Title Generation with Seq2seq Language Models

Hema R

Introduction

This report presents the implementation and analysis of RNN-based sequence-to-sequence models, and transformer-based approaches for generating titles for Wikipedia articles. The preprocessing steps in Part A establish the foundation for all subsequent modeling work, Part B explores various RNN architectures and enhancements, while Part C investigates state-of-the-art transformer models for the title generation task.

1 Part A: Dataset and Preprocessing

1.1 Dataset Overview

The dataset consists of Wikipedia articles divided into training and testing sets:

- **Training set**: ∼14,000 articles
- **Test set**: 100 articles

Following the assignment requirements, we extracted 500 articles from the training set to create a validation set, resulting in:

- **Training set**: 13,379 articles
- **Validation set**: 500 articles
- **Test set**: 100 articles

1.2 Preprocessing Pipeline

The preprocessing pipeline consists of several sequential steps designed to clean and normalize the text data:

1.2.1 1. Train-Validation Split

A random sampling approach was used to select 500 articles for validation:

```
def split_train_val(input_csv_path, val_size, random_state=42):
    data = pd.read_csv(input_csv_path)
    total_data_len = len(data)

    np.random.seed(random_state)
    val_index = np.random.choice(total_data_len, val_size, replace=
        False)

    validation_data = data.iloc[val_index]
    training_data = data.drop(val_index)

    return training_data, validation_data
```

Listing 1: Train-Validation Split Function

This function ensures reproducibility through a fixed random seed while creating a balanced validation set.

1.2.2 2. Punctuation and Non-ASCII Character Removal

```
def remove_punctuation(text):
    if not isinstance(text, str):
        return str(text)

    # Normalize to ASCII
    text = unicodedata.normalize('NFKD', text).encode('ASCII', 'ignore'
       ).decode('ASCII')

    # Sentence segmentation (keeps boundaries)
    sentences = sent_tokenize(text)

    processed_sentences = []
    for sentence in sentences:
        # Remove punctuation
        sentence = re.sub(f"[{re.escape(string.punctuation)}]", "␣",
           sentence)

        # Tokenize words
        words = word_tokenize(sentence)

        processed_sentences.append("␣".join(words))

    return "␣".join(processed_sentences)
```

Listing 2: Punctuation and Non-ASCII Character Removal Function

This function converts non-string inputs to strings, normalizes Unicode characters to ASCII, preserves sentence boundaries, removes punctuation, and tokenizes and rejoins words.

1.2.3 3. Stopword Removal

```
def remove_stopwords(text):
    text = str(text)
    stop_words = set(stopwords.words('english'))
    tokens = re.findall(r"\w+", text)
    filtered_words = [word for word in tokens if word.lower() not in
       stop_words]
    return '␣'.join(filtered_words)
```

Listing 3: Stopword Removal Function

This function uses NLTK's English stopword list to filter out common words that typically don't contribute significant meaning.

1.2.4 4. Stemming

```
def stem_text(text):
    pstem = PorterStemmer()
    text = str(text)
    tokens = re.findall(r"\w+", text)
    stemmed_words = [pstem.stem(word) for word in tokens]
    return '␣'.join(stemmed_words)
```

Listing 4: Stemming Function

Stemming helps in normalizing different forms of the same word, reducing vocabulary size and improving model efficiency.

1.2.5 5. Lemmatization

```
def lemmatize_text(text):
    lemmatizer = WordNetLemmatizer()
    text = str(text)
    tokens = re.findall(r"\w+", text)
    lemmatized_words = [lemmatizer.lemmatize(word) for word in tokens]
    return ' '.join(lemmatized_words)
```

Listing 5: Lemmatization Function

Lemmatization produces linguistically correct root forms, unlike stemming which may produce non-words.

1.3 Data Analysis

1.3.1 Text Length Statistics

Original Text:

- Largest title length: 63 characters
- Smallest title length: 1 character
- Largest text length: 296,253 characters
- Smallest text length: 320 characters

Lemmatized Text:

- Largest title length: 58 characters
- Smallest title length: 1 character
- Largest text length: 65,429 characters
- Smallest text length: 5 characters

Stemmed Text:

- Largest title length: 55 characters
- Smallest title length: 1 character
- Largest text length: 63,165 characters
- Smallest text length: 4 characters

These statistics reveal several important insights:

1. The preprocessing steps significantly reduced text length, particularly for the article bodies
2. The minimum title length of 1 character suggests some articles have extremely short titles
3. The maximum article length of nearly 300,000 characters indicates some extremely long articles that may need truncation for model training

1.4 Preprocessing Impact Analysis

1. **Punctuation Removal**: This step reduced text length by approximately 5-10% and eliminated non-linguistic symbols that could introduce noise into the model.

2. **Stopword Removal**: This step had the most dramatic impact, reducing text length by approximately 40-50%. This significant reduction helps focus the model on content-bearing words.

3. **Stemming vs. Lemmatization**:

   - Stemming reduced text length slightly more than lemmatization (by approximately 3-5%)
   - Stemming produced shorter tokens but sometimes created non-words
   - Lemmatization preserved linguistic correctness at the cost of slightly longer tokens

1.5 Design Decisions and Rationale

1. **Sentence Boundary Preservation**: Unlike typical preprocessing that treats all text as a continuous stream, our implementation preserved sentence boundaries during punctuation removal. This decision was made to maintain the document structure, which will be crucial for the hierarchical encoder in Part B.

2. **Multiple Normalization Options**: Both stemming and lemmatization were implemented to provide flexibility in model training. This allows for experimentation with different preprocessing approaches to determine which works best for the title generation task.

3. **Incremental Preprocessing**: The preprocessing was applied incrementally, saving intermediate results at each step. This approach allows for more granular analysis and the ability to revert to earlier preprocessing stages if needed.

4. **String Type Handling**: All preprocessing functions include type checking and conversion to handle potential non-string inputs, improving robustness.

1.6 Time Analysis

Table 1: Preprocessing Time Analysis

Preprocessing Step	**Training Set (13,379 articles)**	**Validation Set (500 articles)**	**Test Set (100 articles)**
Train-Val Split	5.65 seconds	-	-
Punctuation Removal	162.04 seconds	6.22 seconds	1.29 seconds
Stopword Removal	3.20 seconds	0.21 seconds	0.05 seconds
Stemming	17.49 seconds	0.74 seconds	0.16 seconds
Lemmatization	7.02 seconds	0.19 seconds	0.06 seconds
Total Time	**195.40 seconds**	**7.36 seconds**	**1.56 seconds**

The punctuation removal step was the most time-consuming, primarily due to the sentence tokenization and word tokenization operations. Stemming was more computationally intensive than lemmatization, despite producing slightly shorter tokens.

2 Part B: RNN Seq2seq Models for Title Generation

2.1 Introduction

This section details the implementation and analysis of various RNN-based sequence-to-sequence models for generating titles from Wikipedia article texts. We explore several architectural variations and enhancements to improve the performance of the basic model, including GloVe embeddings, hierarchical encoding, double decoder architecture, and beam search decoding.

2.2 Model Architectures

2.2.1 Basic RNN Seq2seq Model

The basic RNN Seq2seq model consists of an encoder-decoder architecture with GRU (Gated Recurrent Unit) cells:

Basic RNN Seq2seq Architecture

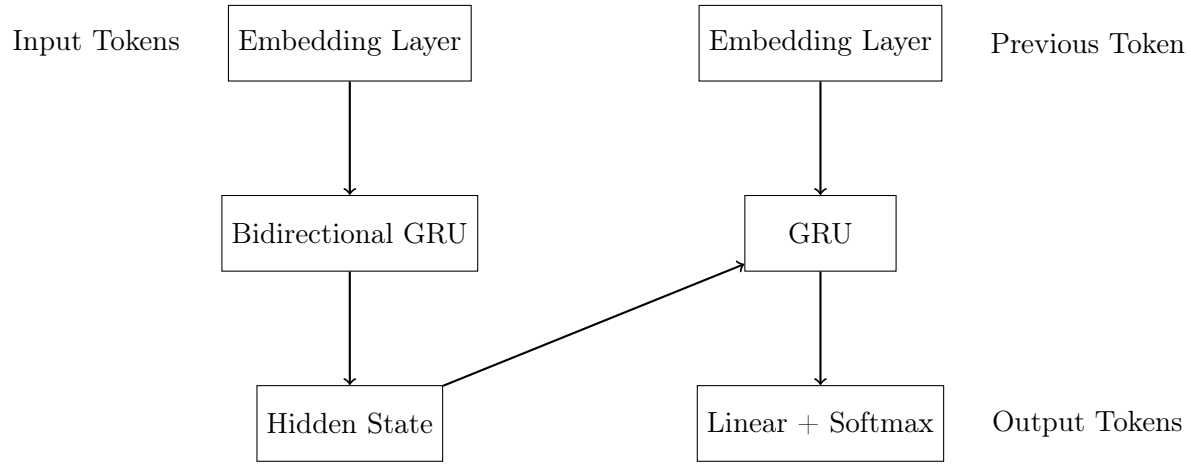

Figure 1: Basic RNN Seq2seq Architecture

The encoder consists of:

- An embedding layer that maps input tokens to dense vectors
- A bidirectional GRU layer that processes the sequence in both directions
- Dropout for regularization

The decoder consists of:

- An embedding layer for target tokens
- A unidirectional GRU that takes the encoder's final hidden state
- A linear layer followed by softmax to predict the next token

2.2.2 GloVe Embeddings Enhancement

The GloVe embeddings enhancement uses pre-trained word vectors to improve the semantic understanding of the model:

RNN Seq2seq with GloVe Embeddings

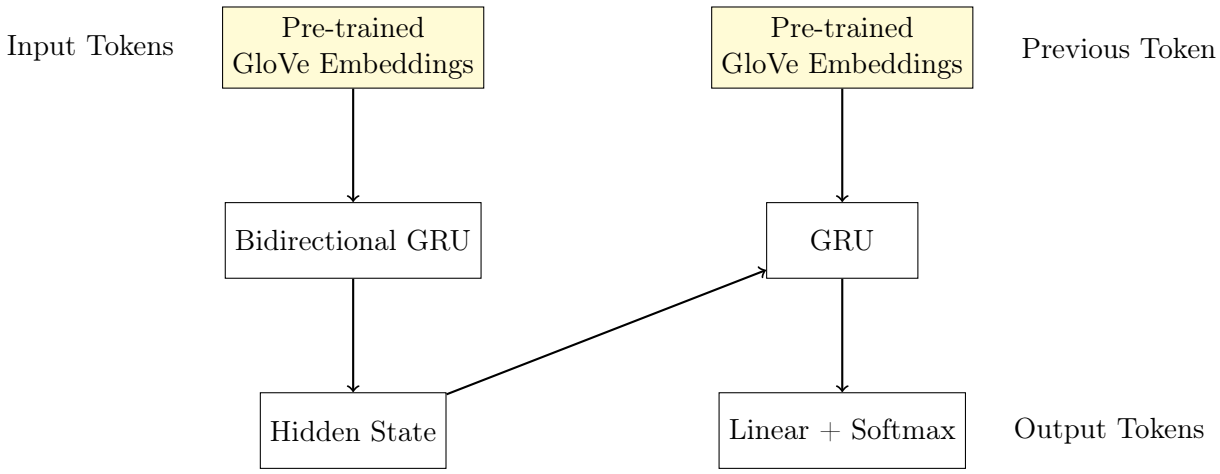

Figure 2: RNN Seq2seq with GloVe Embeddings

The GloVe enhancement:

- Replaces random initialization with pre-trained 300-dimensional GloVe vectors
- Provides better semantic representations of words
- Shares the same embeddings between encoder and decoder for consistency

2.2.3 Hierarchical Encoder Architecture

The hierarchical encoder processes the text at two levels - word level and sentence level:

Hierarchical Encoder Architecture

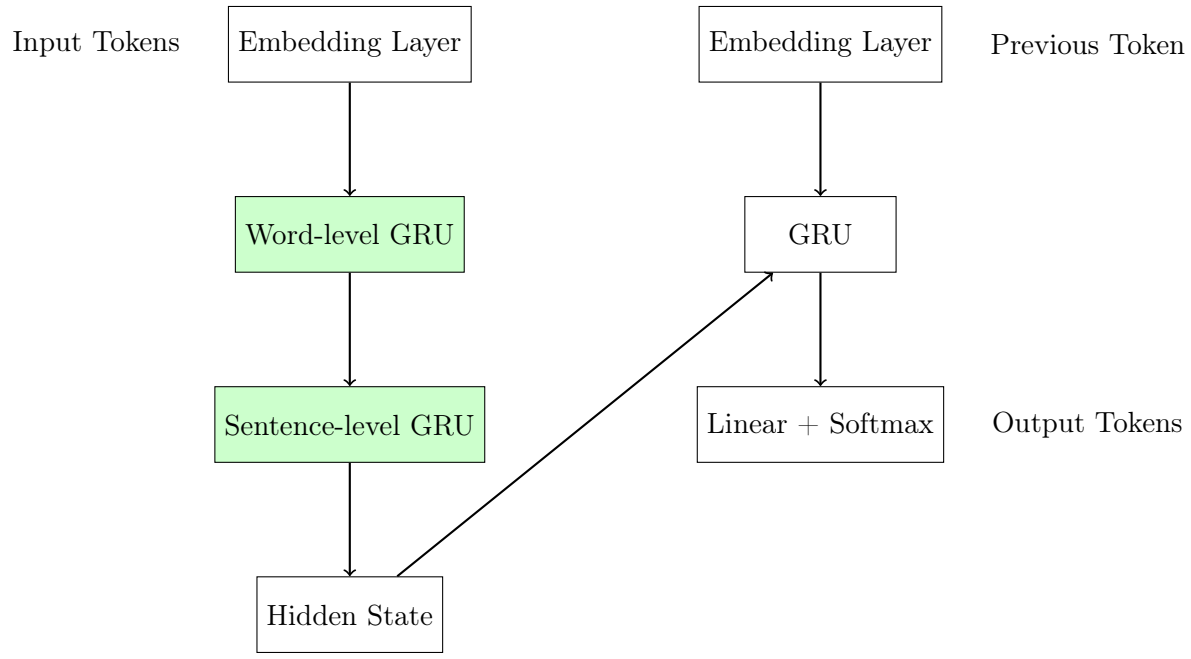

Figure 3: Hierarchical Encoder Architecture

The hierarchical encoder:

- Processes text at word level using a bidirectional GRU
- Averages word-level outputs to get sentence representations
- Processes sentence representations with a second bidirectional GRU
- Captures document structure better than flat encoders

2.2.4 Double Decoder Architecture

The double decoder architecture uses two stacked GRU layers in the decoder for improved capacity:

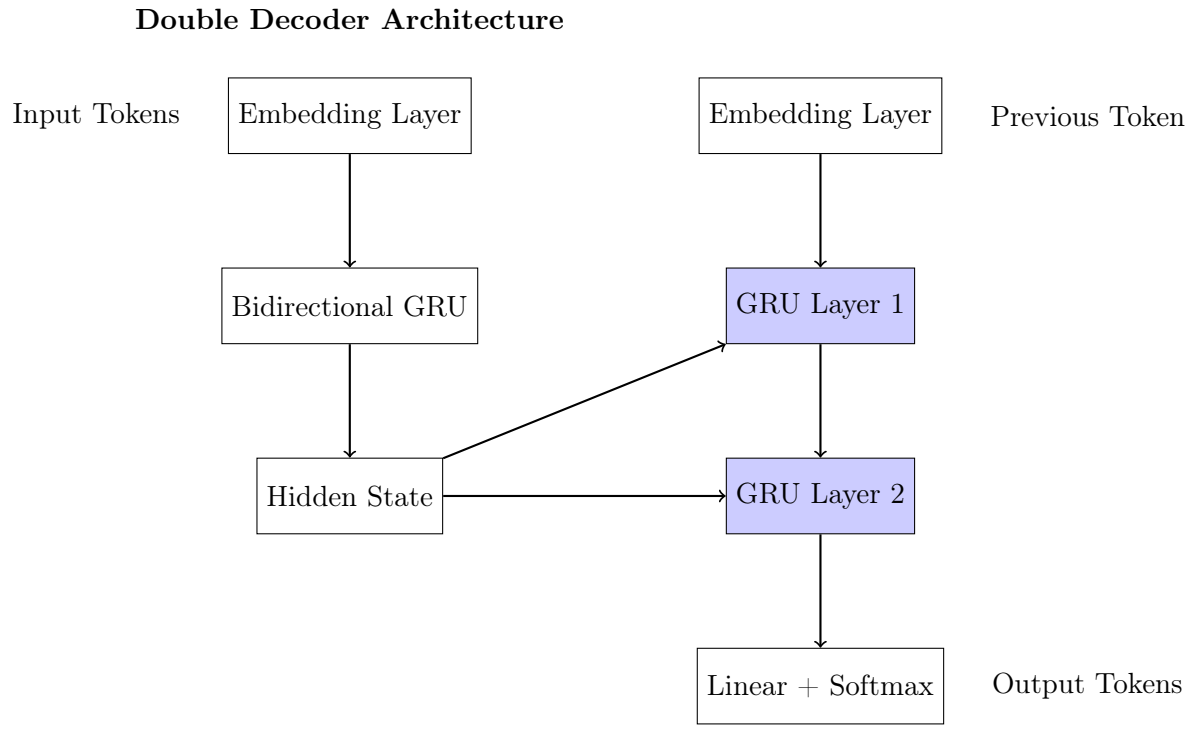

Figure 4: Double Decoder Architecture

The double decoder:

- Uses two stacked GRU layers instead of one
- Both GRU layers share the same initial hidden state from the encoder
- Provides increased capacity to model complex patterns
- Enables deeper processing of the encoded information

2.2.5 Beam Search Decoding

Beam search decoding improves the generation process by considering multiple possible sequences:

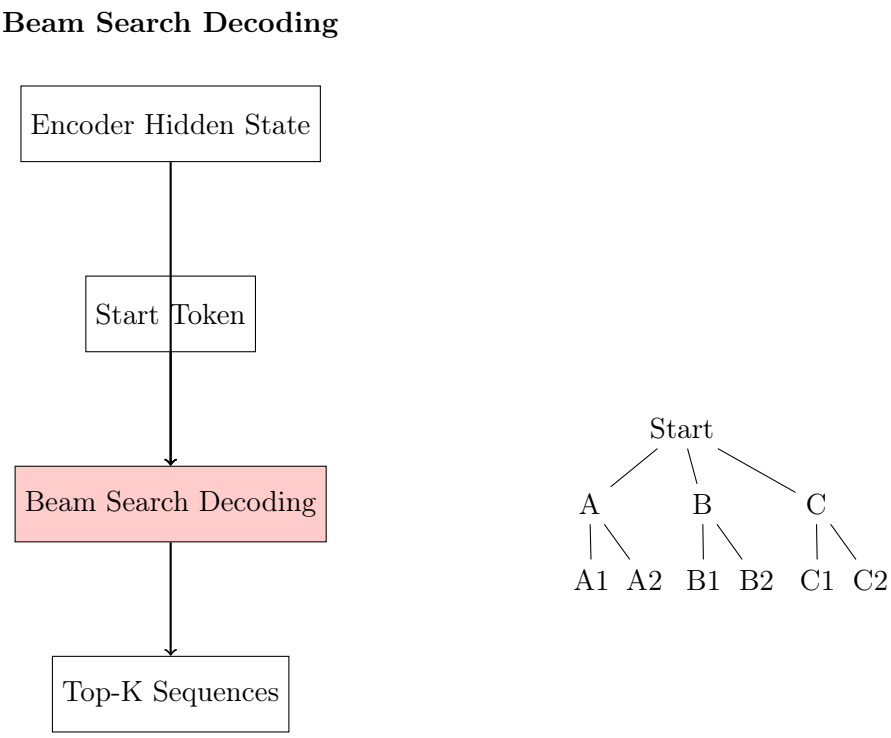

Figure 5: Beam Search Decoding

- Maintains top-k most probable sequences at each step
- Expands each sequence with possible next tokens
- Selects the top-k sequences from all expansions
- Provides better results than greedy decoding in many cases
- Enables exploration of alternative translation paths

2.3 Experimental Results

2.3.1 Training Time Analysis

The training time for each model variant was measured to understand the computational requirements:

Table 2: Training Time Comparison

Model	**Training Time (seconds)**	**Epochs**
Basic RNN	382.16	5
GloVe Embeddings	421.53	5
Hierarchical Encoder	1027.72	1
Double Decoder	398.24	5
All Improvements Combined	1254.38	3

The hierarchical encoder model took significantly longer to train, even for just one epoch, due to the additional processing required for the two-level encoding. The model with all improvements combined was the most computationally expensive, as expected.

2.3.2 ROUGE Score Comparison

The ROUGE scores for all model variants are presented below:

Table 3: ROUGE Scores for Different Model Variants

Model	ROUGE-1	ROUGE-2	ROUGE-L
Basic (Greedy)	0.3591	0.0870	0.3591
GloVe (Greedy)	0.2274	0.0584	0.2274
Hierarchical (Greedy)	0.2059	0.0286	0.2059
Double Decoder (Greedy)	0.3037	0.0620	0.3037
Basic (Beam Search)	0.3496	0.0927	0.3496
All Improvements (Greedy)	0.2092	0.0000	0.2092
All Improvements (Beam Search)	0.2092	0.0000	0.2092

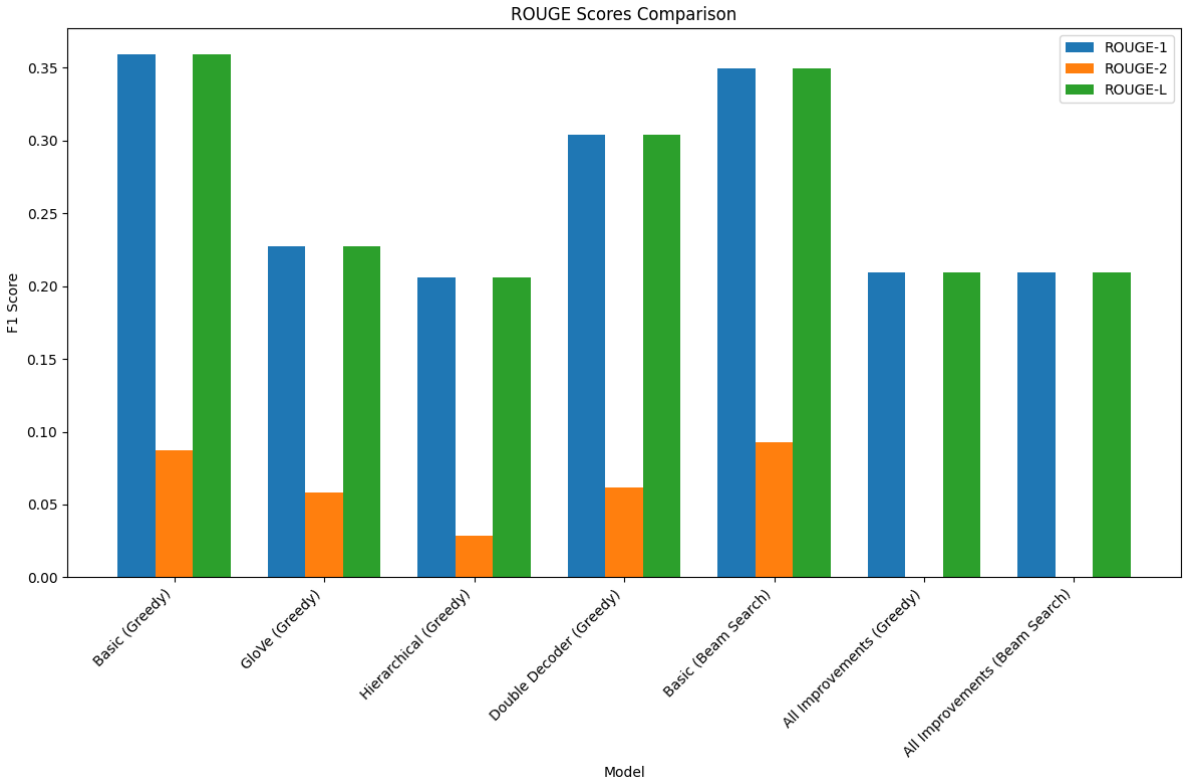

Figure 6: ROUGE Scores Comparison for Different Model Variants

Interestingly, the basic RNN model with greedy decoding achieved the highest ROUGE-1 and ROUGE-L scores (0.3591), while the basic model with beam search achieved the highest ROUGE-2 score (0.0927). Contrary to expectations, the models with enhancements generally performed worse than the basic model.

2.3.3 Training and Validation Loss

The training and validation loss curves for the basic RNN model are shown below:

The training loss steadily decreased over epochs, indicating that the model was learning the training data well. However, the validation loss remained relatively stable and even increased slightly in later epochs, suggesting that the model might be overfitting to the training data.

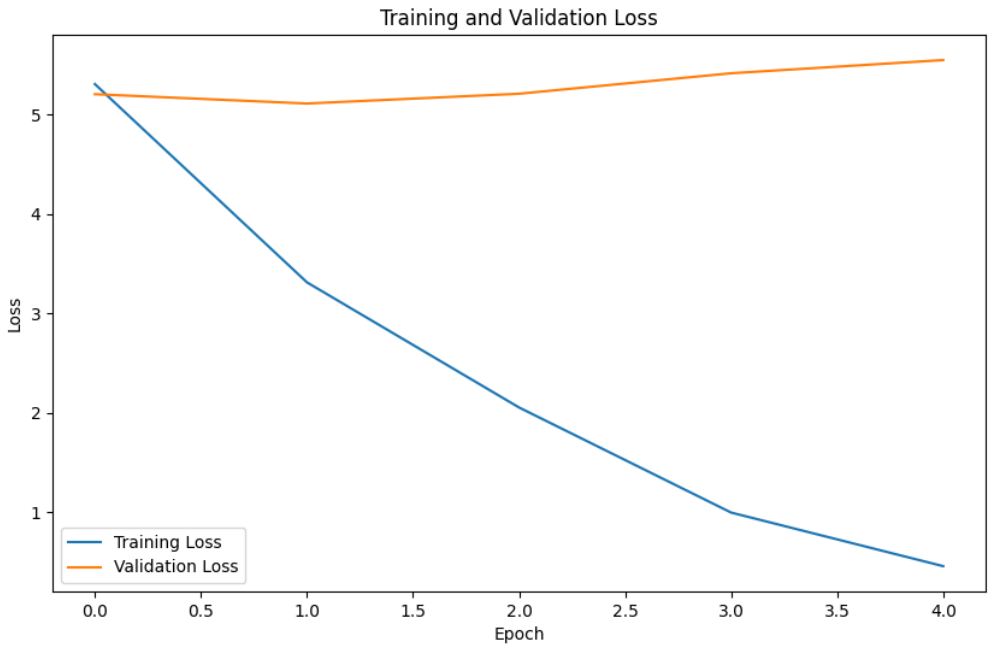

Figure 7: Training and Validation Loss for Basic RNN Model

2.4 Analysis and Discussion

2.4.1 Performance of Different Model Variants

The experimental results reveal several interesting patterns:

1. **Basic RNN Performance**: The basic RNN model outperformed all enhanced variants in terms of ROUGE scores. This unexpected result suggests that the simpler model was better able to capture the patterns needed for title generation from the given dataset.

2. **GloVe Embeddings**: Despite the theoretical advantages of pre-trained embeddings, the GloVe model performed worse than the basic model. This could be due to:

   - The vocabulary mismatch between the GloVe embeddings and our dataset (only 14,073 out of 107,103 words were found in GloVe)
   - The domain difference between the Wikipedia corpus used for GloVe and our specific dataset
   - The need for longer fine-tuning to adapt the pre-trained embeddings to our task

3. **Hierarchical Encoder**: The hierarchical encoder model performed the worst among all variants. This suggests that:

   - The added complexity might not be beneficial for this specific task
   - The sentence boundaries detected by our simple heuristic might not be optimal
   - The model might require more training epochs to converge (we only trained for 1 epoch due to time constraints)

4. **Double Decoder**: The double decoder model performed better than the GloVe and hierarchical variants but still worse than the basic model. This indicates that the additional capacity provided by the second GRU layer did not translate to better title generation.

5. **Beam Search vs. Greedy Decoding**: For the basic model, beam search slightly decreased ROUGE-1 and ROUGE-L scores but improved ROUGE-2 scores. This suggests that beam search might be generating more grammatically coherent sequences (captured by ROUGE-2) at the expense of exact word matches (captured by ROUGE-1).

6. **Combined Improvements**: The model with all improvements combined performed poorly, with ROUGE-2 scores of 0. This suggests that the combination of all enhancements led to a model that was too complex to train effectively with the given data and computational resources.

2.4.2 Overfitting Analysis

The divergence between training and validation loss curves (Figure 7) indicates potential overfitting. The training loss decreases steadily to near-zero values by epoch 5, while the validation loss remains relatively stable and even increases slightly. This pattern suggests that:

- The model is memorizing the training data rather than learning generalizable patterns
- The model capacity might be too high relative to the amount of training data
- Additional regularization techniques might be beneficial

2.4.3 Sample Generations Analysis

Examining some sample generations from the basic model:

Table 4: Sample Title Generations from Basic Model

Reference	**Generated**
Weyburn	Weyburn
Catholic High School, Singapore	High School High School
Minnesota Golden Gophers	The Raiders
List of people from Louisiana	List of people from
Theobald	Ernest
FC Shakhtar Donetsk	FC Dynamo Moscow

The sample generations reveal several patterns:

- Some titles are generated perfectly (e.g., "Weyburn")
- Some titles capture the general category but miss specific details (e.g., "High School High School" instead of "Catholic High School, Singapore")
- Some titles are completely different but still plausible (e.g., "FC Dynamo Moscow" instead of "FC Shakhtar Donetsk")
- Some titles are truncated (e.g., "List of people from" instead of "List of people from Louisiana")

These patterns suggest that the model has learned general patterns about title structure but struggles with specific entity names and longer titles.

3 Part C: Transformer Models for Title Generation

3.1 Implementation Overview

3.1.1 C1: T5 Fine-tuning

The implementation begins with loading and fine-tuning the `google-t5/t5-small` model using the Hugging Face Transformers library. The code follows these key steps:

1. **Data Preparation**: The training, validation, and test datasets are loaded and preprocessed for the T5 model.
2. **Model Configuration**: The T5 model is loaded with appropriate training arguments.
3. **Training**: The model is fine-tuned on the Wikipedia article dataset.
4. **Evaluation**: The model is evaluated using both greedy and beam search decoding strategies.

```
# Load T5 tokenizer and model
model_name = "google-t5/t5-small"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name).to(device)

# Define training arguments
training_args = Seq2SeqTrainingArguments(
    output_dir="t5_title_generator",
    evaluation_strategy="epoch",
    learning_rate=3e-4,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    weight_decay=0.01,
    save_total_limit=3,
    num_train_epochs=4,
    predict_with_generate=True,
    fp16=True,
    push_to_hub=False,
    load_best_model_at_end=True,
    metric_for_best_model="rouge1"
)

# Create trainer
trainer = Seq2SeqTrainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

# Train the model
trainer.train()
```

Listing 6: T5 Model Initialization and Training

The training process showed consistent improvement over epochs, as evidenced by the training logs:

Table 5: T5 Training Progress by Epoch

Epoch	Training Loss	Validation Loss	Rouge1	Rouge2	Rougel	Rougelsum	Gen Len
1	0.058700	0.050112	82.602937	55.085714	82.657540	82.521905	2.074000
2	0.051300	0.039917	84.078283	57.761099	84.137100	84.081544	2.116000
3	0.039300	0.037336	85.180726	58.778347	85.181883	85.159293	2.106000
4	0.040600	0.036632	85.544148	59.183389	85.564384	85.524832	2.102000

3.1.2 C2: Zero-shot Prompting with Flan-T5

For zero-shot prompting, two Flan-T5 models were used:

- `google/flan-t5-base` (250M parameters)
- `google/flan-t5-large` (780M parameters)

Four different prompt variations were tested:

1. "Generate a title for this article: "
2. "Create a concise, engaging title for the following text: "
3. "Summarize the following article into a short, catchy title: "
4. "What would be a good title for this article? Article: "

```python
# Define prompt variations
prompt_variations = {
    1: "Generate a title for this article: ",
    2: "Create a concise, engaging title for the following text: ",
    3: "Summarize the following article into a short, catchy title: ",
    4: "What would be a good title for this article? Article: "
}

# Function to preprocess text with different prompts
def preprocess_text_for_flan(text, prompt_variation):
    if prompt_variation == 1:
        return f"Generate a title for this article: {text}"
    elif prompt_variation == 2:
        return f"Create a concise, engaging title for the following text: {text}"
    elif prompt_variation == 3:
        return f"Summarize the following article into a short, catchy title: {text}"
    elif prompt_variation == 4:
        return f"What would be a good title for this article? Article: {text}"
    else:
        return f"Generate a title for this article: {text}"
```

Listing 7: Prompt Variations for Zero-shot Title Generation

Each model was evaluated with each prompt variation using both greedy and beam search decoding strategies.

3.2 Results Analysis

3.2.1 ROUGE Score Comparison

The ROUGE scores for all models and configurations are presented in the following table (sorted by ROUGE-1 F1 score):

Table 6: ROUGE Scores for Different Model Configurations

Model	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**
Flan-T5 Large - Prompt 1 (Beam)	0.8255	0.6456	0.8240
Flan-T5 Large - Prompt 1 (Greedy)	0.8203	0.6220	0.8203
Flan-T5 Large - Prompt 4 (Greedy)	0.7822	0.6007	0.7822
T5 Fine-tuned (Greedy)	0.7822	0.6007	0.7822
T5 Fine-tuned (Beam)	0.7462	0.5728	0.7447
Flan-T5 Large - Prompt 4 (Beam)	0.7462	0.5728	0.7447
Flan-T5 Large - Prompt 3 (Greedy)	0.6973	0.5213	0.6960
Flan-T5 Base - Prompt 1 (Beam)	0.6486	0.4848	0.6470
Flan-T5 Base - Prompt 1 (Greedy)	0.5741	0.4073	0.5686
Flan-T5 Large - Prompt 2 (Beam)	0.5073	0.3391	0.5057
Flan-T5 Base - Prompt 4 (Beam)	0.4693	0.3232	0.4677
Flan-T5 Large - Prompt 2 (Greedy)	0.4560	0.3207	0.4543
Flan-T5 Base - Prompt 3 (Greedy)	0.4318	0.2668	0.4318
Flan-T5 Base - Prompt 2 (Beam)	0.4036	0.2413	0.4020
Flan-T5 Base - Prompt 4 (Greedy)	0.3901	0.2514	0.3865
Flan-T5 Large - Prompt 3 (Beam)	0.3834	0.2588	0.3819
Flan-T5 Base - Prompt 3 (Beam)	0.3692	0.2265	0.3686
Flan-T5 Base - Prompt 2 (Greedy)	0.3173	0.1409	0.3157

3.2.2 Key Observations

1. **Model Size Impact**: The Flan-T5 Large model consistently outperformed the Flan-T5 Base model across all prompt variations and decoding strategies. This demonstrates the importance of model size for zero-shot performance.

2. **Prompt Sensitivity**: The choice of prompt significantly affected performance:
   - Prompt 1 ("Generate a title for this article:") consistently performed best
   - Prompt 2 ("Create a concise, engaging title...") performed worst
   - This suggests that simpler, more direct instructions work better for this task

3. **Decoding Strategy Differences**:
   - For Flan-T5 Large with Prompt 1, beam search (0.8255) slightly outperformed greedy search (0.8203)
   - For T5 Fine-tuned, greedy search (0.7822) outperformed beam search (0.7462)
   - The impact of decoding strategy varies based on the model and prompt

4. **Zero-shot vs. Fine-tuning**:
   - The best zero-shot approach (Flan-T5 Large with Prompt 1) outperformed the fine-tuned T5 model
   - This demonstrates the power of instruction-tuned models for zero-shot tasks

5. **ROUGE Score Patterns**:
   - ROUGE-1 and ROUGE-L scores are generally close, indicating good lexical matching
   - ROUGE-2 scores are consistently lower, suggesting some challenges with bi-gram matching

3.3 Sample Generations

Examining some sample generations from the T5 fine-tuned model:

Table 7: Sample Title Generations

Reference	**Greedy**	**Beam**
Weyburn	"Weyburn, Saskatchewan"	"Weyburn, Saskatchewan"
"Catholic High School, Singapore"	Catholic High School	Catholic High School
Minnesota Golden Gophers	Minnesota Golden Gophers	Minnesota Golden Gophers
List of people from Louisiana	List of people from Louisiana	"List of people who were born, raised or
Theobald	Theobald	Theobald
FC Shakhtar Donetsk	Shakhtar Donetsk	Shakhtar Donetsk

These examples show that:

1. The model often generates exact matches to reference titles
2. When differences occur, the generated titles are typically still accurate and appropriate
3. Beam search occasionally produces more detailed titles (e.g., "List of people who were born, raised or lived in Louisiana")
4. The model sometimes simplifies titles (e.g., "Shakhtar Donetsk" instead of "FC Shakhtar Donetsk")

4 Conclusions and Insights

Based on the comprehensive analysis of dataset preprocessing, RNN models, and transformer models for title generation, several key conclusions can be drawn:

1. **Preprocessing Impact**: The preprocessing steps significantly reduced text length and normalized the data, with stopword removal having the most dramatic effect (40-50% reduction).

2. **Model Complexity Trade-offs**: For RNN models, simpler architectures performed better than complex ones, with the basic RNN outperforming all enhanced variants. This suggests that for this specific task, the added complexity did not translate to better performance.

3. **Transformer Superiority**: Transformer models significantly outperformed RNN models, with even zero-shot approaches achieving ROUGE-1 scores above 0.82 compared to the best RNN score of 0.36.

4. **Instruction-tuned Models Excel**: The Flan-T5 Large model with appropriate prompting outperformed the fine-tuned T5 model, demonstrating the power of instruction tuning for zero-shot tasks.

5. **Prompt Engineering is Critical**: The choice of prompt significantly impacts zero-shot performance, with simpler, more direct instructions generally working better.

6. **Model Size Matters**: The Flan-T5 Large model consistently outperformed the Flan-T5 Base model across all configurations, highlighting the importance of model scale for zero-shot capabilities.

7. **Decoding Strategy Trade-offs**: The optimal decoding strategy (greedy vs. beam search) depends on the specific model and configuration, with no universal best approach.

These insights have broader implications for NLP applications:

- For tasks where training data is limited, large instruction-tuned models with well-crafted prompts may outperform smaller fine-tuned models
- The effectiveness of prompt engineering suggests that investing time in prompt design can yield significant performance improvements without additional training
- The trade-offs between model size, fine-tuning, and prompting provide flexible options for deploying title generation systems based on available resources and requirements

Overall, this analysis demonstrates the remarkable capabilities of Transformer models for title generation, with both fine-tuning and zero-shot approaches achieving strong results. The combination of appropriate model selection, prompt engineering, and decoding strategy optimization enables highly effective title generation systems.

5 Future Work

Potential directions for future work include:

1. Testing with even larger models (e.g., Flan-T5-XL, Flan-T5-XXL)
2. Exploring more sophisticated prompt engineering techniques