The George Washington University

Final Project (Individual) Report

DATS6312

Natural Language Processing

https://github.com/Keerthana0620/Final-Project-Group4/tree/main/ApoorvaReddy-Bagepalli-individual-project

Apoorva Reddy Bagepalli

Dr. Amir Jafari

THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

Introduction

Our project focuses on exploring multiple text generation models, including classical methods like the Markov Chain and advanced deep learning approaches such as Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), and transformer-based models like BART. These models will be trained and fine-tuned on a dataset of recent news articles, sourced from publicly available collections such as Kaggle's InShorts, BBC articles, and the CNN/DailyMail dataset.

The project will be divided into key tasks, including data preprocessing, model training, and evaluation. Each member of the team will be responsible for specific aspects of the project, with an emphasis on different models and techniques. My individual contribution focuses on the implementation of the Markov Chain generator for text generation. This involves simulating a random walk through a sequence of words, where the probability of each word depends on the previous one, and refining the model to generate meaningful and contextually accurate text.

In the following sections, I will discuss the methodology, model architecture, and tasks I have worked on, along with the evaluation metrics used to assess the performance of the text generation models.

Description of the work

As part of this project, my individual contribution focuses on implementing the Markov Chain model for text generation. A Markov Chain is a statistical model that predicts the likelihood of transitioning from one state to another based solely on the current state. This property, known as the Markov Property, simplifies modeling by assuming that the probability of the next state depends only on the present state, not on the sequence of prior states.

Background of the Algorithm

The Markov Chain is defined by:

1. **States:** Represent the entities being modeled. For text generation, each state corresponds to a word or an *n-gram* (sequence of n words).
2. **Transitions:** These are the probabilities of moving from one state to another, derived from the observed frequencies in the dataset.

Transition Probability Equation

If P_{ij} represents the transition probability from state i to state j, it is calculated as:

$$P_{ij} = \frac{\text{Number of transitions from state } i \text{ to state } j}{\text{Total number of transitions from state } i}$$

Here, P_{ij} satisfies the condition:

$$\sum_j P_{ij} = 1$$

Higher-Order Markov Chains

To incorporate context, we use *n-gram* Markov Chains where each state represents a sequence of nnn words. The transition probabilities are then calculated for these sequences instead of individual words, providing more coherent and meaningful text generation.

Stationary Distribution

A Markov Chain reaches a stationary distribution if the transition probabilities stabilize over time, meaning the likelihood of being in any state becomes constant. This property can help ensure diverse and unbiased text generation.

Development of the Algorithm

1. **Defining States and Transitions**
   - Words and n-grams were extracted from the dataset, forming the states of the Markov Chain.
   - Transition probabilities were computed by traversing the dataset, counting occurrences of word pairs or n-grams, and normalizing the counts.
2. **Text Generation Process**
   - To generate new text, a random walk is simulated on the Markov Chain. Starting from an initial state, the algorithm probabilistically selects the next state based on transition probabilities and continues this process until a stopping criterion (e.g., sentence length) is met.
3. **Implementation Steps**
   - Tokenization: Preprocessed the text to divide it into words or n-grams.
   - Transition Matrix Construction: Created a matrix representing transition probabilities between states.
   - Random Walk Simulation: Generated text by traversing the states based on computed probabilities.

Chapman-Kolmogorov Equation

The Chapman-Kolmogorov theorem is used to calculate the probability of transitioning between states over multiple steps:

$$P_{ij}^{(n)} = \sum_k P_{ik}^{(n-1)} . P_{kj}$$

Here, $P_{ij}^{(n)}$ is the probability of transitioning from state i to state j in n steps.

Portion of my work

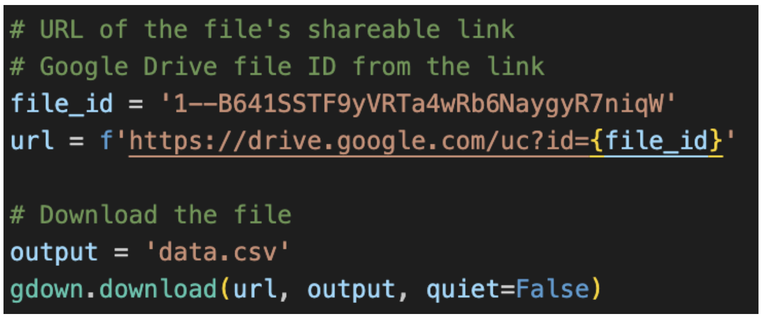
1. Preprocessing

Purpose:

Prepare raw text data for processing and modeling by removing noise, ensuring consistency, and enhancing quality.

Key Steps:

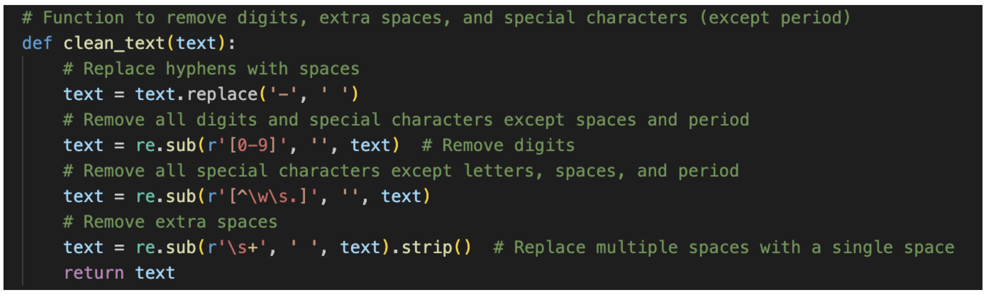
1. **Downloading Dataset**:
   - Used gdown to fetch the dataset directly from Google Drive.
   - Ensured easy and reproducible access to the dataset.



```python
# URL of the file's shareable link
# Google Drive file ID from the link
file_id = '1--B641SSTF9yVRTa4wRb6NaygyR7niqW'
url = f'https://drive.google.com/uc?id={file_id}'

# Download the file
output = 'data.csv'
gdown.download(url, output, quiet=False)
```

2. **Text Cleaning**:
   - Removed digits, special characters (except periods), and excess spaces.
   - Replaced hyphens with spaces for better tokenization.
   - Applied cleaning logic to the Article column and created a new column, cleaned_text.



```python
# Function to remove digits, extra spaces, and special characters (except period)
def clean_text(text):
    # Replace hyphens with spaces
    text = text.replace('-', ' ')
    # Remove all digits and special characters except spaces and period
    text = re.sub(r'[0-9]', '', text)  # Remove digits
    # Remove all special characters except letters, spaces, and period
    text = re.sub(r'[^\w\s.]', '', text)
    # Remove extra spaces
    text = re.sub(r'\s+', ' ', text).strip()  # Replace multiple spaces with a single space
    return text
```

3. **Exporting Cleaned Data**:
    - Dropped unnecessary columns and saved the cleaned data for further processing.

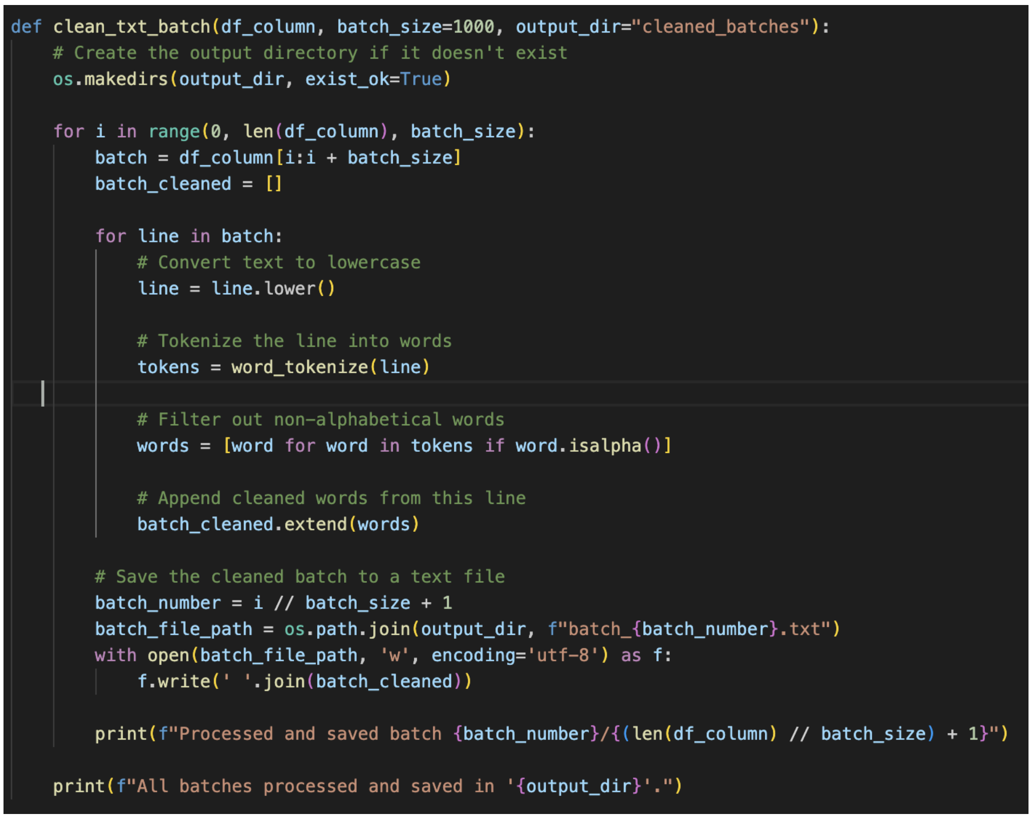
2. Batch Processing

Purpose:

Divide the dataset into manageable chunks, enabling scalable and efficient processing for large datasets.

Key Steps:

1. **Batch Cleaning Function**:
    - Processed text data in chunks (default batch size: 1,000 rows).
    - Performed text cleaning (lowercasing, tokenization, and filtering out non-alphabetic tokens) on each batch.
    - Saved processed text batches as individual files for model training.



```python
def clean_txt_batch(df_column, batch_size=1000, output_dir="cleaned_batches"):
    # Create the output directory if it doesn't exist
    os.makedirs(output_dir, exist_ok=True)

    for i in range(0, len(df_column), batch_size):
        batch = df_column[i:i + batch_size]
        batch_cleaned = []

        for line in batch:
            # Convert text to lowercase
            line = line.lower()

            # Tokenize the line into words
            tokens = word_tokenize(line)

            # Filter out non-alphabetical words
            words = [word for word in tokens if word.isalpha()]

            # Append cleaned words from this line
            batch_cleaned.extend(words)

        # Save the cleaned batch to a text file
        batch_number = i // batch_size + 1
        batch_file_path = os.path.join(output_dir, f"batch_{batch_number}.txt")
        with open(batch_file_path, 'w', encoding='utf-8') as f:
            f.write(' '.join(batch_cleaned))

        print(f"Processed and saved batch {batch_number}/{(len(df_column) // batch_size) + 1}")

    print(f"All batches processed and saved in '{output_dir}'.")
```

2. **Results:**
    - Created cleaned, tokenized text files for each batch, ready for model training.
    - Ensured memory efficiency by avoiding processing the entire dataset at once.

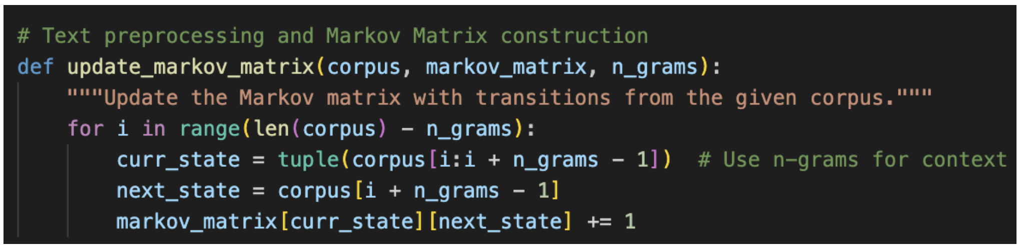
3. Higher-Order Markov Chain Implementation

Purpose:

Generate text based on patterns learned from sequences of tokens in the dataset using configurable n gram.

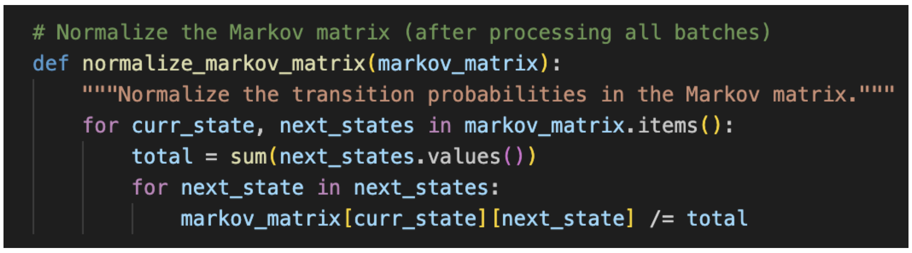
Key Steps:

1. **Markov Matrix Construction**:
   - Constructed a transition matrix to capture the frequency of token sequences (n_grams).
   - Allowed flexibility in the context size (n_grams configurable, e.g., trigrams).



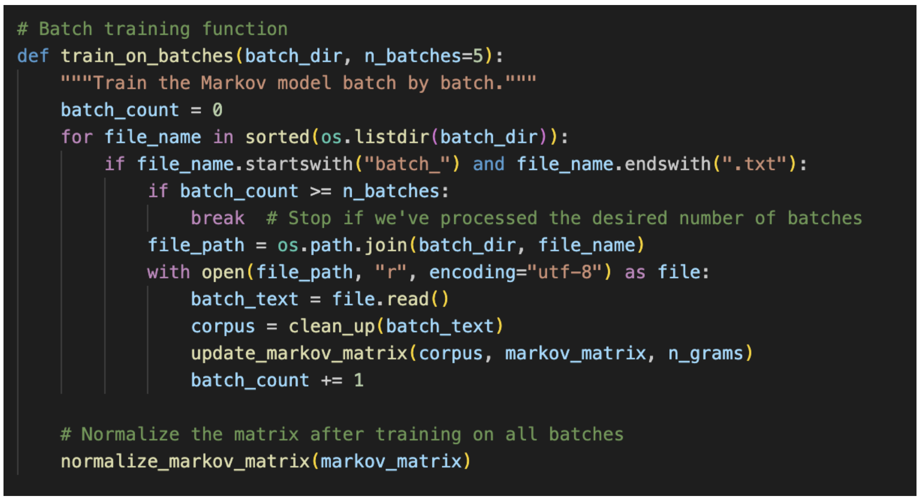
```python
# Text preprocessing and Markov Matrix construction
def update_markov_matrix(corpus, markov_matrix, n_grams):
    """Update the Markov matrix with transitions from the given corpus."""
    for i in range(len(corpus) - n_grams):
        curr_state = tuple(corpus[i:i + n_grams - 1])  # Use n-grams for context
        next_state = corpus[i + n_grams - 1]
        markov_matrix[curr_state][next_state] += 1
```

2. **Smoothing and Normalization**:
   - Smoothed probabilities to handle unseen transitions and avoid zero-probability states.
   - Normalized the transition matrix to calculate probabilities.



```python
# Normalize the Markov matrix (after processing all batches)
def normalize_markov_matrix(markov_matrix):
    """Normalize the transition probabilities in the Markov matrix."""
    for curr_state, next_states in markov_matrix.items():
        total = sum(next_states.values())
        for next_state in next_states:
            markov_matrix[curr_state][next_state] /= total
```

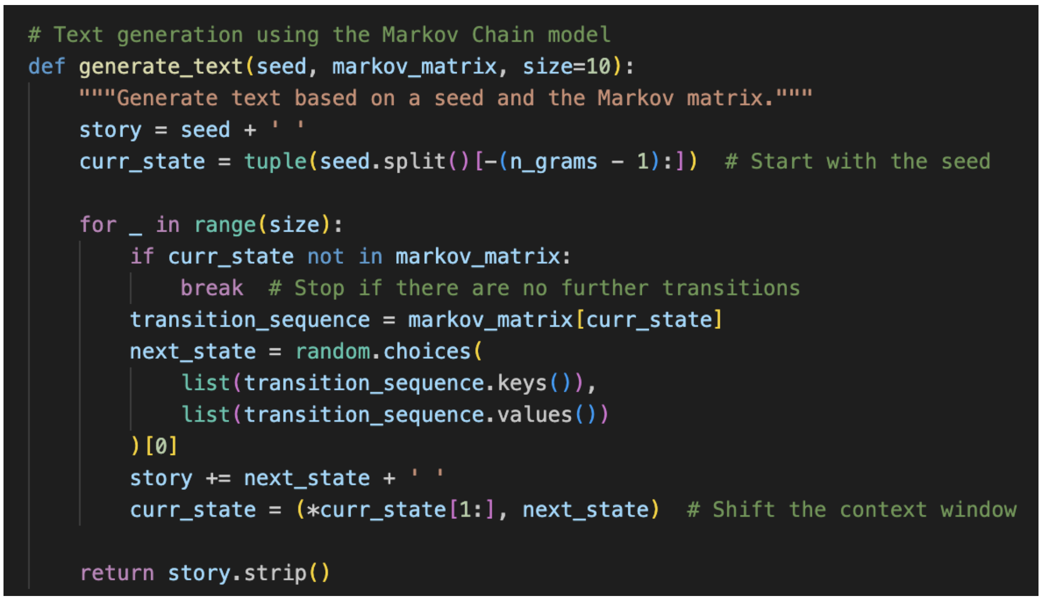
3. **Training on Batches**:
   - Iteratively updated the Markov matrix using cleaned text batches.



```python
# Batch training function
def train_on_batches(batch_dir, n_batches=5):
    """Train the Markov model batch by batch."""
    batch_count = 0
    for file_name in sorted(os.listdir(batch_dir)):
        if file_name.startswith("batch_") and file_name.endswith(".txt"):
            if batch_count >= n_batches:
                break  # Stop if we've processed the desired number of batches
            file_path = os.path.join(batch_dir, file_name)
            with open(file_path, "r", encoding="utf-8") as file:
                batch_text = file.read()
                corpus = clean_up(batch_text)
                update_markov_matrix(corpus, markov_matrix, n_grams)
                batch_count += 1

    # Normalize the matrix after training on all batches
    normalize_markov_matrix(markov_matrix)
```

4. **Text Generation**:
   - Generated sequences based on learned patterns, starting from a seed phrase.



```python
# Text generation using the Markov Chain model
def generate_text(seed, markov_matrix, size=10):
    """Generate text based on a seed and the Markov matrix."""
    story = seed + ' '
    curr_state = tuple(seed.split()[-(n_grams - 1):])  # Start with the seed

    for _ in range(size):
        if curr_state not in markov_matrix:
            break  # Stop if there are no further transitions
        transition_sequence = markov_matrix[curr_state]
        next_state = random.choices(
            list(transition_sequence.keys()),
            list(transition_sequence.values())
        )[0]
        story += next_state + ' '
        curr_state = (*curr_state[1:], next_state)  # Shift the context window

    return story.strip()
```

Result

Base Model

- **Perplexity**: Extremely high (10 billion), indicating poor predictive ability.
- **ROUGE-1 F1 Score**: 1.88e-6, with precision at 1.0 but negligible recall.
- **Discussion**: The model failed to capture meaningful patterns in the data, reflected in its inability to generate coherent or relevant text.

High-Order Gram Model

- **Perplexity**: Slightly improved (9.99 billion) compared to the Base Model.
- **ROUGE-1 F1 Score**: Same as the Base Model (1.88e-6), with no improvement in recall.
- **Discussion**: The higher-order context captured more complex patterns, but this did not translate into better performance due to overfitting or lack of sufficient smoothing.

POS-Tagging Markov Model

- **Perplexity**: Matches the Base Model (10 billion).
- **ROUGE-1 F1 Score**: 8.77e-7, slightly worse than the Base and High-Order Models.
- **Discussion**: Introducing part-of-speech constraints reduced the diversity of transitions, negatively impacting the overall output quality.

Markov Model with Temperature Rating

- **Perplexity**: Dramatically reduced to 22.16, indicating far superior performance.
- **ROUGE-1 F1 Score**: Matches the Base Model and High-Order Model (1.88e-6).
- **Discussion**: The use of temperature scaling improved token sampling, reducing uncertainty. However, this did not yield measurable improvements in BLEU or ROUGE scores, suggesting further tuning is required.

Key Observations

- **Temperature scaling** significantly improved the perplexity but did not enhance BLEU or ROUGE scores.
- **POS tagging** reduced the model's flexibility, leading to slightly lower performance metrics.
- The high perplexity of the **Base** and **High-Order Models** suggests a need for better smoothing or feature representation.

Key Findings

1. **Base Model**:
   - Extremely high perplexity (~10 billion) indicates poor predictive capability.
   - The BLEU and ROUGE scores were negligible, with precision at 1.0 but recall too low to generate meaningful outputs.
   - The model struggled to capture basic patterns in the dataset due to insufficient feature representation.
2. **High-Order Gram Model**:
   - Slight improvement in perplexity compared to the Base Model.
   - Performance metrics (ROUGE and BLEU) remained unchanged, implying limited gains from adding context with higher-order n-grams.
   - Overfitting to specific patterns and lack of smoothing techniques likely hindered performance.
3. **POS-Tagging Markov Model**:
   - Performance metrics were slightly worse than the Base Model.
   - The rigidity introduced by part-of-speech constraints reduced model diversity and flexibility, limiting its effectiveness in generating text.
4. **Markov Model with Temperature Scaling**:
   - Significant reduction in perplexity to 22.16, showing a substantial improvement in predictive capability.
   - Despite this, BLEU and ROUGE scores did not improve, suggesting that temperature scaling enhanced sampling diversity but failed to address deeper structural issues in the model's understanding of context.

Lessons Learned

- **Simplicity of Markov Models**: While Markov Chains are computationally efficient and intuitive, their limited context modeling capabilities make them unsuitable for generating complex or meaningful text.
- **Impact of Smoothing and Scaling**: Techniques like temperature scaling can improve diversity in text generation but need to be complemented by better representation of data features to enhance overall output quality.
- **Importance of Context**: Higher-order models attempted to capture context but suffered from overfitting, highlighting the need for advanced techniques like smoothing or regularization.

Proposed Improvements

1. **Enhanced Preprocessing**:
   - Incorporate lemmatization and stopword removal to refine the tokenization process.
   - Experiment with larger and more diverse datasets to reduce overfitting.
2. **Smoothing Techniques**:
   - Implement techniques like Laplace or Kneser-Ney smoothing to handle sparse transition probabilities more effectively.
3. **Incorporating Neural Models**:
   - Combine Markov Chains with neural models (e.g., RNNs or transformers) to leverage their strengths in modeling long-term dependencies and context.
4. **Evaluation Metrics**:
   - Expand the evaluation to include qualitative assessments of generated text (e.g., coherence, diversity) in addition to BLEU and ROUGE scores.
5. **Future Directions**:
   - Explore hybrid approaches, such as Markov Chains with embeddings or pre-trained models, to strike a balance between simplicity and contextual understanding.
   - Experiment with adaptive temperature scaling to dynamically adjust sampling diversity based on output quality.

Conclusion

This project focused on implementing Markov Chain-based text generation models to explore their ability to generate coherent and meaningful text. Despite the simplicity and efficiency of the Markov Chain, the models struggled to produce high-quality outputs, as reflected in the high perplexity and low ROUGE and BLEU scores. While higher-order n-grams and techniques like temperature scaling showed some improvements, challenges like overfitting and insufficient feature representation persisted. Key lessons from the project highlight the limitations of Markov Chains in modeling complex contexts and the need for advanced smoothing techniques and neural network integration to improve text generation. Future work will explore hybrid approaches combining the strengths of Markov Chains with more advanced models to enhance both diversity and contextual accuracy.

Code Calculation

Lines of code taken from Internet : 124

Lines of code modified : 63

Lines of code developed myself : 188

Percentage of code referred from internet : 19.55%

References

1. Code, R.Markov chain text generator - Rosetta Code. Rosetta Code. https://rosettacode.org/wiki/Markov_chain_text_generator#Python
2. Generating Text With Markov Chains. (2021, January 1). https://healeycodes.com/generating-text-with-markov-chains