COMP 316 GROUP PROJECT - QUESTION ANSWERING SYSTEM FOR WIKIPEDIA ARTICLES.

Group members

1. Ziauddeen Mohamad (223084149)

2. Ethan Govender (223016046)

3. Amahle Ngcobo (223000003)

4. Zaakirah Shaik (222025988)

Introduction

The advancement of Natural Language Processing has paved the way for machines to understand and answer human language with striking accuracy. One of the most impactful applications of NLP is Question answering, which is the task of extracting accurate answers from unstructured text, given a very specific query.

In this project, we aimed to build and fine-tune a QA system using transformer-based models that are trained on the Stanford Question Answering Dataset (SQUAD).

Our strategy involved iteratively training a series of models to evaluate how different configurations including batch size, training epochs, and dataset segmentation affect the performance.

The goal of this staged approach was mainly to develop a more in-depth understanding of how training parameters influence QA model performance while also optimizing resource usage and training time.

In this report we explain our methodology, implementation, experimental design, the training process, and finally, the results.

System Architecture

We have four major components in the system architecture of our question answering system

Keyword Extraction: The KeyBERT module extracts the relevant keywords from the question. This step narrows down the search space of candidate documents. **Document Retrieval:** Sentence Transformer-based Retriever uses the keywords to retrieve the most appropriate Wikipedia articles. Semantic similarity is employed for article ranking and selecting top candidates. **Answer Extraction:** The DistilBERT model with QA fine-tuned on SQuAD dataset takes the question and the article content retrieved as input and outputs the most likely answer span in the text. **Final Answer:** The answer span extracted is provided by the system as the response to the user's question.

This modular design allows questions to be processed effectively and accurate responses to be reaped from a huge corpus of text.

Methodology

Our approach was to create an extractive question answering model using the SQUAD dataset and the Hugging Face Transformers library. We sought to identify answer spans in the given context passages for questions. We

used the DistilBERT model due to its performance-efficiency trade-off. The processes included data preprocessing, tokenization, training, evaluation, and model saving based on performance and training step thresholds.

A. Data Preprocessing

The Wikipedia articles were preprocessed to support long contexts, which are longer than the maximum input size of the DistilBERT model. The articles were divided into smaller, overlapping chunks using the chunk_text_with_stride function. The function divides the text into fixed-size portions, with an overlap specified in order to keep context between the chunks. The articles are chunked specifically by tokens, with a sliding window approach. This saves no information at the chunk edges. The data were processed in SQuAD format, i.e., constructing a list of dictionaries, each consisting of the question, context (a piece of text), and answer (start and end character indices in the context). There was no additional cleaning or normalization since the DistilBERT pre-trained model is insensitive to modifications to the input text.

B. Chunking and Ranking

Chunking is a crucial preprocessing step for question answering on large documents. It allows the QA model to handle smaller and manageable chunks of text.

Chunking techniques:

These two functions display two different strategies for splitting long text into chunks. The chunk_text_by_tokens functions divide the text based on the fixed token limit, which creates non overlapping chunks that ensure each one remains within the maximum input size.

The chunk_text_by_stride function creates chunks which overlap with the previous one by a specified number of tokens.

```python
def chunk_text_by_tokens(text, max_tokens=450):
    """Split long text into chunks based on token length."""
    tokens = tokenizer.encode(text, add_special_tokens=False)
    chunks = [tokens[i:i+max_tokens] for i in range(0, len(tokens), max_tokens)]
    return [tokenizer.decode(chunk, skip_special_tokens=True) for chunk in chunks]

def chunk_text_with_stride(text, max_length=384, stride=128):
    inputs = tokenizer(text, return_overflowing_tokens=True, max_length=max_length, stride=stride, truncation=True, padding="max_length")
    return [tokenizer.decode(input_ids, skip_special_tokens=True) for input_ids in inputs["input_ids"]]
```

We employ a fixed window length and a stride to generate overlapping chunks. Overlapping chunks help avoid splitting the answer into two chunks

Ranking techniques used:

Our chunk selection process involved evaluating two ranking strategies: TF-IDF and semantic ranking. TF-IDF ranking calculates chunk relevance based on word importance within the document set. Semantic ranking, on the other hand, leverages Sentence Transformers to measure the semantic similarity between the query and the chunks. We decided to proceed with semantic ranking because it prioritizes understanding the underlying meaning, offering a more nuanced approach than TF-IDF's reliance on exact word occurrences.

```python
def rank_chunks_semantic(question, chunks, top_k=3):
    question_emb = embedder.encode(question, convert_to_tensor=True)
    chunk_embs = embedder.encode(chunks, convert_to_tensor=True)
    scores = util.cos_sim(question_emb, chunk_embs)[0]
    top_indices = scores.argsort(descending=True)[:top_k]
    return [chunks[i] for i in top_indices]

def rank_chunks_tfidf(question, chunks, top_k=3):
    """Rank text chunks based on TF-IDF similarity to the question."""
    vectorizer = TfidfVectorizer().fit([question] + chunks)
    vectors = vectorizer.transform([question] + chunks)
    scores = cosine_similarity(vectors[0:1], vectors[1:]).flatten()
    top_indices = scores.argsort()[-top_k:][::-1]
    return [chunks[i] for i in top_indices]
```

C. Answer Extraction

Answer extraction is carried out by using a fine-tuned DistilBERT model. DistilBERT is a BERT distillation, which is a strong transformer-based model. We fine-tuned DistilBERT on the SQuAD question-answer dataset within the passage context. The model, when fine-tuned, takes a passage context and a question as input and attempts to predict the answer's start position and end position within the passage. The model predicts a distribution of probabilities over the passage tokens, and it selects the most likely span as the answer. The answer_question function carries out this process.

Preprocessing and Feature Preparation

Data preprocessing is an important and critical step in the training process. Raw text data has to be converted into a format that the model can understand.

This code below defines the prepare_train_features function. This funtion processes the training data for question answering tasks. It tokenizes the question and context from each example, ensuring the sequence is properly truncated, padded, and split when necessary.

The function also calculates the start and end positions of the answer within the context. If an answer is not found, it assigns the start and end positions to the cls_token_id, indicating that there is no answer. The function also handles token overflow. It does this by mapping tokens back to the corresponding sample and then adjusting for potential overlap between the context and answer span. The resulting tokenized examples are returned with start and end positions for each answer.

```python
def prepare_train_features(examples):
    tokenized_examples = tokenizer(
        examples["question"],
        examples["context"],
        truncation="only_second",
        max_length=384,
        stride=128,
        return_overflowing_tokens=True,
        return_offsets_mapping=True,
        padding="max_length",
    )

    sample_mapping = tokenized_examples.pop("overflow_to_sample_mapping")
    offset_mapping = tokenized_examples.pop("offset_mapping")

    start_positions = []
    end_positions = []

    for i, offsets in enumerate(offset_mapping):
        input_ids = tokenized_examples["input_ids"][i]
        cls_index = input_ids.index(tokenizer.cls_token_id)

        sequence_ids = tokenized_examples.sequence_ids(i)
        sample_index = sample_mapping[i]
        answers = examples["answers"][sample_index]
        if len(answers["answer_start"]) == 0:
            start_positions.append(cls_index)
            end_positions.append(cls_index)
        else:
            start_char = answers["answer_start"][0]
            end_char = start_char + len(answers["text"][0])

            token_start_index = 0
            while sequence_ids[token_start_index] != 1:
                token_start_index += 1

            token_end_index = len(input_ids) - 1
            while sequence_ids[token_end_index] != 1:
                token_end_index -= 1

            if not (offsets[token_start_index][0] <= start_char and offsets[token_end_index][1] >= end_char):
                start_positions.append(cls_index)
                end_positions.append(cls_index)
            else:
                while token_start_index < len(offsets) and offsets[token_start_index][0] <= start_char:
                    token_start_index += 1
                start_positions.append(token_start_index - 1)

                while offsets[token_end_index][1] >= end_char:
                    token_end_index -= 1
                end_positions.append(token_end_index + 1)

    tokenized_examples["start_positions"] = start_positions
    tokenized_examples["end_positions"] = end_positions
    return tokenized_examples
```

Implementation

We implemented this project in Python using the Hugging Face Transformers library. We designed a script that can progressively adjust the batch sizes, load checkpoints, and also control training iterations and evaluation. Each model is trained with fluctuating and distinct parameters or data segments to explore their effects on the performance.

Experimental Design

We trained 20 models in total. Models 1 to 9 were trained on the entire SQUAD dataset for one epoch each, with a consistent batch size of 8, saving checkpoints every 2000 steps. No data shuffling or subset selection was applied. These models served to test how different seeds and training instances perform on full data.

For models 10 to 20, the training setup was modified to use only 1000 randomly selected SQUAD examples for each model. Model 10 continued from model 5, which was our best performing model, also using a batch size of 8. While model 11 increased the batch size to 16 to examine effects on training stability and performance. Later models adjust batch size based on these end results.

Training Process

Training was done using GPU acceleration on Google Colab. Each model was trained using the AdamW optimizer, batch size 16, 3 epochs and a learning rate of 5e-5.

The training loop logged evaluation metrics like exact match (EM) and F1 scores using the SQUAD metric computation provided by Hugging Face. These scores helped contrast the effectiveness of several configurations like dataset size, batch size and number of epochs.

The DistilBERT model was fine-tuned on the SQuAD dataset using the AdamW optimizer. AdamW is a variant of the Adam optimizer that incorporates weight decay, which helps to improve generalization. The following hyperparameters were used:

The model was trained on a Tesla T4 GPU provided by Google Colab. Early stopping was used to prevent overfitting. The training process took approximately 2 hours.

```
for epoch in range(3):
    small_train = squad["train"].shuffle(seed=seed1).select(range(1000))

# Tokenize the reduced dataset
    tokenized_squad = small_train.map(
        prepare_train_features,
        batched=True,
        remove_columns=small_train.column_names
    )


# DataLoader
    train_loader = DataLoader(tokenized_squad, batch_size=16, shuffle=True, collate_fn=default_data_collator)
    progress = tqdm(train_loader)
    for batch in progress:
        batch_num += 1

        input_ids = batch["input_ids"].to(device)
        attention_mask = batch["attention_mask"].to(device)
        start_positions = batch["start_positions"].to(device)
        end_positions = batch["end_positions"].to(device)

        outputs = model(
            input_ids=input_ids,
            attention_mask=attention_mask,
            start_positions=start_positions,
            end_positions=end_positions,
        )
```

```
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        optimizer.zero_grad()

        totalLoss += loss.item()
        LossVector.append(loss.item())
        avg = totalLoss / batch_num

        progress.set_description(f"Loss: {avg:.4f}")
count += 1
#save_path = f"/content/drive/My Drive/Distil-BERT/QA/Model" + str(count)
save_path = f"/content/drive/My Drive/Colab Notebooks/Distil-BERT/QA/Model" + str(count)                #
model.save_pretrained(save_path)
tokenizer.save_pretrained(save_path)
print(f"Model saved at {save_path}")
print(f"Model {count} Loss Vector: {LossVector}")
```

Evaluation

After training, the models were then evaluated using Exact Match (EM), F1 score, and Accuracy to assess their ability to correctly answer questions on the SQUAD dataset. The metrics were extracted for each model and

plotted over time. This allows us to contrast visually the difference in performance. Each model's performance was determined by an average score calculated from the three metrics.

The evaluation process involved feeding a set of questions and contexts from the SQuAD dataset to the trained model and comparing the model's predictions against the actual answers.

To thoroughly assess the model's performance, we implemented a comparative evaluation across multiple model configurations. This involved training several models with different hyperparameters and settings, including variations in batch size and dataset segmentation. The results of these evaluations were visualized using matplotlib to facilitate a clear comparison of each model's performance across the evaluation metrics.

The evaluation.ipynb script includes functions to calculate these metrics and generate visualizations. The script also incorporates text normalization steps, such as expanding slang and removing unwanted characters, to ensure a fair evaluation.

The comparative analysis enabled us to identify the model configuration that yielded the best overall performance. We also analysed the impact of different training parameters on the evaluation metrics, providing insights into the model's behaviour and generalization capabilities.

The performance of the question answering system was evaluated using three standard metrics:

- **Exact Match (EM):** This metric measures the percentage of predictions that exactly match the ground truth answer. It is a strict metric that requires the predicted answer span to be identical to the correct answer.
- **F1 Score:** This metric measures the overlap between the predicted answer and the ground truth answer. It is the harmonic mean of precision and recall. F1 score is more lenient than EM, as it considers partial overlaps.
- **Accuracy:** The accuracy metric was also calculated.

These metrics provide a comprehensive evaluation of the system's performance. EM gives a sense of how often the system produces the exact correct answer, while F1 score measures how well the system identifies the correct answer span, even if it doesn't match it exactly. Accuracy provides a general measure of the correctness of the model's predictions.

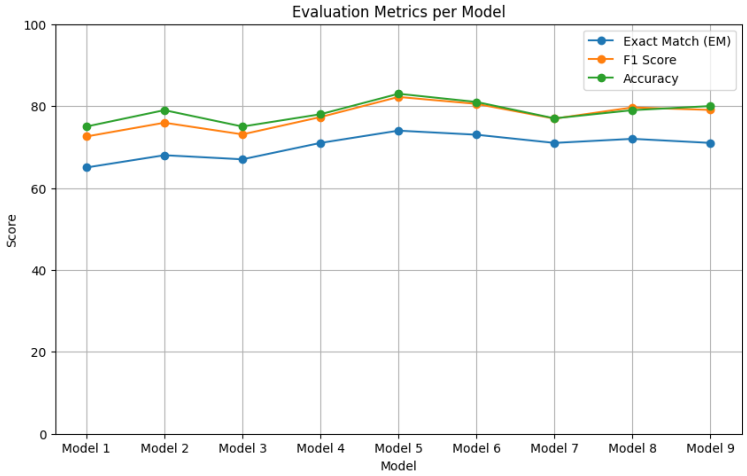
```python
em_scores = [row[0] for row in metrics]
f1_scores = [row[1] for row in metrics]
accuracy_scores = [row[2] * 100 for row in metrics]

# Numeric x-axis
x = list(range(len(metrics)))
model_labels = [f"Model {i+1}" for i in x]

# Compute average for each model
average_scores = [(em + f1 + acc) / 3 for em, f1, acc in zip(em_scores, f1_scores, accuracy_scores)]
best_model_index = average_scores.index(max(average_scores))
best_model_label = model_labels[best_model_index]

# Plot
plt.figure(figsize=(12, 6))
plt.plot(x, em_scores, label="Exact Match (EM)", marker='o')
plt.plot(x, f1_scores, label="F1 Score", marker='o')
plt.plot(x, accuracy_scores, label="Accuracy", marker='o')

# Annotate max values
def annotate_max(values, label):
    idx = values.index(max(values))
    plt.annotate(f"Max {label}: {max(values):.2f}\n({model_labels[idx]})",
                 (x[idx], values[idx]),
                 textcoords="offset points",
                 xytext=(0,10), ha='center',
                 fontsize=9, color='red', fontweight='bold')
```

```
annotate_max(em_scores, "EM")
annotate_max(f1_scores, "F1")
annotate_max(accuracy_scores, "Accuracy")

# Highlight best overall model
plt.axvline(x=best_model_index, color='gray', linestyle='--', alpha=0.5)
plt.text(best_model_index, 95, f"Best Avg: {best_model_label}\n({average_scores[best_model_index]:.2f})",
         ha='center', fontsize=9, bbox=dict(facecolor='yellow', alpha=0.3))

# Finalize plot
plt.xticks(ticks=x, labels=model_labels, rotation=45)
plt.title("Evaluation Metrics per Model")
plt.xlabel("Model")
plt.ylabel("Score")
plt.ylim(0, 100)
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

Results and Insights

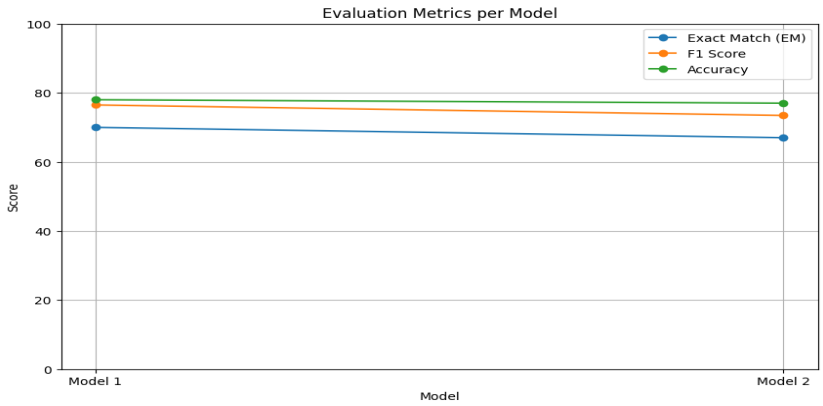
This analysis explores both the training and evaluation of the various models used across different phases, datasets and training conditions.

1. Phase 1 – Baseline Training (Models 1 to 9)
   This phase involves training 9 models on the full SQUAD dataset for just one epoch.
   Model 5 stood out with the strongest, most consistent metrics, making it a solid baseline.



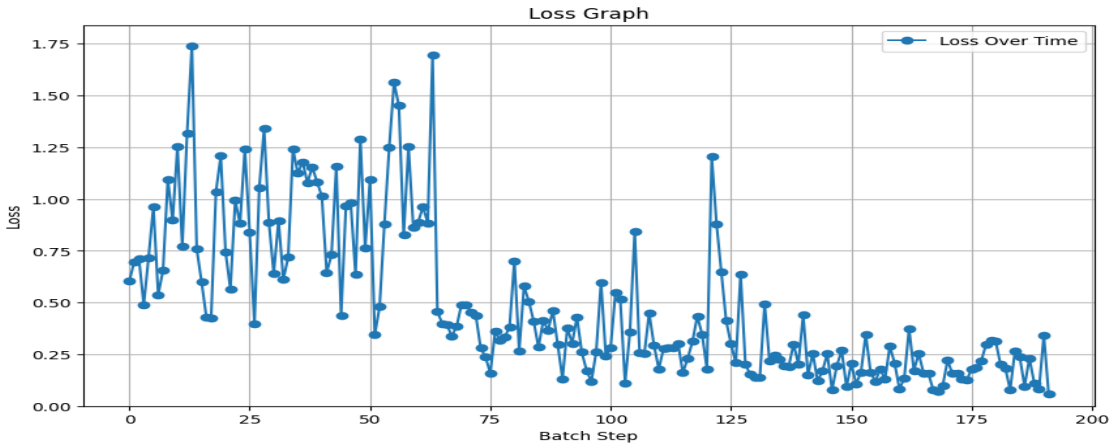
2. Phase 2 – Fine-tuning with small Dataset (Model 10)
   Model 5 was fine-tuned on a small dataset of 1000 examples for 5 epochs to simulate a low resource setting. This produced Model 10, which used a batch size of 16 and Model 11, which used a batch size of 8. Model 10 consistently outperformed model 11. Indicating a larger batch size was more effective for this setup.

Model 10 vs 11

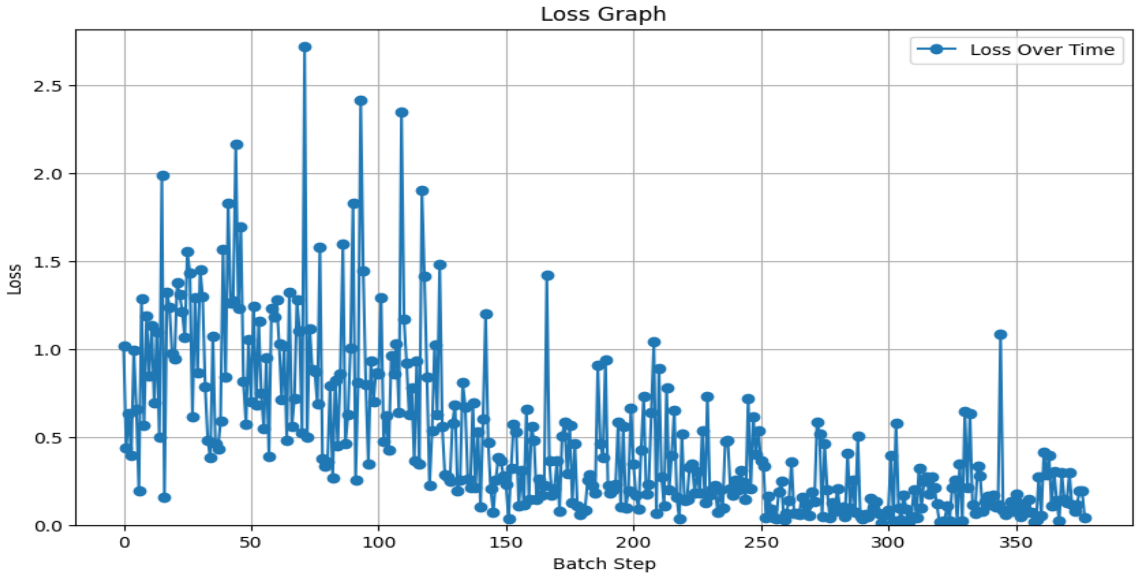


The loss graphs depict how the training loss decreases over batch steps. Model 10 has a smoother, more consistent graph which indicates Model 10 is more stable and hence more effective.

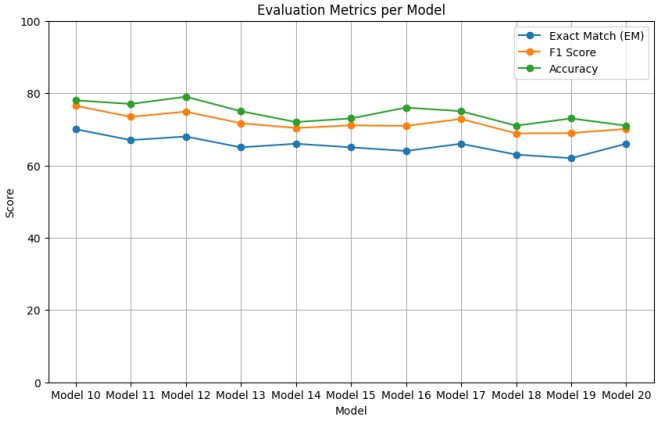
Model 10 Loss Graph



Model 11 Loss Graph



3. Phase 3 – Repeated Fine-Tuning (Models 12 to 20)
   Models 12 to 20 were trained for 5 epochs with a batch size of 16, each using a different randomly selected subset of the SQUAD training set.

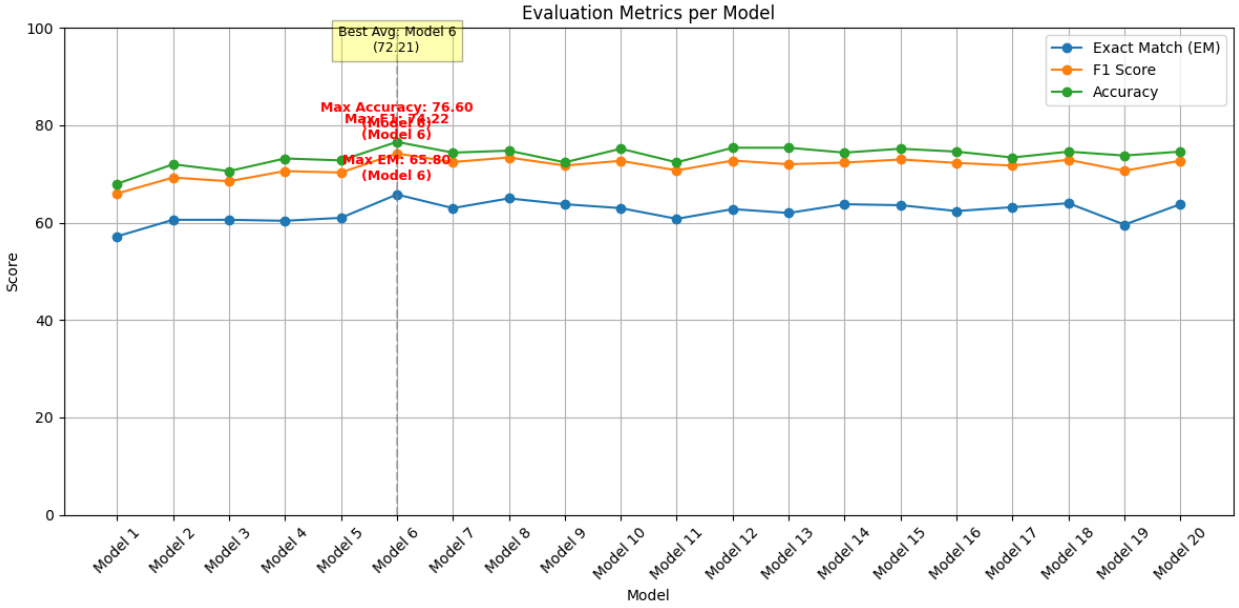


Conclusion

This project aimed to build an extractive question answering system using Hugging Face's Transformers library and the SQUAD dataset. We experimented with dataset size, batch size, number of training epochs and checkpointing strategies by training and evaluating 20 different models.

Using the same learning rate, controlled random sampling and model checkpointing gave us reproducible results and a clear understanding of each parameter.

In the final evaluations, including the average performance comparisons across top models, Model 6 emerged as the best overall performer.



Resources and APIs used:

1. Hugging Face Transformers Library:
   - A popular Python library that provides pre-trained models and tools for NLP.
     https://huggingface.co/docs/transformers/index

2. PyTorch:
   - An open source deep learning framework that provides tools for building, training, and deploying neural networks.
     https://docs.pytorch.org/docs/stable/

3. Google Colab:
   - A cloud-based platform that provides free access to powerful computing resources
     https://colab.research.google.com/notebooks/intro.ipynb

4. Progress Bar (tqdm):
   - A Python library that allows you to create progress bars to track the progress of loops and tasks in your code.
     https://tqdm.github.io/

5. Python Standard Library:
   - A collection of modules and packages that come bundled with Python, providing essential tools for handling tasks.
     https://docs.python.org/3/

6. SQuAD Dataset:
   - The Stanford Question Answering Dataset is a large-scale dataset consisting of question-answer pairs based on a set of Wikipedia articles.
     https://huggingface.co/datasets/rajpurkar/squad

7. File Management in Google Drive:
   - Google Drive is a cloud storage service that allows you to store, manage, and share files online.
     https://developers.google.com/workspace/drive