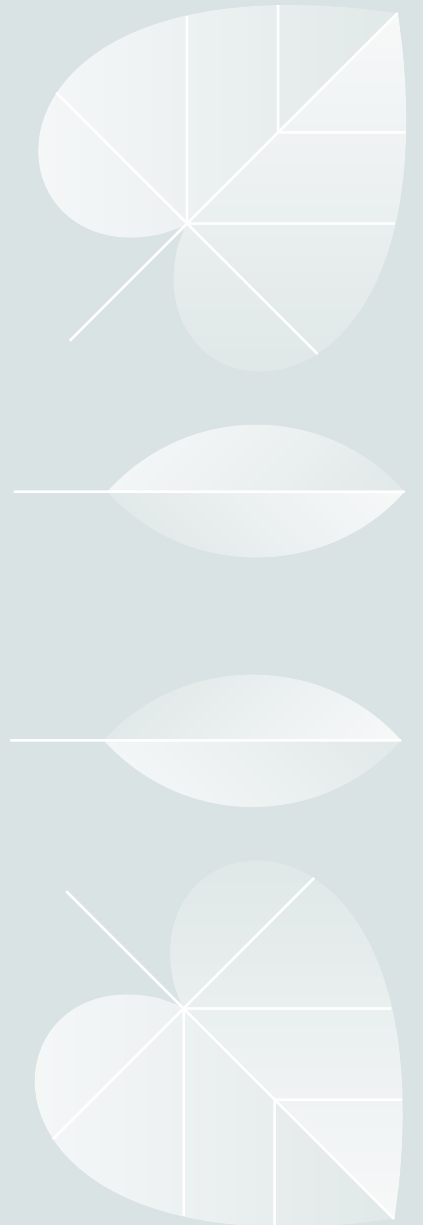
Advantages

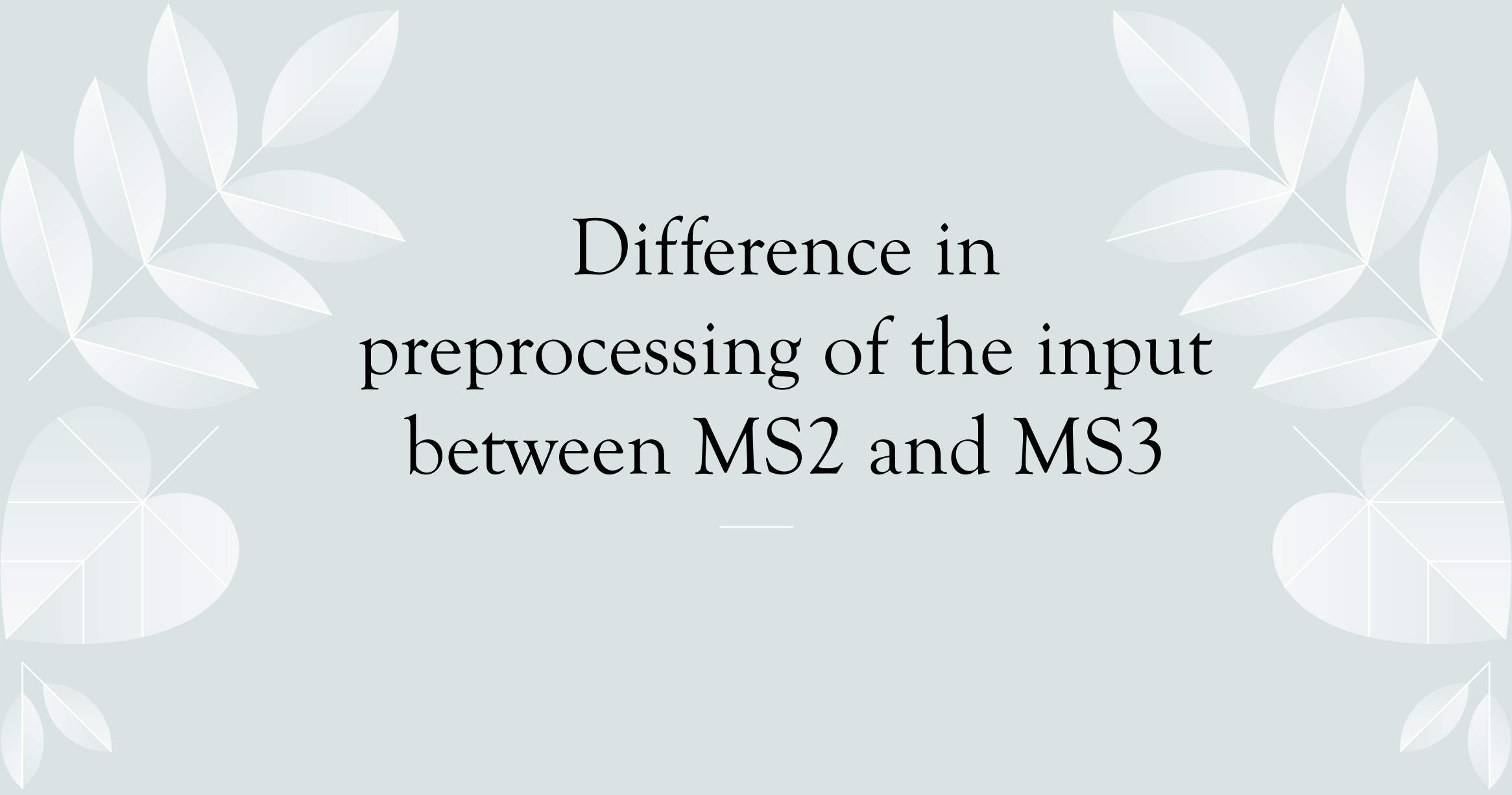
- **Efficiency:** Despite being a smaller variant of BERT, TinyBERT still benefits from the efficiency and scalability of transformer architectures, making it suitable for a wide range of NLP tasks.
- **Self-Attention Mechanism:** The self-attention mechanism in transformers allows them to capture dependencies between all tokens in a sequence, regardless of their distance from each other, leading to better handling of long-range dependencies.



Limitations

- **Complexity and Interpretability:** The architecture of transformers is more complex, making them harder to interpret compared to simpler models like LSTMs
- **Computational Resources:** Transformers require significant computational resources, both in terms of memory and processing power.
- **Efficiency:** in my case, BERT performed worse than the LSTM model. BERT had an accuracy of 0.8561 compared to the LSTM which had 0.9987.



The slide features a light gray background with decorative white line art of leaves in the corners. The top-left and top-right corners each contain a cluster of several elongated, pointed leaves. The bottom-left and bottom-right corners each contain a single, larger, rounded leaf with internal vein details. The central text is in a black serif font.

Difference in preprocessing of the input between MS2 and MS3

Milestone 2

Tokenization: Each sentence is tokenized into words or sub-words.

Vocabulary Creation: I created a vocabulary from the words in my dataset. Each unique word is assigned an index.

Encoding Tokens: Convert tokens into their corresponding integer indices based on the created vocabulary.

Padding Sequences: Since sentences have varying lengths, I padded them to ensure that all sequences have the same length.

Milestone 3

- Load the pre-trained tokenizer specific to the TinyBERT model.
 - Sentences are tokenized into sub-word units, and special tokens like '[CLS]' (classification token) and '[SEP]' (separator token) are added to the sequences.
 - **Padding:** Ensure that all tokenized sequences are of the same length by padding. The tokenizer's pad_sequences is used.
 - converted tokenized input sequences and their corresponding POS tags into PyTorch tensors
- Created a tensor dataset by combining the input IDs and labels.
 - Initialized the data loader object which provides an efficient way to iterate over the dataset in batches, with shuffling for randomness.

The slide features a light gray background with decorative white line art of leaves in the corners. The top-left and top-right corners each contain a cluster of several elongated, pointed leaves. The bottom-left and bottom-right corners each contain a single, larger, heart-shaped leaf with internal vein details. A thin horizontal line is positioned below the main title.

Reasons for Choosing TinyBERT

1) Efficiency

TinyBERT is a smaller, more efficient variant of BERT, making it faster and less resource-intensive while still providing strong performance.

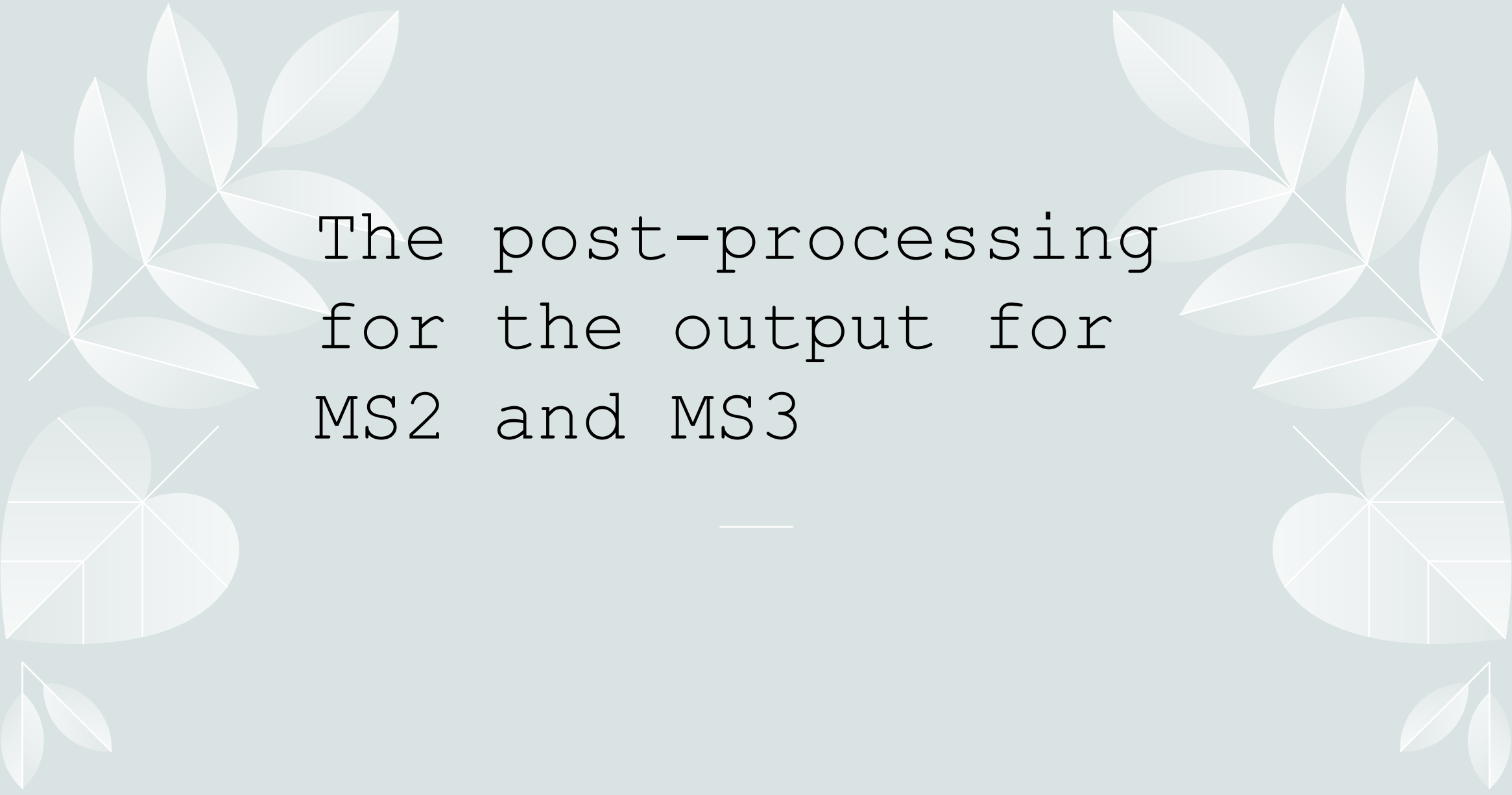
Its reduced computational requirements make it suitable for environments with limited resources

2) Performance

TinyBERT retains competitive accuracy and performance.

3) **Robustness to Out-of-Vocabulary Words:** BERT's subword tokenization mechanism allows it to handle out-of-vocabulary words by breaking them down into subword units. This can be particularly beneficial for POS tagging tasks where encountering rare or unseen words is common.

4) **Bidirectional Context:** BERT utilizes a bidirectional architecture, meaning it considers both left and right context when encoding each word. This bidirectionality helps in capturing the context around each word more effectively.

The slide features a light gray background with decorative white line art of leaves and branches in the corners. The text is centered in a monospaced font.

The post-processing
for the output for
MS2 and MS3

Milestone 2

A decorative graphic of several overlapping, stylized leaves in a light gray color, positioned on the left side of the slide.

- The model outputs a sequence of probabilities for each token in the input sequence, with each probability corresponding to a POS tag.
- For each token in the sequence, I take the argmax of the predicted probabilities to get the index of the most likely POS tag.

Milestone 3

The model outputs logits for each token in the sequence. Logits are the raw scores for each POS tag.

Similar to the LSTM model, I take the argmax of the logits to get the index of the most likely POS tag for each token.

A decorative graphic of several overlapping, stylized leaves in a light gray color, positioned on the left side of the slide.

Conclusion

- The LSTM model had an accuracy of 0.9989.
- The BERT model had an accuracy of 0.8561.
- The LSTM model performed better than the BERT model. This could be due to Insufficient Fine-Tuning.
- While BERT excels in many NLP tasks, for simpler sequence labeling tasks like POS tagging with limited data, LSTM models might sometimes perform better due to their simpler and more direct approach to handling sequences.



Thank you

Any questions ?