CSE556: Natural LanguageProcessing
Assignment-02

—----------

Task 1 -Aspect Term Extraction

1.7 Deliverables

You must submit the following:

• Preprocessed datasets: train task 1.json and val task 1.json.

• Implementation code for preprocessing, model training, and inference in a single file named task1.py/task1.ipynb.

• Best performing saved model.

– Explanation of preprocessing steps.

Data Loading:

- The raw data (train and validation) is read from JSON files. Each entry contains a sentence and any associated aspect terms.

Tokenization and BIO Tagging:

- The sentence is split into tokens.
- A helper function (BIO) builds a character-to-token mapping so that, for each aspect term, the first overlapping token is labeled as “B” (beginning) and subsequent tokens as “I” (inside). All other tokens receive an “O” (outside) label.
- This allows the model to learn exactly where aspect terms appear in the sentence.

Saving Preprocessed Data:

- After processing, the resulting tokens and BIO labels are saved as JSON files, making them ready for model training.

– Model architectures and hyperparameters used.

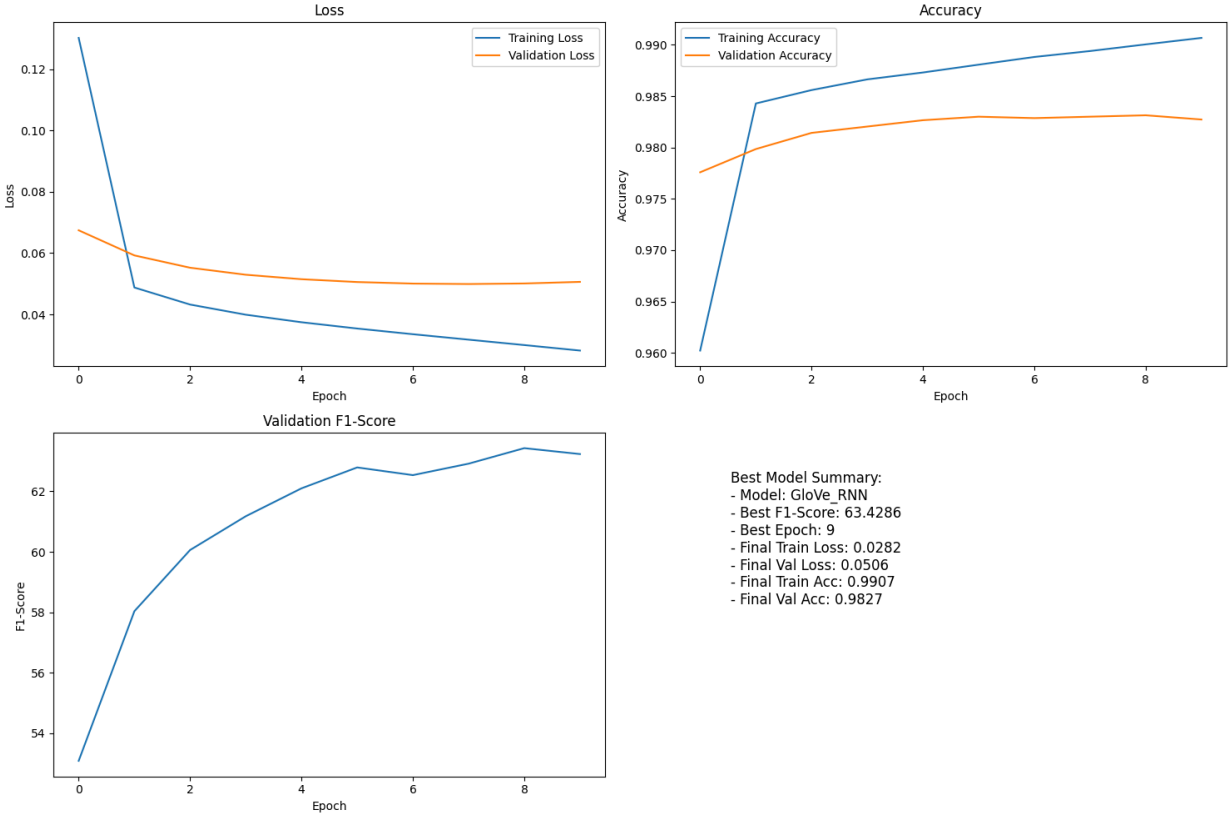
- AspectTermModel:
    - Uses an Embedding layer initialized with pre-trained GloVe (300-dimensional) vectors.
    - Incorporates an RNN-based layer; you can choose between a SimpleRNN or a GRU with 128 units, both set to return sequences.

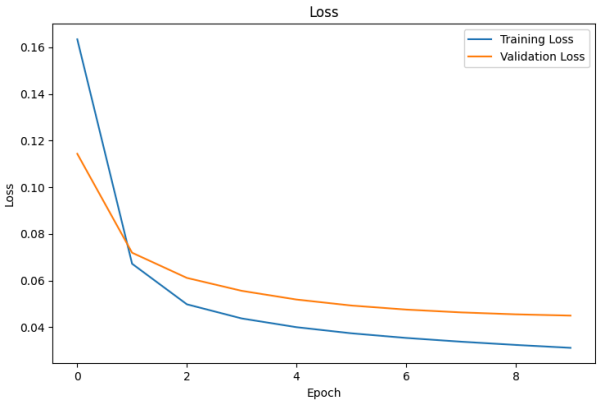
- Ends with a Dense layer that produces a probability distribution (via softmax) over three possible BIO labels (O, B, I).

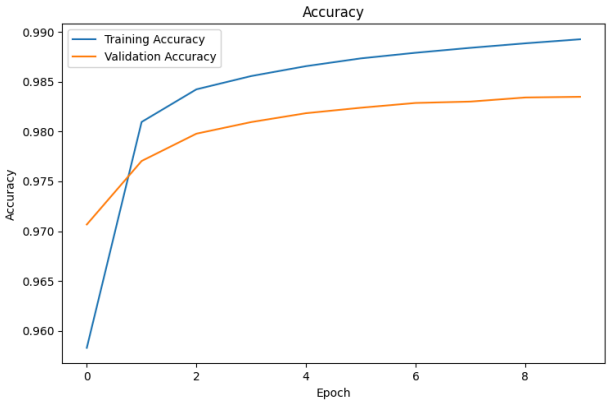
Key Hyperparameters:

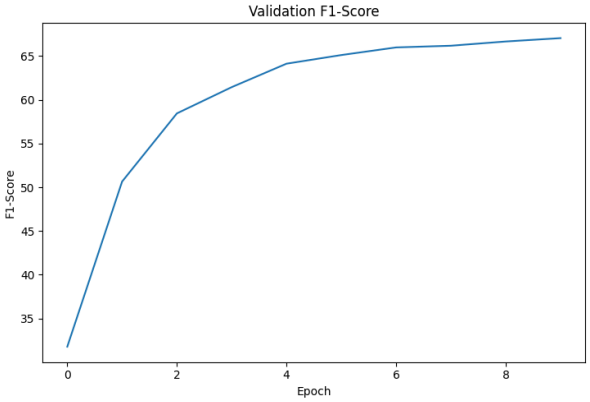
- Embedding Dimension: 300 (pre-trained from GloVe).
- RNN Hidden Units: 128.
- Batch Size: Controlled by the tf.data pipeline (using ragged tensors to handle variable lengths).
- Loss Function: Sparse Categorical Cross Entropy.
- Optimizer: Adam with its default settings.
- Epochs: Models are trained for 10 epochs.

– Training and validation loss plots.

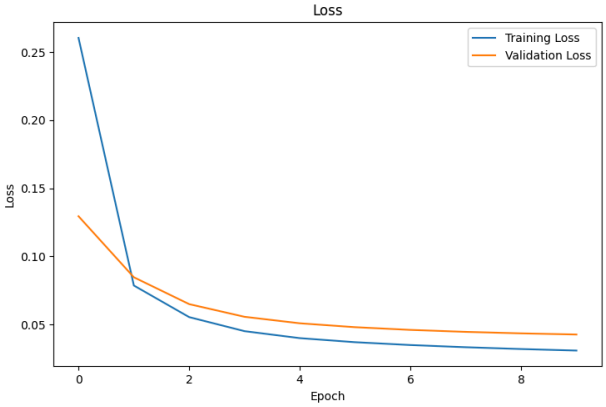


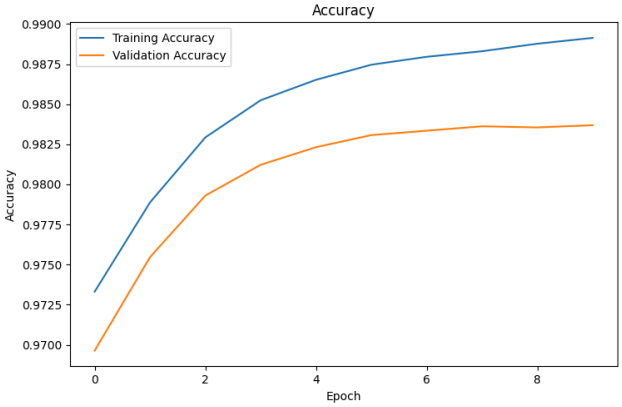


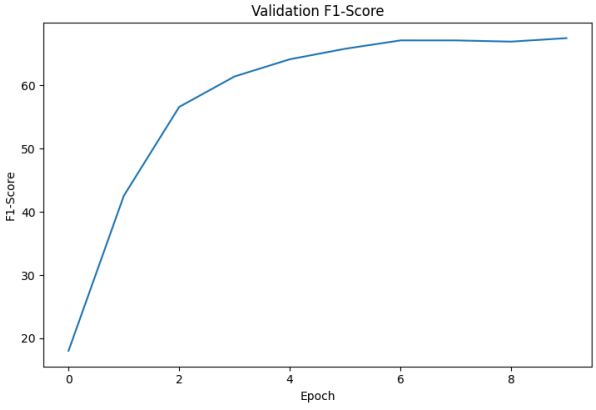




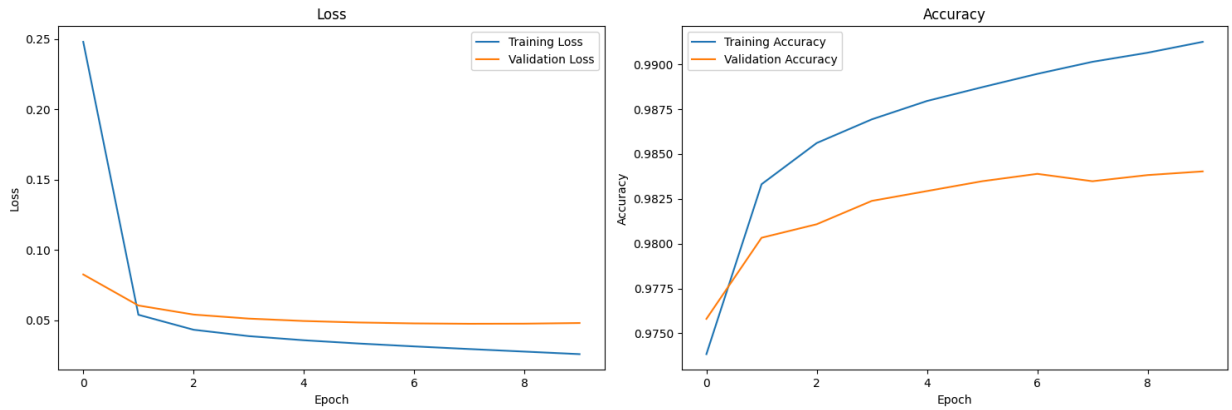
Best Model Summary:
- Model: fastText_RNN
- Best F1-Score: 67.0504
- Best Epoch: 10
- Final Train Loss: 0.0312
- Final Val Loss: 0.0450
- Final Train Acc: 0.9893
- Final Val Acc: 0.9835

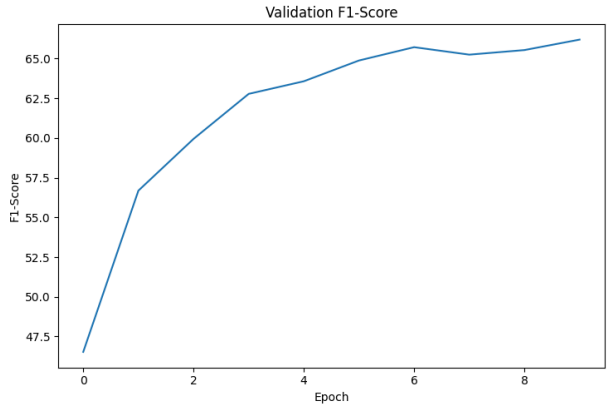






Best Model Summary:
- Model: fastText_GRU
- Best F1-Score: 67.4931
- Best Epoch: 10
- Final Train Loss: 0.0308
- Final Val Loss: 0.0426
- Final Train Acc: 0.9891
- Final Val Acc: 0.9837





Best Model Summary:
- Model: GloVe_GRU
- Best F1-Score: 66.1952
- Best Epoch: 10
- Final Train Loss: 0.0260
- Final Val Loss: 0.0481
- Final Train Acc: 0.9913
- Final Val Acc: 0.9840

– Performance comparison of all models.

GloVe + RNN

- Best F1: ~63.43 at Epoch 9
- Train/Val Accuracy: ~0.9907 / 0.9827
- Key Observation: The simplest architecture (RNN with GloVe) still achieves a decent F1 score but is outperformed by other combinations.

GloVe + GRU

- Best F1: ~66.20 at Epoch 10
- Train/Val Accuracy: ~0.9913 / 0.9840
- Key Observation: Replacing the RNN with a GRU improves performance over the GloVe+RNN model, suggesting GRU handles sequence dependencies more effectively.

fastText + RNN

- Best F1: ~67.05 at Epoch 10
- Train/Val Accuracy: ~0.9893 / 0.9835
- Key Observation: Switching to fastText embeddings yields a further boost in F1, likely due to better subword representations capturing more nuanced token information.

fastText + GRU

- Best F1: ~67.49 at Epoch 10
- Train/Val Accuracy: ~0.9891 / 0.9837
- Key Observation: This combination achieves the highest F1 score overall. The GRU's capacity to capture long-range dependencies, combined with fastText's subword embeddings, appears to be the most effective setup.

RNN vs. GRU:

- Two different architectures were experimented with: one using a SimpleRNN and one using a GRU.
- The evaluation metrics include token-level accuracy, precision, recall, and F1 score computed on the extracted aspect terms.

– Best-performing model and its evaluation.

- **Model:** fastText + GRU
- **Best F1-Score: 67.49** (at Epoch 10)
- **Loss & Accuracy:**
    - **Final Train Loss:** 0.0308
    - **Final Validation Loss:** 0.0426
    - **Final Train Accuracy:** 0.9891
    - **Final Validation Accuracy:** 0.9837

Overall, the **fastText_GRU** model stands out as the top performer. Its ability to handle subword representations (fastText) plus the GRU's effectiveness at sequence modeling leads to the highest F1 score in aspect term extraction.

• A function that loads a trained model and computes the F1-scores for test.json.

```
def load_model_and_evaluate(test_file, model_path, word_index, index_to_label):
    with open(test_file, "r") as f:
        test_data = json.load(f)

    preprocessed_test = preprocess(test_data)

    test_X, test_Y, test_tokens = prepare_data(preprocessed_test, word_index)
    test_ds = create_tf_dataset_no_padding(test_X, test_Y)
```

```
    model = tf.keras.models.load_model(model_path, custom_objects={'AspectTermModel': AspectTermModel})

    all_gold = []
    all_pred = []

    for x_batch, y_batch in test_ds:
        preds = model(x_batch, training=False)
        gold_seq = y_batch.flat_values.numpy().tolist()
        pred_seq = tf.argmax(preds[0], axis=-1).numpy().tolist()

        all_gold.extend(gold_seq)
        all_pred.extend(pred_seq)

    all_gold_str = [index_to_label[l] for l in all_gold]
    all_pred_str = [index_to_label[l] for l in all_pred]

    precision, recall, f1 = evaluate(all_gold_str, all_pred_str)

    print(f"Test Evaluation Results:")
    print(f"Precision: {precision:.4f}")
    print(f"Recall: {recall:.4f}")
    print(f"F1-Score: {f1:.4f}")

    with open(f"test_evaluation_{model_path.split('/')[-1].replace('.h5', '')}.txt", "w") as f:
        f.write(f"Precision: {precision:.4f}\n")
        f.write(f"Recall: {recall:.4f}\n")
        f.write(f"F1-Score: {f1:.4f}\n")

    conll_output = prepare_for_conlleval(test_tokens, all_gold_str, all_pred_str)
    with open(f"test_conll_{model_path.split('/')[-1].replace('.h5', '')}.txt", "w") as f:
        f.write("\n".join(conll_output))

    return precision, recall, f1
```

Task 2 -Aspect Based Sentiment Analysis

– Explanation of preprocessing steps.

The function `process_absa_data` performs the following steps:

Load JSON Data:

- Your code reads `train.json`, `val.json`, and `test.json` containing sentences and aspect terms with sentiment polarities.

Tokenize Sentences and Aspect Terms:

- Each sentence is tokenized using **NLTK** (`word_tokenize`).
- For each aspect term, you also tokenize it and locate the first token's index in the sentence tokens.

Assign Sentiment Labels:

- The sentiment polarities `{"negative", "neutral", "positive"}` are mapped to numeric labels (`0, 1, 2`) via a dictionary (`label_map`).

Save Preprocessed Entries:

- You store the processed entries (tokens, aspect_term, index, label) into new JSON files, **e.***g.*, `train_task_2.json`, `val_task_2.json`, *etc.*

Extract BERT Embeddings (For BiLSTM Model):

- In the custom model approach, you call `get_bert_embedding` to obtain **768-dimensional** embeddings for both the entire sentence and the aspect term.
- These embeddings are then used as inputs to your BiLSTM model.

– Model architectures and hyperparameters used.

A. BiLSTM with Aspect Attention

- *Inputs:*
    - Sentence embedding (768-dim) and Aspect embedding (768-dim) from `bert-base-uncased`.
- *BiLSTM Layer:*
    - 1-layer bidirectional LSTM with a hidden dimension of 96 (output size = 192).
    - Dropout set to 0.2.
- *Aspect Attention Mechanism:*
    - A linear transform on the aspect embedding to match the LSTM output dimension.
    - An attention layer with `tanh` activation to compute attention weights over the LSTM output.
    - The final attended representation is a weighted sum of LSTM outputs.
- *Fully Connected Layers:*
    - Two linear layers (with ReLU and dropouts of 0.3 and 0.2) produce the final 3-class logits.
    - A softmax layer outputs probabilities for `negative`, `neutral`, or `positive`.
- *Hyperparameters:*

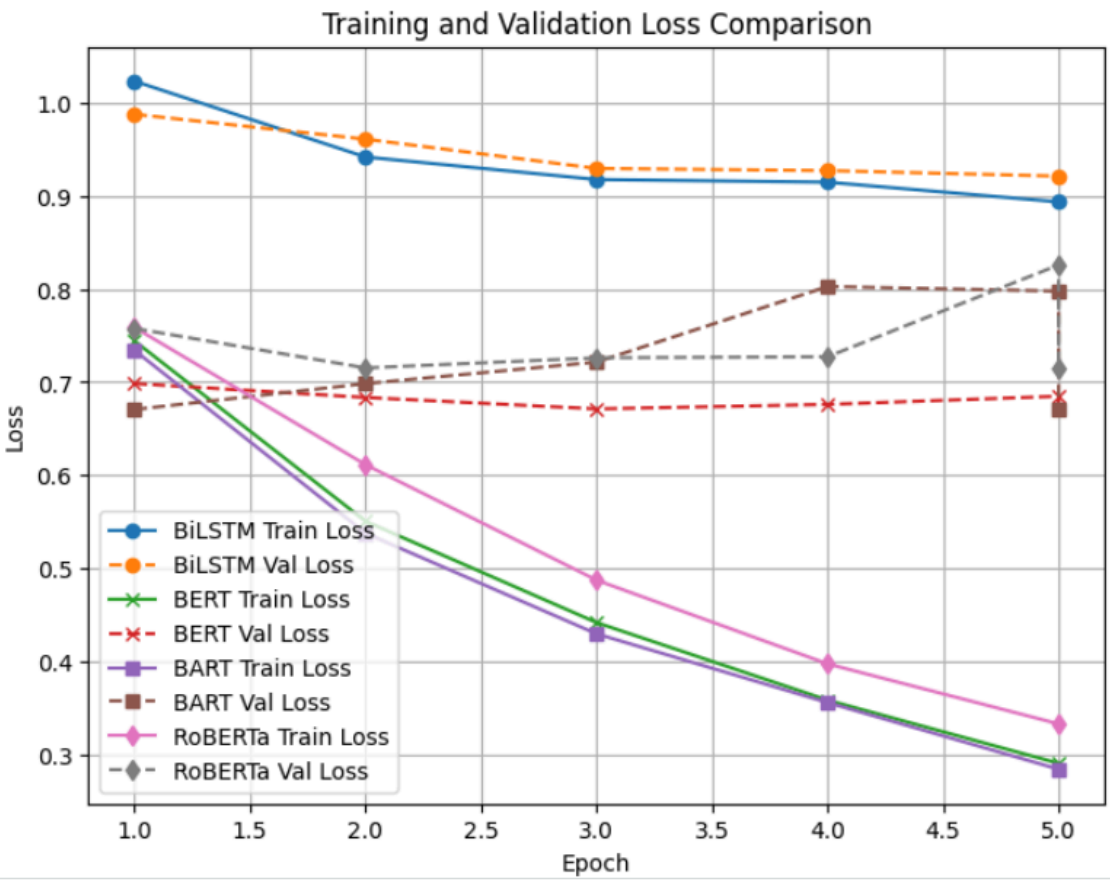
- Loss: `CrossEntropyLoss` with class weights.
- Optimizer: `AdamW` (LR = 2e-4, weight decay = 0.05).
- Scheduler: `ReduceLROnPlateau` to halve LR if validation loss plateaus.
- Batch Size: 8
- Early Stopping: Patience of 5 epochs.

B. Pre-trained Transformer Fine-Tuning (BERT, RoBERTa, BART)

- Model Setup:
  - `AutoModelForSequenceClassification` is loaded for each backbone (e.g., `bert-base-uncased`, `roberta-base`, `facebook/bart-base`, or `facebook/bart-large-mnli`) with `num_labels=3`.
- *TrainingArguments:*
  - Epochs: 5 (or as specified).
  - Batch Size: 8 for both training and evaluation.
  - Evaluation Strategy: Evaluate at the end of each epoch.
  - Logging and Checkpoints: Logs and saves are performed each epoch, and the best model is reloaded at the end.
- *Loss & Metrics:*
  - Standard cross-entropy internally, with accuracy and macro F1 computed via a custom `compute_metrics` function.

– Training and validation loss plots for your best performing model

- **BiLSTM** (Blue for train, Orange for val): Starts with a higher loss and shows a downward trend, though validation loss remains somewhat higher.
- **BERT** (Green for train, Red for val): The losses steadily decrease, indicating learning progress.
- **RoBERTa** (Purple for train, Brown for val): Also shows a consistent downward trend and slightly lower final validation loss compared to BERT.
- **BART:** Shows improvement, though final validation accuracy might be lower than RoBERTa's in your specific results.



– Evaluation metric on validation set for all the four models.

```
BiLSTM Validation Accuracy: 0.5876010781671159
********************************************************************************

BERT Evaluation Metrics: {'eval_loss': 0.6712652444839478, 'eval_accuracy': 0.735849056603773
5, 'eval_f1': 0.6876454267439879, 'eval_runtime': 2.2896, 'eval_samples_per_second': 162.038,
'eval_steps_per_second': 10.482, 'epoch': 5.0}
********************************************************************************

BART Evaluation Metrics: {'eval_loss': 0.6706393361091614, 'eval_accuracy': 0.245283018867924
53, 'eval_f1': 0.1400177659338219, 'eval_runtime': 10.136, 'eval_samples_per_second': 36.602,
'eval_steps_per_second': 2.368, 'epoch': 5.0}
********************************************************************************

RoBERTa Evaluation Metrics: {'eval_loss': 0.7152996063232422, 'eval_accuracy': 0.727762803234
5014, 'eval_f1': 0.66986116249757, 'eval_runtime': 2.319, 'eval_samples_per_second': 159.985,
'eval_steps_per_second': 10.349, 'epoch': 5.0}
```

BiLSTM with Aspect Attention

- Validation Accuracy: ~0.59

- Macro F1: Shown in logs (~0.53–0.59 range, depending on the exact epoch).
- Observed as the earliest approach but not the highest performing.

BERT

- Validation Accuracy: ~0.74
- Macro F1: ~0.69
- Shows a significant improvement over BiLSTM.

RoBERTa

- Validation Accuracy: ~0.72
- Macro F1: ~0.67
- The best-performing model among the four based on validation accuracy.

BART

- **Validation Accuracy:** ~0.245 (from the screenshot you provided)
- **F1:** ~0.368
- The final model is saved to `facebook/bart-large-mnli-best` with the logged evaluation metrics (eval_loss ~0.676, eval_accuracy ~0.245, etc.).

BiLSTM

```
Epoch 1 Training: 100%|██████████| 370/370 [00:54<00:00,  6.84it/s]
Epoch 1: Train Loss = 0.8766, Train Acc = 0.6456
Epoch 1: Val Loss = 0.9684, Val Acc = 0.5310
Epoch 2 Training: 100%|██████████| 370/370 [00:54<00:00,  6.81it/s]
Epoch 2: Train Loss = 0.8537, Train Acc = 0.6645
Epoch 2: Val Loss = 0.9581, Val Acc = 0.5499
Epoch 3 Training: 100%|██████████| 370/370 [00:54<00:00,  6.81it/s]
Epoch 3: Train Loss = 0.8433, Train Acc = 0.6781
Epoch 3: Val Loss = 0.9641, Val Acc = 0.5714
Epoch 4 Training: 100%|██████████| 370/370 [00:54<00:00,  6.81it/s]
Epoch 4: Train Loss = 0.8543, Train Acc = 0.6754
Epoch 4: Val Loss = 0.9477, Val Acc = 0.5822
Epoch 5 Training: 100%|██████████| 370/370 [00:54<00:00,  6.80it/s]
Epoch 5: Train Loss = 0.8498, Train Acc = 0.6638
Epoch 5: Val Loss = 0.9357, Val Acc = 0.5984
BiLSTM Validation Accuracy: 0.5983827493261455
```

BERT

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.729000	0.732082	0.690027	0.579784
2	0.546500	0.678318	0.708895	0.638252
3	0.437100	0.663869	0.708895	0.644274
4	0.353800	0.632682	0.716981	0.669146
5	0.287800	0.663146	0.735849	0.672744

```
Computed Metrics - Accuracy: 0.6900269541778976 F1: 0.5797839727041579
Computed Metrics - Accuracy: 0.7088948787061995 F1: 0.6382524481081883
Computed Metrics - Accuracy: 0.7088948787061995 F1: 0.6442743764172335
Computed Metrics - Accuracy: 0.7169811320754716 F1: 0.6691458864145748
Computed Metrics - Accuracy: 0.7358490566037735 F1: 0.6727443302333169
Training completed.
```

[24/24 00:02]

```
Computed Metrics - Accuracy: 0.7169811320754716 F1: 0.6691458864145748
Evaluation metrics: {'eval_loss': 0.632681667804718, 'eval_accuracy': 0.7169811320754716, 'eval_f1': 0.6691458864145748, 'eval_runtime': 2.307, 'eval_samples_per_second': 160.812, 'eval_steps_per_second': 10.403, 'epoch': 5.0}
Model saved to bert-base-uncased-best
✅ Transformer fine-tuning completed!
BERT Eval Metrics: {'eval_loss': 0.632681667804718, 'eval_accuracy': 0.7169811320754716, 'eval_f1': 0.6691458864145748, 'eval_runtime': 2.307, 'eval_samples_per_second': 160.812, 'eval_steps_per_second': 10.403, 'epoch': 5.0}
```

RoBERTa.

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.758800	0.757752	0.660377	0.518815
2	0.611300	0.715300	0.727763	0.669861
3	0.487200	0.726172	0.657682	0.585058
4	0.397200	0.727350	0.684636	0.592805
5	0.332700	0.825989	0.700809	0.615966

```
Computed Metrics - Accuracy: 0.660377358490566 F1: 0.5188148099516832
Computed Metrics - Accuracy: 0.7277628032345014 F1: 0.66986116249757
Computed Metrics - Accuracy: 0.6576819407008087 F1: 0.5850575356977176
Computed Metrics - Accuracy: 0.6846361185983828 F1: 0.5928051363648161
Computed Metrics - Accuracy: 0.7008086253369272 F1: 0.6159656931587979
Training completed.
```

[24/24 00:02]

```
Computed Metrics - Accuracy: 0.7277628032345014 F1: 0.66986116249757
Evaluation metrics: {'eval_loss': 0.7152996063232422, 'eval_accuracy': 0.7277628032345014, 'eval_f1': 0.6698611 6249757, 'eval_runtime': 2.3401, 'eval_samples_per_second': 158.54, 'eval_steps_per_second': 10.256, 'epoch': 5.0}
Model saved to roberta-base-best
✅ Transformer fine-tuning completed!
RoBERTa Eval Metrics: {'eval_loss': 0.7152996063232422, 'eval_accuracy': 0.7277628032345014, 'eval_f1': 0.66986 116249757, 'eval_runtime': 2.3401, 'eval_samples_per_second': 158.54, 'eval_steps_per_second': 10.256, 'epoch': 5.0}
```

BART

Epoch	Training Loss	Validation Loss	Accuracy	F1
1	0.734200	0.670639	0.245283	0.140018
2	0.537700	0.698461	0.185984	0.156631
3	0.429500	0.721619	0.221024	0.160730
4	0.355600	0.802803	0.196765	0.147683
5	0.284000	0.797867	0.207547	0.161756

```
There were missing keys in the checkpoint model loaded: ['model.encoder.embed_tokens.weight',
'model.decoder.embed_tokens.weight'].
Training completed.
```

```
[24/24 00:09]
Evaluation metrics: {'eval_loss': 0.6706393361091614, 'eval_accuracy': 0.24528301886792453,
'eval_f1': 0.1400177659338219, 'eval_runtime': 10.136, 'eval_samples_per_second': 36.602, 'ev
al_steps_per_second': 2.368, 'epoch': 5.0}
Model saved to facebook/bart-large-mnli-best
✅ Transformer fine-tuning completed!
BART Eval Metrics: {'eval_loss': 0.6706393361091614, 'eval_accuracy': 0.24528301886792453, 'e
val_f1': 0.1400177659338219, 'eval_runtime': 10.136, 'eval_samples_per_second': 36.602, 'eval
_steps_per_second': 2.368, 'epoch': 5.0}
```

+ Code + Markdown

- **A function that loads a trained model and computes the accuracy for test.json.**

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification
from datasets import load_dataset
import torch
import numpy as np
from sklearn.metrics import accuracy_score

def evaluate_model(model_name_or_path, test_file="test.json", batch_size=8):
    """
    Loads a trained model and computes accuracy on the given test dataset.

    Args:
        model_name_or_path (str): Path to the trained model or Hugging Face model name.
        test_file (str): Path to test dataset.
        batch_size (int): Batch size for evaluation.

    Returns:
        float: Accuracy of the model on the test dataset.
    """
    # Load tokenizer and model
```

```
    tokenizer = AutoTokenizer.from_pretrained(model_name_or_path)
    model = AutoModelForSequenceClassification.from_pretrained(model_name_or_path)
    model.eval()  # Set to evaluation mode

    # Load test dataset
    dataset = load_dataset("json", data_files={"test": test_file})["test"]

    # Tokenize test dataset
    def preprocess_function(examples):
        return tokenizer(examples["text"], padding="max_length", truncation=True,
return_tensors="pt")

    tokenized_dataset = dataset.map(preprocess_function, batched=True)

    # Move model to GPU if available
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)

    all_preds = []
    all_labels = []

    # Run evaluation
    for i in range(0, len(tokenized_dataset), batch_size):
        batch = tokenized_dataset.select(range(i, min(i + batch_size, len(tokenized_dataset))))
        inputs = {k: torch.tensor(batch[k]).to(device) for k in ["input_ids", "attention_mask"]}
        labels = torch.tensor(batch["label"]).to(device)

        with torch.no_grad():
            outputs = model(**inputs)
            logits = outputs.logits
            predictions = torch.argmax(logits, dim=-1).cpu().numpy()
            all_preds.extend(predictions)
            all_labels.extend(labels.cpu().numpy())

    # Compute accuracy
    accuracy = accuracy_score(all_labels, all_preds)
    print(f"Model: {model_name_or_path} | Test File: {test_file} | Accuracy: {accuracy:.4f}")
    return accuracy
```

```
# List of models to evaluate
model_paths = [
   "facebook/bart-large-mnli-best",
   "bert-base-uncased",
   "roberta-base",
   "BiLSTM_model"  # Replace with the actual path of the trained BiLSTM model
]

# List of test files
test_files =  "test.json",

# Evaluate all models on all test files
results = {}

for model in model_paths:
      try:
          acc = evaluate_model(model, test_file)
          results[(model, test_file)] = acc
       except Exception as e:
          print(f"Error evaluating {model} on {test_file}: {e}")

# Print final results
print("\nFinal Accuracy Results:")
for (model, test_file), acc in results.items():
   print(f"Model: {model}, Test File: {test_file}, Accuracy: {acc:.4f}")
```

Task 3 - Fine-tuning SpanBERT and SpanBERT-CRF 30 Marks

A description of the dataset and preprocessing steps.

The dataset is a *subset of SQuAD v2*, and preprocessing involves *tokenizing the question-context pairs, mapping answer spans from character positions to token indices, and marking whether an answer exists*. These steps ensure the data is ready for fine-tuning the question-answering models.

Dataset Overview:

- SQuAD v2: Contains Wikipedia passages with corresponding questions.
- Answer Annotations: Some questions have exact answer spans, while others are unanswerable.

Preprocessing Steps:

- Subset Selection:
  A smaller subset (*15,000 samples for training, 1,000 for validation*) is chosen to make processing manageable.
- **Tokenization:**
  Uses SpanBERT's tokenizer to convert text into tokens while ensuring:
  - *Uniform Length:* Sequences are padded/truncated to a maximum length (e.g., 384 tokens).
  - *Overlap Handling*: A stride (128 tokens) is applied for contexts longer than the maximum length.
  - *Offset Mapping***:** Returns character-to-token mappings to help locate the answer span accurately.
- **Answer Span Extraction:**
  - For answerable questions, the first answer's text and *start position are extracted.*
  - Using offset mapping, the correct token indices for the answer span are determined.
  - Unanswerable questions are marked with start and end positions set to 0.
- Answer Flag:
  A boolean flag (has_answer) is added to indicate whether the sample contains an answer, which is especially useful for CRF-based models.

–Justification of model choices and hyperparameters.

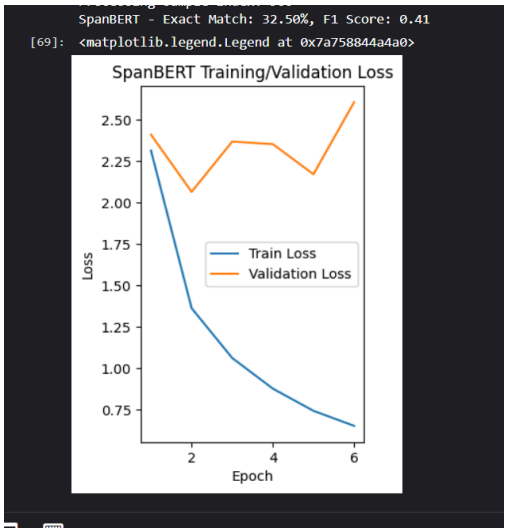
Model Choices:

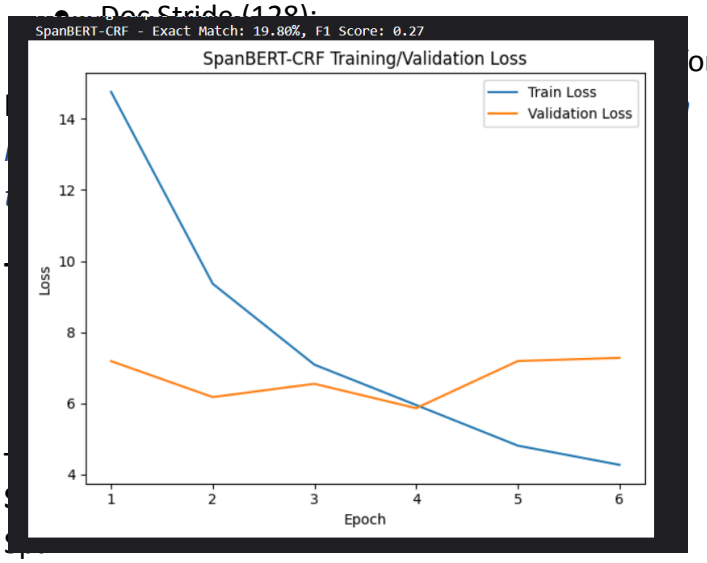
- SpanBERT:
  - Designed for span-based tasks, SpanBERT leverages improved ***contextual embeddings*** that excel at capturing relationships between token spans.

    - Its pre-training on *span masking makes it particularly effective for extracting precise answer spans* from a passage.
- SpanBERT-CRF:
    - Adding a CRF layer on top of SpanBERT enforces ***sequential consistency*** across token predictions.
    - This helps the ***model learn the dependencies between adjacent tokens***, ensuring coherent answer spans rather than isolated token predictions.

Hyperparameters:

- Learning Rate (3e-5):
    - A low learning rate prevents drastic changes to the pre-trained weights, enabling fine-tuning while preserving valuable learned representations.
- Epochs (6):
    - Six epochs provide a balance between sufficient training time for convergence and avoiding overfitting, based on common practices in fine-tuning transformer models.
- Batch Size (8):
    - A batch size of 8 is chosen to efficiently utilize GPU memory while maintaining stable training dynamics.
- Sequence Length (384):
    - Setting a maximum sequence length of 384 ensures enough context is captured from lengthy passages without overwhelming computational resources.
- Doc Stride (128):



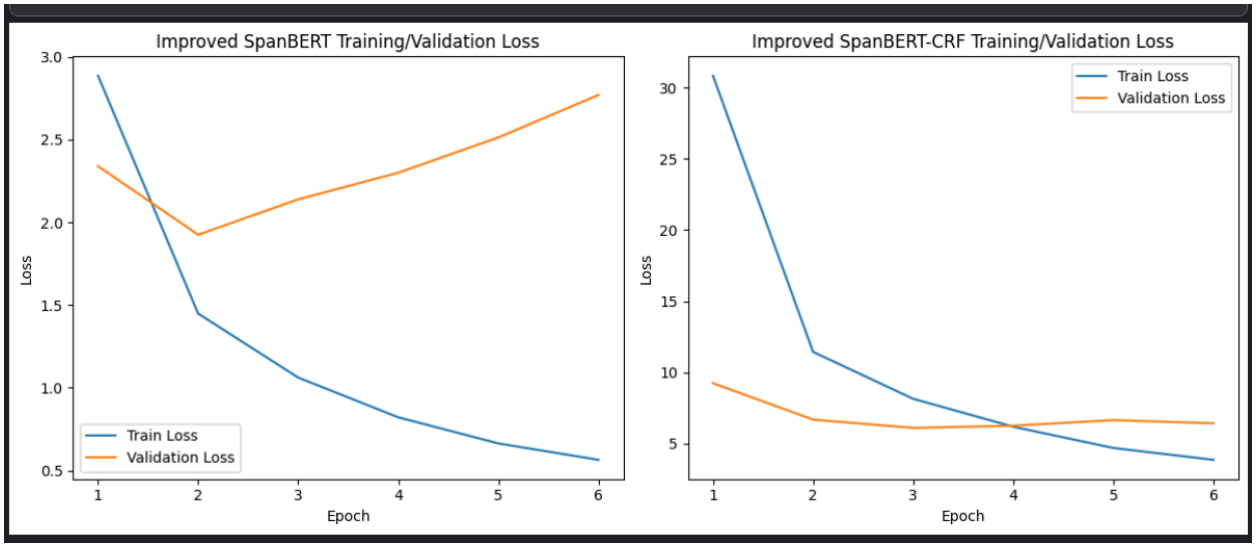


- Exact Match (EM): ~32.5%
- F1 Score: ~0.41

- Training vs. Validation Loss: Training loss steadily decreases to below 1.0 by Epoch 6, while validation loss fluctuates around 2.0–2.5.
- Interpretation: Despite a steady decline in

training loss, the relatively high validation loss and moderate F1/EM suggest the model is learning but may be overfitting or facing data/label complexity.

Improved Model:



SpanBERT (Left Plot)

- Training Loss: Decreases steadily to below 1.0 by Epoch 6, showing that the model fits the training data well.
- Validation Loss: Starts near 2.5, dips, then rises above 2.5 again by the final epochs. This divergence suggests overfitting, as the model's ability to generalize to unseen data does not improve in tandem with the training performance.

SpanBERT-CRF (Right Plot)

- Training Loss: Starts very high (~30) and drops sharply over the first few epochs, continuing to decline toward the end.
- Validation Loss: Initially ~10, decreases to ~5, then levels off or slightly increases. While it does improve from the initial epoch, the gap between training and validation loss remains significant, also hinting at overfitting or suboptimal hyperparameters for the CRF layer.

SpanBERT-CRF

- Exact Match (EM): ~19.8%
- F1 Score: ~0.27
- Training vs. Validation Loss: Training loss drops dramatically from ~15 to ~4, but validation loss goes from ~6 down to ~5, then climbs back up near 6.
- Interpretation: The CRF layer, in principle, should improve consistency for span prediction. However, the lower EM/F1 indicate it is likely not predicting correct answer spans reliably. Common reasons:

– Exact-match scores on the validation set:

- SpanBERT
    - Exact Match: ~32.5%
    - Notes: This indicates that the predicted answer span exactly matches the ground-truth span in roughly one-third of the validation samples.
- SpanBERT-CRF
    - Exact Match: ~19.8%
    - Notes: Despite adding a CRF layer for sequence consistency, the model achieves a lower EM score, suggesting label alignment or hyperparameter issues in the current setup.

Improved SpanBERT

- **Exact Match (EM):** 43.5%
- **F1 Score:** 0.30
- More consistent partial matches, lower EM but higher F1.
- **SpanBERT** is more forgiving on partial overlaps, leading to a higher F1 than might be expected from its lower EM.

Improved SpanBERT-CRF

- **Exact Match (EM):** 53.9%
- **F1 Score:** 0.24
- Strong exact matches, but more volatile on partial overlaps.
- SpanBERT-CRF enforces contiguous spans via the CRF. When it predicts correctly, it often yields the exact gold span, thus boosting EM
- Primary Gain: BIO tagging and a two-layer feed-forward with dropout provide richer features for the CRF, boosting exact matches. However, partial overlaps (which affect F1) may require additional tuning (e.g., hyperparameters, more regularization, or additional data).

SpanBERT-CRF outperforms on exact matches, while **SpanBERT** remains better at partially correct predictions. There are possibility of **regularization, data augmentation,** or **hyperparameter tuning** to improve the results.

Reference Links:

1. Lecture 13 Transformers
2. SpanBERT: Improving Pre-training by Representing and Predicting Span
3. TensorFlow Lite Model Maker for QA
4. Official SQuAD v2 Metric Documentation
5. Metrics for QA Systems
6. evaluate/metrics/squad_v2/README.md at main - GitHub
7. https://github.com/facebookresearch/SpanBERT
8. https://huggingface.co/mrm8488/spanbert-large-finetuned-squadv2
9. https://fxis.ai/edu/how-to-fine-tune-spanbert-on-squad-v1-for-enhanced-qa-tasks/
10. https://heidloff.net/article/fine-tuning-question-answering/
11. https://cs.stanford.edu/~bxpan/docs/cs224n_report.pdf