NLP Milestone1

Sohaila Ibrahim
ID: 52-21225

Nada Elbehery
ID: 52-8973

April 22, 2025

Contents

1 Dataset

1.1 Why chose SQuAD ?

The Stanford Question Answering Dataset (SQuAD) is widely used for extractive Question Answering tasks. It is an excellent choice for the following reasons:

- **Real-world Structure**: It contains over 100,000 questions based on Wikipedia articles, making it both diverse and grounded in real-world knowledge.
- **Character-level Answers**: Answers are specified as spans in the context, allowing fine-grained supervision and making it ideal for span-based models.
- **Variety of Context Lengths and Question Styles**: The dataset features a broad range of paragraph and question lengths, supporting generalization to various inputs.
- **Well-suited for Extractive Models**: The SQuAD dataset is inherently designed for extractive question answering, where the goal is to predict an answer span directly from the context. which leverage contextual embeddings to identify the most probable start and end tokens within the passage.

1.2 Loading and Processing the Dataset

We define a function `load_squad_subset` to extract and filter samples from the dataset. It performs the following:

- Filters out by choosing contexts that do not exceed 200 length.
- Tokenizes the context and questions using BERT tokenizer.
- Aligns character-based answer indices to token-based indices using offset mappings.
- Verifies the validity of each token mapping.
- Prepare the dataset for training.

1.3 Character to Token Mapping using BERT Tokenizer

The BERT tokenizer (e.g., `bert-base-cased`) is used with `return_offsets_mapping=True` to map tokens back to the original character positions.

1.3.1 How Offset Mapping Works

Given a context, the BERT tokenizer returns:

- `tokens()`: Tokenized subwords (e.g., "unbelievable" → "un", "##believable").
- `offset_mapping`: A list of tuples with (start_char, end_char) for each token.

This makes it possible to:

1. Convert character-based answer spans to token indices for model training.
2. Convert predicted token spans back to character indices for evaluation.

1.3.2 Aligning Character to Token Indices

The function `align_character_to_token_indices(context, answer_start_char, answer_text)`:

- Uses `offset_mapping` to find which tokens overlap with the answer span.
- Returns token indices for the start and end of the answer.

1.3.3 Verifying Token Mapping

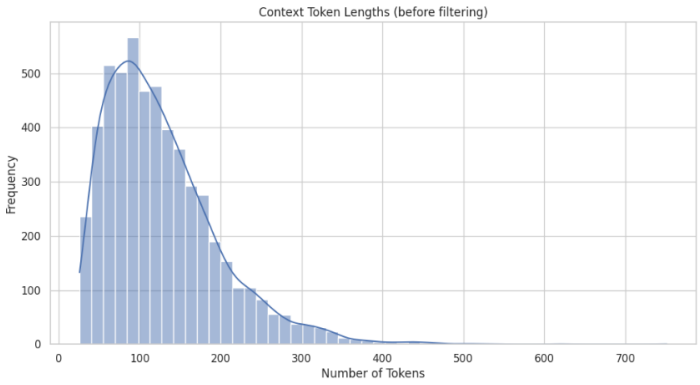
The function `verify_token_mapping()`:

- Compares the original answer text to the substring derived from the predicted token span.
- Normalizes text by lowercasing and removing extra whitespace.

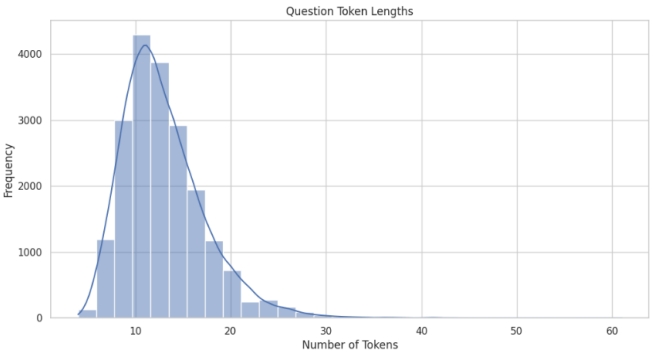
1.4 Dataset Analysis and Insights

The function `plot_dataset_insights()` visualizes key statistics:

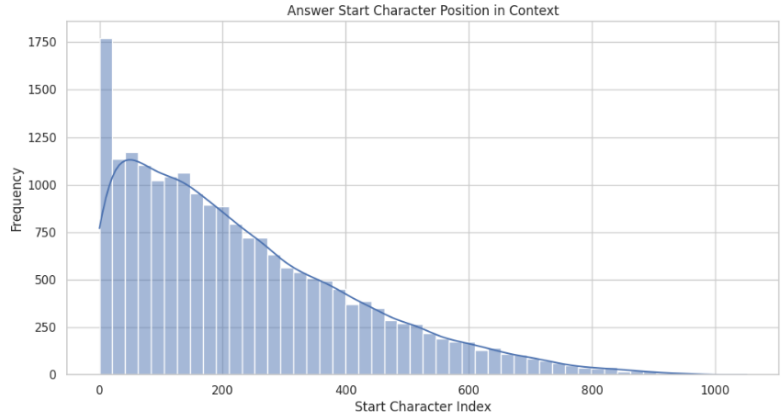
1. **Context Token Lengths:** Distribution before filtering.
2. **Question Token Lengths:** Typically shorter than contexts.
3. **Answer Start Positions:** Location of answers in context.
4. **Answer Lengths:** In characters.
5. **Token Mapping Validity:** Pie chart of valid vs. invalid mappings.



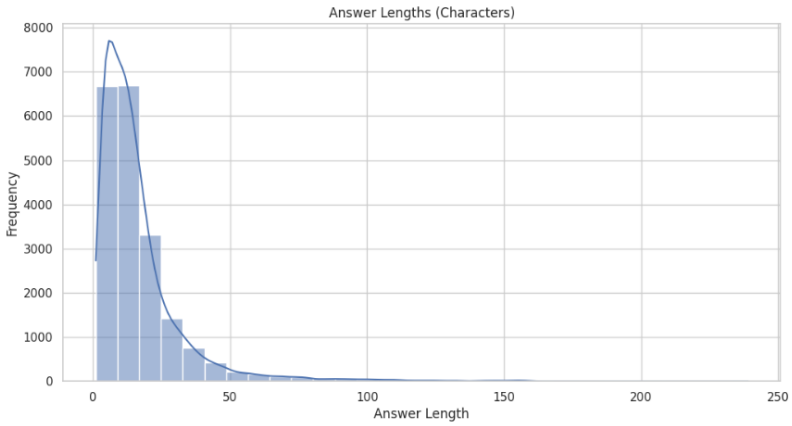
Distribution of Context Token Lengths



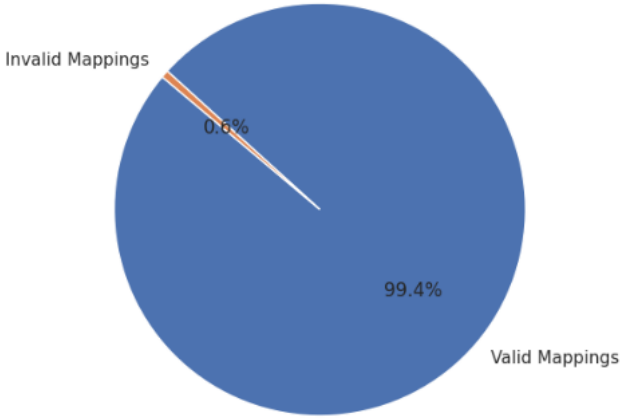
Distribution of Question Token Lengths



Answer Start Character Position



Answer Length in Characters



Percentage of Valid vs. Invalid Token Mappings

2 Data Preprocessing

2.1 Vocabulary Construction

To enable deep learning models to operate on textual data, the first step is to convert tokens (words or subwords) into unique integer indices. This process is known as vocabulary building.

- **Tokenization**: For each SQuAD sample, both the context and the question are tokenized using a subword tokenizer (e.g., BERT tokenizer).
- **Frequency Filtering**: Tokens are counted, and only those that appear at least `min_freq` times are included in the vocabulary.

- **Special Tokens**: Reserved tokens like `[PAD]`, `[UNK]`, `[SEP]`, and `[CLS]` are added for padding, unknown words, and sequence delimiting.
  - `[PAD]`: Used to pad sequences to the same length for batch processing.
  - `[UNK]`: Represents unknown or out-of-vocabulary tokens.
  - `[SEP]`: Separates segments like question and context in the input.
  - `[CLS]`: Placed at the beginning and used to represent the whole input sequence.

2.2 Word Embedding Initialization with GloVe

Pre-trained embeddings like GloVe (Global Vectors for Word Representation) are used to initialize the embedding layer with semantically meaningful vectors:

- **Loading GloVe**: The GloVe file is read line by line, mapping each word to a dense 300-dimensional vector.
- **Mapping to Vocabulary**: Each token in the vocabulary is assigned the corresponding GloVe vector if available.
- **Handling Subword Tokens**: For BERT-style subword tokens (e.g., `ing`), the model attempts to fall back to the base word (e.g., `ing`) if the full subword isn't found in GloVe.
- **Random Initialization**: If neither the token nor its base form is found, the vector is initialized randomly from a normal distribution to maintain variance.

2.3 Dataset Structuring for Extractive QA

To train the extractive model, a custom PyTorch dataset class is constructed:

- **Input Encoding**: Each sample's question and context are tokenized and converted into index sequences using the built vocabulary.
- **Input Concatenation**: The final input sequence follows the format:

  `[CLS]` `question_tokens` `[SEP]` `context_tokens`,

  which aligns with how BERT-based models expect input.
- **Padding**: The sequence is padded or truncated to a fixed maximum length (`max_len`) to allow batch processing.
- **Label Alignment**: Start and end indices of the answer span (originally found within the context) are shifted to account for the `[CLS]`, question, and `[SEP]` tokens prepended to the sequence.
- **Output Format**: The dataset returns a tuple of:

- The padded input sequence (as a tensor of token indices),
- The start index of the answer span,
- The end index of the answer span.

2.4 Data Splitting and Loading with DataLoader

To prepare the dataset for training and evaluation, the following steps are performed:

- **Splitting the Data**: The dataset is split into training and testing subsets. Using `train_test_split` from scikit-learn, 80% of the data is used for training and 20% for testing. This is achieved with the following code:

  ```
  train_samples, test_samples = train_test_split(samples, test_size=0.2, random_state=42)
  ```

- **Creating Custom Datasets**: The training and testing samples are wrapped in custom `SquadExtractiveDataset` classes. This ensures the data is processed and tokenized consistently using the built vocabulary:

  ```
  train_dataset = SquadExtractiveDataset(train_samples, vocab)

  test_dataset = SquadExtractiveDataset(test_samples, vocab)
  ```

- **DataLoader**: The datasets are then loaded into PyTorch's `DataLoader` for batch processing. The `train_loader` shuffles the training data for random sampling, while the `test_loader` processes the test data without shuffling:

  ```
  train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

      test_loader = DataLoader(test_dataset, batch_size=64)
  ```

  The `DataLoader` objects handle batching and data loading in parallel, facilitating efficient model training and evaluation.

3 Loss Function and Evaluation Metrics

3.1 Loss Function

The model predicts the start and end positions of an answer span within a context. For each example, the predicted start and end logits are compared to the ground-truth indices using the cross-entropy loss.

Let:

- $s_{\text{logits}} \in \mathbb{R}^L$ be the predicted start logits.
- $e_{\text{logits}} \in \mathbb{R}^L$ be the predicted end logits.

- $s_{\text{true}} \in \{0, 1, ..., L-1\}$ be the true start index.
- $e_{\text{true}} \in \{0, 1, ..., L-1\}$ be the true end index.
- L be the length of the input sequence.

The total loss is the sum of the cross-entropy losses for start and end positions:

$$\mathcal{L} = \text{CE}(s_{\text{logits}}, s_{\text{true}}) + \text{CE}(e_{\text{logits}}, e_{\text{true}})$$

where CE denotes the standard cross-entropy loss:

$$\text{CE}(z, y) = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

3.2 Evaluation Metrics

3.2.1 Exact Match (EM)

This metric checks whether the predicted span exactly matches the ground truth span:

$$\text{EM} = \begin{cases} 1, & \text{if } \hat{s} = s_{\text{true}} \text{ and } \hat{e} = e_{\text{true}} \\ 0, & \text{otherwise} \end{cases}$$

where $\hat{s}$ and $\hat{e}$ are the predicted start and end indices.

3.2.2 Token-level F1 Score

To compute the F1 score at the token level, we treat the predicted and true spans as sets of token indices:

$$\text{Predicted span} = \{\hat{s}, \hat{s}+1, \ldots, \hat{e}\} \text{True span} = \{s_{\text{true}}, s_{\text{true}}+1, \ldots, e_{\text{true}}\}$$

Let P be the set of predicted tokens, and T be the set of true tokens. Then:

$$\text{Precision} = \frac{|P \cap T|}{|P|}, \quad \text{Recall} = \frac{|P \cap T|}{|T|}$$

The token-level F1 score is the harmonic mean of precision and recall:

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

If there is no overlap ($P \cap T = \emptyset$), or if either span is empty, F1 is defined to be zero.

4 Model Architecture

4.1 Architecture

The model used for training is an extractive Question Answering system based on a bidirectional GRU network. It consists of the following components:

- **Embedding Layer:** Converts input word indices into dense vectors using pre-trained GloVe embeddings. This layer captures semantic information of the words.
- **Dropout Layer:** Applies dropout to the embeddings to reduce overfitting by randomly setting a fraction of the input units to zero during training.
- **Bidirectional GRU Encoder:** A 3-layer bidirectional Gated Recurrent Unit (GRU) processes the embedded input. The bidirectional nature allows the model to capture context from both past and future tokens in the sequence.
- **Prediction Layers:** Two linear layers predict the start and end positions of the answer span in the input context. Each position is modeled as a separate classification task.

4.2 Model Hyperparameters

The extractive Question Answering model is configured with the following key hyperparameters:

- **vocab_size:** Set to `len(vocab)`, this defines the total number of unique tokens in the vocabulary. It determines the input dimension for the embedding layer, where each token is mapped to a specific embedding vector.
- **emb_dim:** The embedding dimension is set to 300, which corresponds to the size of the GloVe word vectors. Each word in the vocabulary is represented by a 300-dimensional dense vector that captures semantic meaning.
- **hidden_dim:** Set to 512, this defines the number of hidden units in each direction of the bidirectional GRU. A higher hidden dimension allows the network to model more complex patterns and dependencies in the input sequences.
- **embedding_matrix:** This matrix contains pre-trained GloVe embeddings and is used to initialize the weights of the embedding layer. It is moved to the appropriate computation device (e.g., GPU) using `.to(device)` to ensure efficient processing.
- **dropout:** A dropout rate of 0.7 is applied to the embeddings. This means that during training, 70% of the units are randomly set to zero, which helps prevent overfitting by encouraging the model to be more robust.

4.3 Training Procedure

During training, the model processes batches of 64 samples from the DataLoader. For each batch:

- Predictions for start and end logits are computed.
- Cross-entropy losses for both logits are calculated and averaged.
- Gradients are computed using backpropagation, and the optimizer updates the model parameters.

4.4 Evaluation Process

The evaluation function operates in a similar manner to the training loop but does not update model parameters. It uses `torch.no_grad()` to disable gradient calculations, making the process more efficient.

Each batch is processed as follows:

- Context tokens are passed through the model to obtain start and end logits.
- Cross-entropy loss is calculated for both logits to evaluate how well the model generalizes to unseen data.
- Start and end indices are predicted by selecting the positions with the maximum logits (using `argmax`).
- Metrics such as loss, exact match (EM), and F1 score are computed and averaged over the dataset.

5 Inference Pipeline for Extractive QA

5.1 Input Tokenization

Given:

- A question q
- A context passage c

We tokenize the input as:

$$\text{tokens} = [\texttt{[CLS]}] + \text{tokenize}(q) + [\texttt{[SEP]}] + \text{tokenize}(c)$$

Each token is mapped to an ID using a vocabulary:

$$\text{token\_ids} = [\text{vocab[tok] for tok in tokens}]$$

The sequence is padded/truncated to a maximum length L (e.g., 300) and converted into a tensor:

$$\text{input} \in \mathbb{Z}^{1L}$$

2. Model Forward Pass

The model outputs start and end logits:

$$s_{\text{logits}}, e_{\text{logits}} \in \mathbb{R}^L$$

These logits are sliced to extract scores corresponding to context tokens only. If the context begins at position p in the token list:

$$s_{\text{context}} = s_{\text{logits}}[p : p + |c|], \quad e_{\text{context}} = e_{\text{logits}}[p : p + |c|]$$

5.2 Span Prediction with Heuristics

To find the best span (i, j) such that $i \leq j$, we maximize a scoring function over all valid spans of length ≤ 15:

$$\text{score}(i, j) = \text{softmax}(s_{\text{context}})_i + \text{softmax}(e_{\text{context}})_j + \text{HeuristicBonus}(i, j)$$

Heuristic Adjustments

Based on the question type (e.g., *who, when, how many*), additional scores are added if the span satisfies specific conditions using spaCy entity recognition:

- **How much / How many:** boost if span contains numbers, money, or percent entities.

- **Who:** boost if span contains a `PERSON` entity.
- **When:** boost if span contains a `DATE` or `TIME` entity.
- **What happened / What did:** boost if the span contains verbs and is clause-like.
- **General NER boost:** small boost if span contains `ORG`, `GPE`, or `LOC` entities.

5.3 Extracting Final Answer

Once the best token span (i^*, j^*) is selected, we use BERT-style offset mappings to retrieve the character-level answer from the context:

$$\text{char\_start} = \text{offset\_mapping}[i^*][0], \quad \text{char\_end} = \text{offset\_mapping}[j^*][1]$$

The predicted answer is:

$$\text{Answer} = c[\text{char\_start} : \text{char\_end}]$$

6 Results

7 Model Performance Analysis

7.1 Training and Evaluation Results

Figure 1 shows the training and validation curves for loss, Exact Match (EM), and F1 score across 20 epochs. The model shows a consistent decrease in training loss, along with a steady improvement in both training EM and F1 scores, indicating effective learning on the training data.

The validation loss initially decreases but begins to plateau and slightly increase after epoch 13, suggesting the onset of overfitting. Despite this, the validation EM and F1 metrics show general upward trends until around epoch 15, after which the improvements become marginal or unstable.

The best model was saved at **epoch 15**, based on the peak validation F1 score:

- **Training Loss:** 4.5018 — **Validation Loss:** 5.6138
- **Training EM:** 0.2541 — **Validation EM:** 0.1777
- **Training F1:** 0.3431 — **Validation F1:** 0.2430

The training metrics continue to improve beyond epoch 15, but the validation metrics do not follow this trend, confirming that epoch 16 represents the optimal point before the model begins to overfit. This balance between performance and generalization justifies it being selected as the best checkpoint.

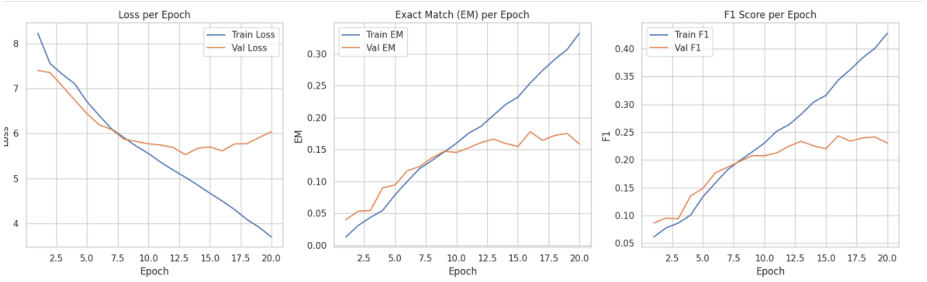


Figure 1: Training and validation metrics across 20 epochs.

7.2 Answer Prediction Analysis

- **Performance Patterns**:
  - Strong on simple factoid questions (dates, names)
  - Struggles with longer named entity spans

  - Partial matches common for correct entities with missing context

- **Answer Length**:
  - Tendency to predict shorter answers than ground truth
  - Focuses on salient parts but misses details

- **Entity Recognition**:
  - Confuses similar entity types (team vs league names)
  - Weakness in distinguishing related entities

- **Question Type Performance**:
  - High accuracy for "When" and "How many" questions
  - Lower performance for descriptive answers

Error Analysis

- **Main Error Types**:
  - Entity Confusion
  - Incomplete Spans
  - Excessive Spans
  - Completely Wrong predictions

- **Contributing Factors**:
  - Context Ambiguity
  - Answer Length
  - Entity Proximity
  - Question Complexity

- **Error Patterns by Question Type**:
  - "Who" questions: Entity confusion
  - "What" questions: Incomplete spans
  - Specialized knowledge questions: Higher error rates

Prediction Quality Breakdown

- **Correct or Near-Correct Predictions**:
  - Strong on fact-based questions (names, dates, places)
  - Handles semantically similar phrases well
  - Minor differences like missing articles still reflect comprehension
- **Partially Correct Predictions**:
  - Predicts part of correct answer or includes extra context
  - Common for longer/descriptive answers
  - Often due to misaligned start/end token indices
- **Incorrect or Unrelated Predictions**:
  - Generic, irrelevant, or incomplete answers
  - Suggests low confidence or weak context understanding
  - Common for ambiguous or multi-hop questions

Model Behavior Summary

- **Strengths**:
  - Handles short, factual answers well
  - Recognizes named entities in clear contexts
- **Weaknesses**:
  - Sensitive to boundary errors
  - Struggles with complex answers
  - Occasionally produces unrelated outputs

8 Limitations

8.1 Architectural Constraints

- Sequential Processing Limitation
  - **Core Issue**: Processes information strictly in order (word-by-word)
  - **Why It Matters**: Later words can't directly influence early interpretations
  - **Real-World Effect**: Misinterprets sentences where key context appears late
  - **Example**: For "The award was won by Company A, not Company B", it might fixate on "Company B" before seeing the negation
- Fixed Context Window
  - **Core Issue**: Can only consider a limited amount of text at once **Why It Matters**: Loses track of earlier references in long documents **Real-World Effect**: Answers may ignore crucial earlier context **Example**: In a 10-page document, might miss that "the defendant" refers to a person introduced on page 1

8.2 Knowledge Limitations

- Domain Specialization Gap
  - **Core Issue**: Primarily trained on general-purpose texts **Why It Matters**: Lacks technical vocabulary understanding **Real-World Effect**: Struggles with field-specific terminology **Example**: Might confuse medical terms like "angiogenesis" with similar-sounding common words
- Static Knowledge Base
  - **Core Issue**: Knowledge cutoff at training date **Why It Matters**: Can't incorporate new information **Real-World Effect**: Gives outdated answers about evolving topics **Example**: Wouldn't know about recent scientific discoveries or current events

8.3 Interaction Challenges

- Question-Type Bias
  - **Core Issue**: Handles different question formats unevenly **Why It Matters**: Performance varies unpredictably **Real-World Effect**: Simple "who/when" questions work better than complex "why/how" **Example**: Easily answers "When was X born?" but struggles with "Why did X occur?"

- Precise Span Identification
  - **Core Issue**: Difficulty pinpointing exact answer boundaries **Why It Matters**: Extracts incomplete or overlong answers **Real-World Effect**: Gets the "gist" right but not the exact wording **Example**: For "Where was the treaty signed?", might extract "The treaty was signed in Paris" instead of just "Paris"

8.4 Systemic Limitations

Error Amplification

- **Core Issue**: Early mistakes affect later processing **Why It Matters**: Small errors lead to completely wrong answers **Real-World Effect**: Produces confidently wrong responses **Example**: Misreading a name might generate detailed but false biographical information

9 Improvements

Top Architectural Improvements

- *Transformer Architecture*
  - *Replace GRU layers with transformer blocks*
  - *Benefits: Better long-range dependency modeling*
  - *Targets: Entity confusion errors in predictions*
- *Contextual Embeddings*
  - *Upgrade from static GloVe to BERT/RoBERTa embeddings*
  - *Benefits: Dynamic word representations*
  - *Impact: Significant QA performance gains*

Most Impactful Training Refinements

- *Focal Loss Implementation*
  - *Replace standard cross-entropy loss*
  - *Benefits: Focuses on difficult examples*
  - *Targets: Complex questions and entity differentiation*
- *Learning Rate Scheduling*
  - *Add cosine annealing with warm restarts*
  - *Benefits: Escapes optimization plateaus*
  - *Evidence: Validation metrics stalling mid-training*

Critical Data Processing Improvements

- *Question Type Classification*
  - *Add question type as auxiliary feature*
  - *Benefits: Better "who"/"what" distinction*
  - *Targets: Entity confusion patterns*
- *Boundary Enhancement*
  - *Improve span detection mechanisms*
  - *Benefits: Reduces incomplete spans*
  - *Evidence: Most frequent error category*

Essential Post-Processing Techniques

- *Entity-Aware Decoding*
  - *Incorporate NER for boundary detection*
  - *Benefits: More precise entity extraction*
  - *Targets: Incomplete span predictions*
- *Answer Verification Module*
  - *Add candidate span scoring*
  - *Benefits: Filters incorrect answers*
  - *Impact: Reduces completely wrong outputs*