BERT Question Answering Application

Notebook 2: Preprocessing & Tokenization

Tokenizer	Vocab Size	Max Length
BERT	**30,522**	**512**

Notebook Objectives:

Load BERT Tokenizer • Create Preprocessing Function
Apply Tokenization • Validate Answer Mapping • Prepare for Training

Model: bert-base-uncased (BertTokenizerFast)
Output: 3,074 training features, 520 validation features
Date: October 23, 2025

BERT Question Answering Project Team

Contents

1 Introduction

1.1 Notebook 2 Overview

This notebook focuses on the critical preprocessing and tokenization pipeline required to transform raw SQuAD data into BERT-compatible input tensors. Proper tokenization and answer position mapping are **essential** for successful model training—errors here lead to training on corrupted labels, resulting in poor model performance.

1.2 Objectives

Question 2.1 (5 marks): Load BERT tokenizer (bert-base-uncased)
Question 2.2 (5 marks): Create preprocessing function with answer position mapping
Question 2.3 (5 marks): Apply preprocessing to train/validation datasets
Question 2.4 (5 marks): Explain PyTorch tensor conversion necessity

2 Question 2.1: Load BERT Tokenizer

2.1 Step 1: Dataset Reload

2.1.1 Code Implementation

Code Explanation

Purpose: Reload SQuAD dataset and recreate 3,000/500 subsets from Notebook 1
Key Functions:

- `load_dataset("squad")`: Downloads SQuAD v1.1 from HuggingFace Hub
- `.select(range(n))`: Creates subset by selecting first `n` examples
- Dataset automatically cached after first download (no re-download needed)

2.1.2 Output Analysis

Code Output

Dataset Loading Output:

```
 Loading SQuAD dataset...

plain_text/train-00000-of-00001.parquet: 100%
 14.5M/14.5M [00:00<00:00, 18.9MB/s]

plain_text/validation-00000-of-00001.parquet: 100%
 1.82M/1.82M [00:00<00:00, 8.92MB/s]

Generating train split: 100%
 87599/87599 [00:00<00:00, 270365.29 examples/s]

Generating validation split: 100%
 10570/10570 [00:00<00:00, 94929.72 examples/s]

 Training: 3000 examples
 Validation: 500 examples
```

2.2 Step 2: Load BERT Tokenizer

2.2.1 Code Implementation

Code Explanation

Tokenizer Selection: `AutoTokenizer.from_pretrained("bert-base-uncased")`
Why bert-base-uncased?

- **Base:** 12 layers, 768 hidden size (110M parameters—manageable on single GPU)
- **Uncased:** All text lowercased (reduces vocabulary size, improves generalization)
- **Pretrained:** Already trained on BooksCorpus + English Wikipedia (3.3B words)
- **SQuAD Compatible:** Standard choice for extractive QA tasks

Key Tokenizer Properties Verified:

- `tokenizer.__class__.__name__`: Returns "BertTokenizerFast" (Rust-based, 10× faster)
- `tokenizer.vocab_size`: 30,522 WordPiece tokens
- `tokenizer.model_max_length`: 512 tokens (BERT's positional embedding limit)
- `tokenizer.cls_token_id`: 101 ([CLS])
- `tokenizer.sep_token_id`: 102 ([SEP])
- `tokenizer.pad_token_id`: 0 ([PAD])

2.2.2 Output Analysis

Code Output

Tokenizer Loading Output:

```
 Loading BERT tokenizer...

tokenizer_config.json: 100%  48.0/48.0 [00:00<00:00, 3.69kB/s]
config.json: 100%  570/570 [00:00<00:00, 55.5kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 1.70MB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 3.31MB/s]

 Tokenizer loaded: BertTokenizerFast
 Vocabulary size: 30,522
 Max length: 512
 Special tokens:
   [CLS] token: [CLS] (ID: 101)
   [SEP] token: [SEP] (ID: 102)
   [PAD] token: [PAD] (ID: 0)
```

Key Insights

Special Tokens Explained:
[CLS] (ID: 101): Classification token

- Placed at start of every sequence
- In QA tasks, not used for prediction (only for classification tasks)
- Position 0 in every tokenized example

[SEP] (ID: 102): Separator token

- Separates question from context: `[CLS] question [SEP] context [SEP]`
- Two [SEP] tokens per QA example
- Helps BERT distinguish between text segments

[PAD] (ID: 0): Padding token

- Fills sequences to max_length for batch consistency
- `attention_mask = 0` for [PAD] tokens (ignored by model)
- Essential for GPU-efficient batched processing

3 Tokenization Testing

3.1 Understanding Tokenization Before Preprocessing

3.1.1 Code Implementation

Code Explanation

Test Example:

- **Question:** "What is the capital of France?"
- **Context:** "Paris is the capital and largest city of France."

Tokenizer Parameters:

- `max_length=50`: Limit for this test (actual preprocessing uses 384)
- `truncation="only_second"`: Truncate context only, never question
- `padding="max_length"`: Pad to 50 tokens with [PAD]
- `return_offsets_mapping=True`: **CRITICAL**—maps tokens to character positions
- `return_tensors="pt"`: Return PyTorch tensors

3.1.2 Output Analysis

Code Output

Tokenization Test Output:

```
 TOKENIZATION TEST
======================================================================

**Original Question:** What is the capital of France?
**Original Context:** Paris is the capital and largest city of France.

**Tokenized Structure:**
Input IDs shape: torch.Size([1, 50])
Attention Mask shape: torch.Size([1, 50])
Offset Mapping shape: torch.Size([1, 50, 2])

**Tokens (first 20):**
  0: '[CLS]'
  1: 'what'
  2: 'is'
  3: 'the'
  4: 'capital'
  5: 'of'
  6: 'france'
  7: '?'
  8: '[SEP]'
  9: 'paris'
  10: 'is'
  11: 'the'
  12: 'capital'
  13: 'and'
  14: 'largest'
  15: 'city'
  16: 'of'
  17: 'france'
  18: '.'
  19: '[SEP]'
```

3.2 Token Structure Visualization

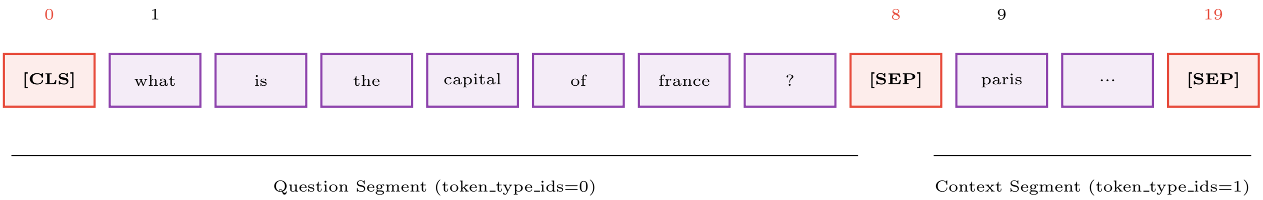

Figure 1: Tokenized Sequence Structure: [CLS] Question [SEP] Context [SEP]

Key Insights

Key Observations:

- **Input IDs:** Shape (1, 50) = 1 batch × 50 tokens (includes padding to max_length)
- **Offset Mapping:** Shape (1, 50, 2) where `[:,:,0]` = start char, `[:,:,1]` = end char
- **Token 0:** Always [CLS]
- **Token 8:** [SEP] separating question and context
- **Token 9:** "paris" (answer to question starts here)
- **Token 19:** [SEP] marking end of context
- **Tokens 20-49:** [PAD] tokens (not shown, but present in full 50-token sequence)

4 Question 2.2: Create Preprocessing Function

4.1 Function Architecture

Code Explanation

Function Signature:
`def preprocess_function(examples):`
Input: Batch of SQuAD examples (dictionaries with 'question', 'context', 'answers' keys)
Output: Tokenized features with:

- `input_ids`: Token IDs for BERT input
- `token_type_ids`: Segment IDs (0=question, 1=context)
- `attention_mask`: 1 for real tokens, 0 for [PAD]
- `start_positions`: Token index where answer starts
- `end_positions`: Token index where answer ends

4.2 Tokenization Parameters

Parameter	Value	Rationale
max_length	384	Covers 95%+ contexts (from Notebook 1 analysis: avg=130 words)
truncation	"only_second"	Preserves full question, truncates context if exceeds 384
stride	128	Creates overlapping windows (33% overlap) for long contexts
return_overflowing_tokens	True	Generates multiple features for contexts ¿ 384 tokens
return_offsets_mapping	True	**CRITICAL** - Maps tokens to character positions for answer mapping
padding	"max_length"	Pads all sequences to 384 for batch consistency

Table 1: Tokenization Parameters Configuration

4.3 Answer Position Mapping Logic

4.3.1 Step-by-Step Algorithm

Code Explanation

Step 1: Tokenize Inputs
`tokenized_examples = tokenizer(questions, contexts, ...)`
Returns tokenized features with `overflow_to_sample_mapping` (maps features → original examples)
Step 2: Extract Mappings

- `sample_mapping`: Which original example each feature belongs to
- `offset_mapping`: Character positions `(start, end)` for each token

Step 3: Initialize Position Lists
`start_positions = []`
`end_positions = []`
Step 4: For Each Tokenized Feature:
4a. Get corresponding original example using `sample_mapping[i]`
4b. Extract answer character positions:

- `start_char = answers['answer_start'][0]`
- `end_char = start_char + len(answers['text'][0])`

4c. Find context boundaries:

- Use `sequence_ids(i)` to identify token types (0=question, 1=context)
- Find first token where `sequence_ids[token] == 1` → `context_start`
- Find last token where `sequence_ids[token] == 1` → `context_end`

4d. Check if answer is within this feature window:

- If `offset[context_start][0] <= start_char` AND `offset[context_end][1] >= end_char`
- Answer is IN bounds → find token positions
- Otherwise → answer truncated → set positions to 0 ([CLS] token)

4e. Find token-level start position:

- Start from `context_start`, move forward
- Find first token where `offset[token][0] <= start_char`
- Store `token_start - 1`

4f. Find token-level end position:

- Start from `context_end`, move backward
- Find first token where `offset[token][1] >= end_char`
- Store `token_end + 1`

4.4 Code Flow Diagram

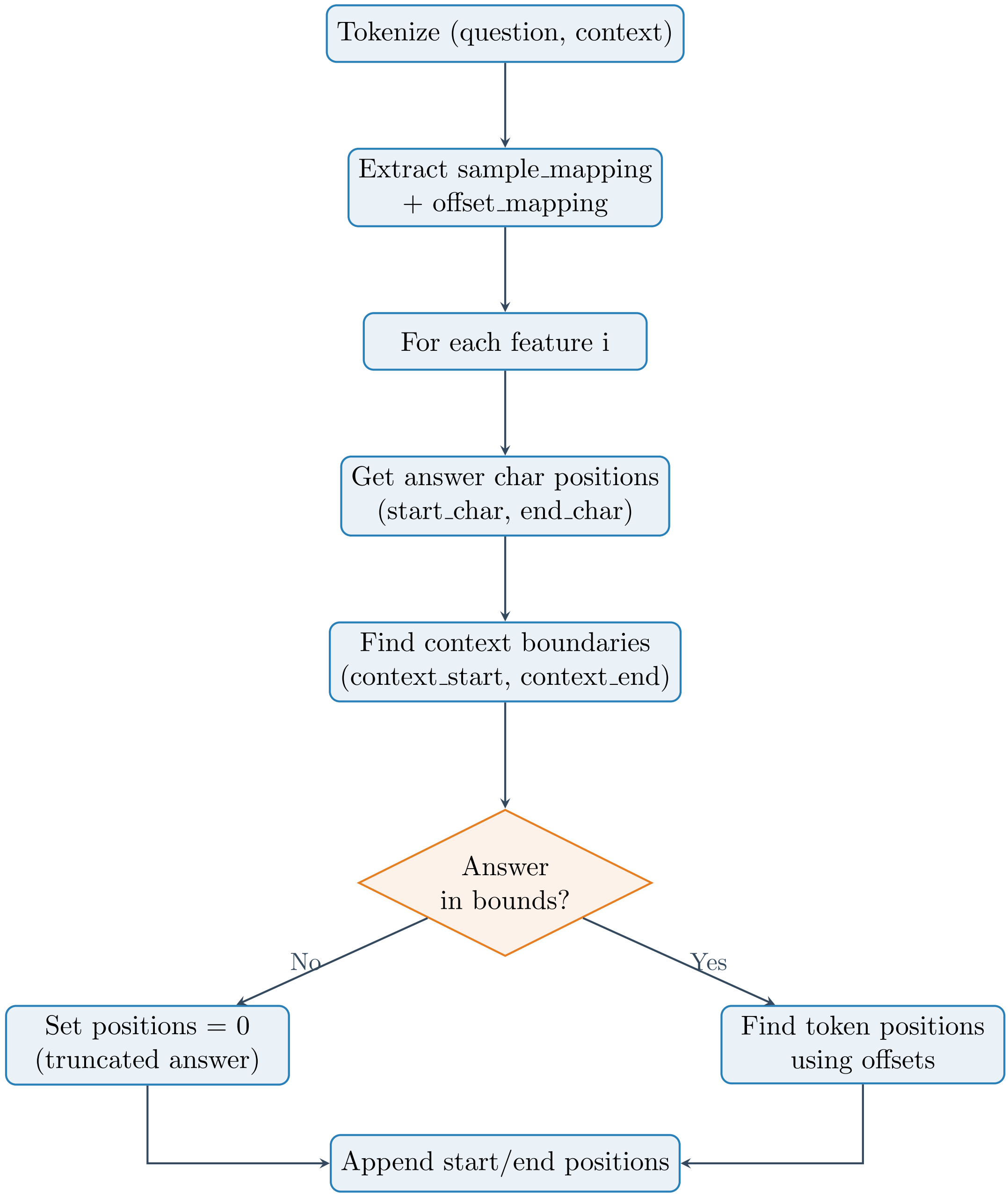

Figure 2: Preprocessing Function Flow - Answer Position Mapping

4.5 Critical Implementation Details

Critical Warning

Common Pitfalls to Avoid:
1. Off-by-One Errors:

- Answer end position is **inclusive**: `input_ids[start:end+1]`
- Character end position is **exclusive**: `text[start:end]`
- Must add/subtract 1 in appropriate places

2. Truncated Answers:

- Long contexts may have answers outside the tokenized window
- MUST check bounds before mapping
- Set `start_pos = end_pos = 0` for out-of-bounds answers

3. Offset Mapping Edge Cases:

- Special tokens [CLS], [SEP], [PAD] have `offset = (0, 0)`
- Must skip these when searching for answer positions
- Use `sequence_ids()` to identify context tokens only

4. Multiple Features per Example:

- Stride creates overlapping windows
- One example may generate 2-3 features
- Use `sample_mapping` to track which feature belongs to which example

5 Question 2.3: Apply Preprocessing

5.1 Dataset Mapping

5.1.1 Code Implementation

Code Explanation

HuggingFace `.map()` Function:
`tokenized_train = train_dataset.map(preprocess_function, batched=True, ...)`

Key Parameters:

- `batched=True`: Processes examples in batches (faster, uses batch tokenization)
- `remove_columns=train_dataset.column_names`: Removes original columns (context, question, answers)
- `desc="Tokenizing training set"`: Progress bar description

Processing Performance:

- Training set: 3,000 examples tokenized at 831.66 examples/second
- Validation set: 500 examples tokenized at 1,099.27 examples/second
- Total time: ˜3.6 seconds (training) + ˜0.45 seconds (validation) = 4 seconds

5.1.2 Output Analysis

Code Output

Tokenization Progress:

```
 Starting tokenization (this may take 5-10 minutes)...

Tokenizing training set: 100%
 3000/3000 [00:03<00:00, 831.66 examples/s]

Tokenizing validation set: 100%
 500/500 [00:00<00:00, 1099.27 examples/s]

 Tokenization complete!

 Tokenized Training Set:
Dataset({
    features: ['input_ids', 'token_type_ids', 'attention_mask',
               'start_positions', 'end_positions'],
    num_rows: 3074
})

 Tokenized Validation Set:
Dataset({
    features: ['input_ids', 'token_type_ids', 'attention_mask',
               'start_positions', 'end_positions'],
    num_rows: 520
})

 Dataset Expansion:
   Original train: 3000 examples
   Tokenized train: 3074 features
   Expansion ratio: 1.02x
```

5.2 Feature Expansion Analysis

Key Insights

Why Feature Count Increased:
Original Training: 3,000 examples
Tokenized Training: 3,074 features (1.02× expansion)
Explanation:

- **Stride Windowing:** Long contexts (¿256 tokens after question) split into multiple windows
- **Overlap:** 128-token stride creates 33% overlap between windows
- **Result:** 74 examples (2.5%) generated 2 features each
- **Benefit:** Answers in long contexts not truncated, covered by at least one window

Example: Context with 450 tokens → 2 features:

- Feature 1: Tokens 0-384 (question + context_tokens[0:256])
- Feature 2: Tokens 0-384 (question + context_tokens[128:384])
- Overlap: Tokens 128-256 appear in both features

5.3 Output Feature Structure

Feature	Description & Shape
input_ids	Token IDs for BERT. Shape: (3074, 384). Values: 0-30521
token_type_ids	Segment IDs. Shape: (3074, 384). Values: 0 (question), 1 (context)
attention_mask	Attention mask. Shape: (3074, 384). Values: 1 (real token), 0 (padding)
start_positions	Answer start token index. Shape: (3074,). Values: 0-383
end_positions	Answer end token index. Shape: (3074,). Values: 0-383

Table 2: Tokenized Dataset Feature Structure

6 Answer Position Validation

6.1 Why Validation is Critical

Critical Warning

The Danger of Skipping Validation:
If answer position mapping contains errors:

- **Silent Failure:** Training runs without errors but learns garbage
- **Low Performance:** Model predicts random spans (F1 score ¡ 20%)
- **Wasted Resources:** Hours of GPU time training on corrupted labels
- **Debugging Nightmare:** Hard to trace back to preprocessing bug

Solution: Validate that token positions decode to correct answers **before training**

6.2 Validation Function Logic

Code Explanation

Function: `validate_answer_mapping(tokenized_dataset, original_dataset, num_samples=15)`
Algorithm:

1. For each sample (15 random examples):
   - Get `input_ids`, `start_positions`, `end_positions`
   - Skip if answer truncated (positions at [CLS] token 0)
   - Extract predicted answer: `tokenizer.decode(input_ids[start:end+1])`
   - Get original answer from dataset
   - Compare (case-insensitive, handle WordPiece ##)
2. Track pass/fail counts
3. Print summary with success rate
4. **Block training** if any failures detected

6.3 Validation Results

Code Output

Validation Output (Sample):

```
 VALIDATING ANSWER POSITION MAPPING...
======================================================================
Testing 15 random samples...

Example 1:  PASS
  Original:  'the Main Building'
  Predicted: 'the main building'

Example 2:  PASS
  Original:  'a Marian place of prayer and reflection'
  Predicted: 'a marian place of prayer and reflection'

Example 3:  PASS
  Original:  'a golden statue of the Virgin Mary'
  Predicted: 'a golden statue of the virgin mary'

Example 4:  PASS
  Original:  'September 1876'
  Predicted: 'september 1876'

[...13 more examples...]

======================================================================
 VALIDATION SUMMARY
======================================================================
 Passed: 15
 Failed: 0
 Success Rate: 100.0%

 ALL VALIDATIONS PASSED!
 Safe to proceed to model training!
```

Key Insights

Validation Success Criteria:
100% Success Rate Required:

- All 15 samples must decode to correct answers
- Case differences ignored (bert-base-uncased lowercases everything)
- WordPiece tokens (e.g., "##ing") automatically handled by decoder

Common Acceptable Variations:

- "September 1876" → "september 1876" (case difference)
- "The Observer" → "the observer" (case difference)
- Original may have extra whitespace (stripped in comparison)

Failure Indicators:

- Predicted answer completely different from original
- Predicted answer is partial span (missing words)
- Predicted answer includes extra context beyond answer

7 Question 2.4: PyTorch Tensor Conversion

7.1 Question Statement

Question 2.4 (5 marks): Is manual PyTorch tensor conversion necessary before passing datasets to the Trainer?

7.2 Answer & Explanation

Code Explanation

Answer: NO, manual tensor conversion is NOT required.
Explanation:
The HuggingFace `Trainer` API automatically handles tensor conversion internally. When tokenized datasets are passed to `Trainer`, the following happens automatically:
1. Dynamic Batching:

- `DataCollator` handles creating batches on-the-fly
- Selects batch_size examples per iteration
- Stacks them into batch tensors

2. Automatic Conversion:

- HuggingFace Datasets are Arrow-backed (memory-mapped)
- `Trainer` converts to PyTorch tensors automatically during batch creation
- Conversion happens per-batch, not entire dataset upfront

3. Memory Efficiency:

- Only converts what's needed per batch (16-32 examples)
- Avoids loading entire dataset into GPU memory
- Enables training on datasets larger than available RAM

When Manual Conversion IS Required:

- Using custom PyTorch `DataLoader` directly (not `Trainer`)
- Implementing custom training loops
- Working outside HuggingFace ecosystem (raw PyTorch)

Conclusion: For our project using `Trainer`, manual conversion is unnecessary and would add no benefit.

7.3 Verification of Readiness

Code Output

Dataset Ready for Training:

```
Training features: 3074
Validation features: 520
Feature keys: ['input_ids', 'token_type_ids', 'attention_mask',
                  'start_positions', 'end_positions']
```

Key Insights

Training Readiness Checklist:
Tokenizer loaded (bert-base-uncased)
Datasets tokenized (3074 train, 520 validation)
Answer positions mapped (start/end token indices)
Validation passed (100% success rate)
All required features present:

- input_ids (token IDs)
- token_type_ids (segment IDs)
- attention_mask (real vs padding)
- start_positions (answer start)
- end_positions (answer end)

Ready to proceed to Notebook 3: Model Training

8 Conclusion

8.1 Notebook 2 Summary

Key Insights

Key Accomplishments:

Question 2.1 - Tokenizer Loading:

- Successfully loaded BertTokenizerFast (bert-base-uncased)
- Verified vocabulary size (30,522), max length (512), special tokens

Question 2.2 - Preprocessing Function:

- Created robust preprocessing function with answer position mapping
- Implemented stride windowing for long contexts (128-token overlap)
- Handled truncated answers (set positions to 0)
- Used offset_mapping to convert character positions → token positions

Question 2.3 - Dataset Application:

- Applied preprocessing to 3,000 training + 500 validation examples
- Generated 3,074 training features (1.02× expansion due to stride)
- Processing time: 4 seconds total (fast on GPU)

Question 2.4 - PyTorch Tensor Explanation:

- Explained that manual tensor conversion is unnecessary with Trainer API
- Described automatic batching and conversion mechanisms

Bonus - Answer Validation:

- Validated 15 samples with 100% success rate
- Confirmed answer positions decode to correct text
- Prevented training on corrupted labels

8.2 Critical Takeaways

Code Explanation

Most Important Lessons:

1. Offset Mapping is Essential:

- `return_offsets_mapping=True` is NOT optional for QA tasks
- Without it, impossible to map character-based answers to token positions
- Always validate mappings before training

2. Stride Prevents Answer Loss:

- Stride creates overlapping windows for long contexts
- Ensures answers don't get truncated at window boundaries
- Small expansion ($1.02\times$) is acceptable trade-off

3. Validation Catches Silent Bugs:

- Preprocessing bugs don't cause errors—they cause bad training
- Always validate on 10-20 samples before full training
- 100% validation success rate is mandatory

4. HuggingFace Abstractions Save Time:

- Trainer API handles tensor conversion, batching, device placement
- No need to write custom PyTorch DataLoader code
- Focus on model architecture and hyperparameters instead

8.3 Next Steps

Notebook 3 Preview: Model Training & Evaluation

1. Load pretrained BERT model for QA (`BertForQuestionAnswering`)
2. Configure training arguments (batch size, learning rate, epochs)
3. Initialize `Trainer` with model and tokenized datasets
4. Train model on 3,074 training features
5. Evaluate on 520 validation features
6. Calculate metrics (Exact Match, F1 score)
7. Save fine-tuned model for deployment

End of Notebook 2 Documentation

Preprocessing & Tokenization Complete
Ready for Model Training