**DTE-2501 AI og Algoritmer**

**Graded Assignment 1**

**Jon-Fredrik Hopland**

**Introduction (Task 1)**

This assignment requires us to create a generative language model using Markov chains, a technique used in Natural Language Processing (NLP). Markov chains use the probability of one word following another to create sentences that make sense based on the data it's trained on.

I intend to gather a dataset on a specific topic, preprocess the data to make it suitable for training, and then employ a Markov chain approach to model word sequences. Once the model is trained, it can generate new coherent text sequences in the topic of the trained data.

**Preprocessing Text (Task 2)**

I chose to create a “Book title generator” for this assignment. I sourced the book titles from the websites Imagine Forest (<https://www.imagineforest.com/blog/book-title-generator/>) and Bucolicaholic (<https://bucolicaholic.com/2017/10/29/reading-list-top-100-american-literature-titles/>). I stored the text in a .txt file to facilitate preprocessing.

The main goal of preprocessing in this project was to structure the raw lyrical data in a way that makes it suitable for the model training. The preprocessing steps were taken to clean and standardize the data. The preprocessed text was then stored in a new file called ‘preprocessed.txt’.  
  
**The processing steps:**

* Lower casing
* Punctual removal
* Stopwords removal
* Frequent words removal
* Rare words removal
* Emojis and emoticons removal
* Urls removal
* Empty strings removal

**Training the Generator (Task 3)**

In Task 3, we focused on using the preprocessed text to generate a weighted graph. This graph helps us see patterns in the original text . By understanding these patterns, we can make new text that match the style of the original text . This step is important because it makes sure the new text fit well with the original.

**Reading and Parsing Function**

I created a reading and parsing function that returns the preprocessed text . The function also creates a new file containing the preprocessed text. I could use that file to load the preprocessed text. However, for this assignment, I chose to just return the preprocessed text from the read\_and\_parse function.

**Construct Weighted Graph Function**

This function processes a list of sentences. As the function goes through each sentence, it identifies pairs of consecutive words and adds an entry for them into the graph. We use the "#" symbol to tell the beginning of a sentence. Following this, the count of the subsequent word is incremented. As for the remaining words within the sentence, the function increments the count of the subsequent word relative to the current word. At the end of its process, the function provides a detailed graph showing word pairs and how often they appear together.

**The "WordState" helper class**

This class facilitates the creation and handling of the graph. It uses a dictionary, \_next\_words, to keep a record of subsequent potential words alongside their frequencies. The add\_next\_word method either introduces a word or increases its existing count. To determine the availability of subsequent words, the has\_next method is employed. Furthermore, the get\_next method leverages the rand.choices function to get a random next word, basing its selection on the recorded frequencies.

**Generating Random Book Titles (Task 4)**

In Task 4, we leveraged the code we developed in the WordState class in Task 3 to create random book titles. The function, generate\_random\_sentence, starts with the special character # and moves through the graph, picking subsequent words based on their weights in the constructed graph. Each time a word is chosen, the current\_word is updated, guiding the selection of the next word. This makes the generated sentences follow the structure and transitions of the original graph, providing a balance between randomness and consistency.

The generate\_sentences function calls the generate\_random\_sentence function multiple times to generate a list of random sentences. Using this method, we get different titles that still follow the patterns from the original data.

**The User Interface (Task 5)**

**Creating a Simple UI to Interact with the Generator**

To make the Markov text generator more user-friendly and interactive, I built a web-based interface using Flask. This UI offers a clean environment for the user to generate random sentences based on the Markov chain model.

When users visit the main page, they see an interface from the "index.html" page where they can specify how many sentences they want to generate and then click on the "Generate" button. The application then refers to pre-processed text data, which is stored in a global variable named "graph", and then shows the generated sentences to the users.

**Saving Generated Text in a File**

This function allows users to save the generated sentences to a file directly from the web interface. For the purpose of this assignment, the file is just stored locally in the project forlder. I added a "Save to File" button to the interface. When users click this button, the application takes the generated sentences and saves them in a file called "generated\_text.txt". After the saving process, users receive a message confirming that their text was successfully saved in the mentioned file.

**Accepting Multiple Training Data Sources**

To make the tool versatile, the capability to train the generator on various datasets was added. Users can upload multiple text files, and the system retrains the model based on this new data. I incorporated an "Upload and Retrain" button on the interface for users to upload several text files. After a user has clicked on the “Choose Files” button and chosen their files, the user can then click on "Upload and Retrain". Once clicked, it merges the contents of these files, processes them, and updates the generator with this newly processed data.