# “An Exploration Into the Feasibility of Markov Chains Applied to Formal Grammars as a Method of Natural Language Generation”

Submitted April 2024, in partial fulfilment of

the conditions for the award of the degree **BSc Computer Science.**


**20331452**

School of Computer Science

University of Nottingham


I hereby declare that this dissertation is all my own work, except as indicated in the text:

14,850 words

**Signed:** **D.H.**

**Date:** **19 / 04 / 2024**

## Abstract:

Natural Language Generation (NLG) is a computationally expensive task that can limit the range of potential use cases due to lack of available computing power or cost. This research explores the use of word usage statistics and sentence structure when applied to natural language generation. An experiment in which users were tasked with selecting real quotes from an array of real and generated quotes from a certain novel. The responses showed that using statistics applied to structure is an insufficient method of coherent NLG. There must be other layers to the solution, and the tracking of other contexts may be required further in development of a NLG algorithm that can run on a low power machine. The research outlines potential steps that could be taken to extend the developed algorithm.


## Table of Contents:

# 1. Introduction

Natural Language Processing (NLP) is an area of computing concerned with the generation and understanding of language. In recent years, NLP has been used increasingly in accompaniment with Artificial Intelligence (AI), most recognisably with applications such as ChatGPT (OpenAI, 2023). NLP is a powerful tool that has the potential to be used in virtual assistant bots, automatic authoring or for generating immersive non-player characters in video games. However, in order to produce a unique deep-learning based NLP model, a large investment of both money and time is required. To train a single model, it takes approximately 120 hours of processing time and costs around $175 (Strubell et al., 2020). For a single model, this cost isn't prohibitive but to train multiple models the cost could become too large to be practical. For substantial models, this cost can be even greater. For example, the GPT-3 Model costs billions of dollars to train each iteration (Brown et al., 2020). This severely limits the potential of smaller use cases that could require a specifically trained Large Language Model (LLM).

Natural Language Generation (NLG), a component of NLP concerned with the production of text, can be outsourced to applications such as “ChatGPT” by OpenAI or “Gemini” by Google, in that these applications will utilise the Application Programming Interface (API) for the respective model to perform the powerful computation. API calls to these solutions can grow to be expensive, as they charge per token and a request could contain thousands of tokens (*OpenAI Platform*, n.d.). There exist only a small number of specific use models for developer use. An algorithm that could be implemented on a user’s local machine (removing the need for any API calls over the internet) could reduce the cost. Implementing a model locally may even speed up response times by removing the factors of potentially slow internet speeds as well as the time that these models take to produce text. Layers of interconnected nodes in a model such as GPT-4 would have a large computational footprint (Meghani, 2023) that would prohibit running programs locally on machines with less resources. It is assumed that due to the limitations of smaller machines and the lowered computing power, that text will either take longer to generate or will be of lower quality.

It is important that algorithms for NLG produce text that is both syntactically and contextually correct. Otherwise, they are incomprehensible. For example, Noam Chomsky writes that “the notion “grammatical” cannot be identified with “meaningful” or “significant” in any semantic sense”, citing the idea that the sentence “Colorless green ideas sleep furiously.” is a syntactically correct sentence and obeys the rules of grammar, but is nonsensical (1957). Recurrent grammars are a way to produce such potentially nonsensical text, in that they generate a word to fit a gap from a set of possible words. This works well for specific templates where the aim is to generate text in a given form by swapping complete components of sentences. (Compton et al,. 2015).

Only finite state languages are able to be generated by recurrent grammar. A finite state language is "finite or infinite set of strings (sentences) of symbols (words) generated by a finite set of rules (the grammar), where each rule specifies the state of the system in which it can be applied, the symbol which is generated, and the state of the system after the rule is applied.” (Chomsky et al., 1958). English is not a finite state language, and so whilst it is impossible to create a grammar that would generate the complete

set of sentences in English that both make sense and are grammatically correct, we can certainly generate a subset of these sentences that are at least grammatically correct.

## 2. Motivation

Situations such as a small group of people playing Tabletop Role-Playing Games (TTRPGs) like Dungeons and Dragons (D&D), needing to generate character speech for a video game or wanting to randomise a description for a creative writing generator all illustrate the need for a smaller, less generalised language model like this as players may need to generate descriptions or plots in a time critical sense. Due to the structure of D&D allowing players to do essentially anything (so long as it is feasible in the world of the game) the players may take actions for which the Dungeon Master, who acts as the storyteller and guide, has not prepared and therefore must quickly produce an idea on the spot. Similarly, players could want to quickly generate a random character backstory for themselves or a Non-Player Character. Performing these tasks using a web app like ChatGPT can be a tentative solution, but often requires time to explain and set up the surroundings. Similarly, a trained instance of a model such as GPT also takes time to set up, but then may grow to be expensive as it is used due to the large number of tokens in each request. GPT is constantly advancing, backed by billions in investment, and so over the course of the writing of this paper, the project has developed the ability to read information to and from PDF files. This would help it generate prompts specific to a campaign.

Role Playing Games (RPGs) are not the only use case that would benefit from procedural NLG. For example, open world video games with lots of characters might want to randomise the things they say when interacted with. It may also be beneficial to generate large amounts of text for inspiration for a creative project or even replicating the work of another. Models like GPT and Gemini would be sufficient in this use case, but once again, come with drawbacks when implemented into an application.

A program that can run on a user's local machine could save the user both time and money on API calls, and even speed up response times should such a system be quick enough. The size of a model like GPT-4 would be prohibitive of this kind of computation and so there is evidence to suggest that it is worth pursuing a more suitable method of NLG. It is fairly quick to read information from a PDF and extract it to a text file and so data extraction would not be an issue for a low power machine.Therefore, the aim is to produce an algorithm that can run on a wide number of machines with low computational power, and high quality and efficiency. Both quality and efficiency is particularly important for applications like TTRPGs where quick generation of descriptions is crucial.

When discussing LLMs, Menghani states “Although these models perform well on the tasks they are trained on, they might not necessarily be efficient enough for direct deployment in the real world” (2023). Models excel at a specific purpose, and so creating a general model for all sorts of text would be expensive, as it is effectively retraining a model to perform different tasks. However, if the model is able to read a specific example of text, or a mass of texts, and be trained to use this information to write new text, then this could be considered general purpose. Of course, to truly extend the definition, the algorithm would be able to take in a prompt that relates to a question or task it would then perform or answer. The

algorithm to be created shall follow a simplified procedure compared to this more rigorous definition, to ensure sufficient time for testing and improvement

The aim of the project is to create an algorithm that is able to be implemented on a low power, local machine that is able to produce text by learning the patterns of word placement in a sentence and apply that to grammatically correct sentence structure. It is my hypothesis that this will be an effective way of generating natural language as through analysing the relations between neighbouring words, the implicit context tracking will minimise the chance of the wrong form of a word being used, like in the example of plurality.

# 3. Related Work

## 3.1 Chomskyan Phrase Structure

In Chomsky's work "Syntactic Structures" (1957), he considers a language to be a set of sentences, each finite in length and constructed from a finite set of elements. By this definition we can understand that all possible sentences, grammatical or not, are valid. He further states that the "fundamental aim" of linguistic analysis of some language, L is to separate the grammatical sentences from the ungrammatical sentences. This means that in order to produce natural language, we must find a way to generate only grammatical sentences. Kelmendi writes that there are only certain ways for a sentence to be structured from constituent parts (Kelmendi, 2020), in that you cannot take sentence components and put them together in any order. From this, we can understand that there exists a finite set of grammatical sentence structures and that the items in these sets can be thought of as lists of constituent parts.

Chomsky writes that linguistic description is formulated in terms of constituent analysis (1957). Using constituent analysis, we can consider a simple sentence to be a noun phrase and a verb phrase, and further break down these elements. When followed, we find that derivation of language based on this ruleset is straightforward but becomes vastly more complicated when more use cases are considered. This is because English is a context-dependent language. Chomsky further states that it is impossible to build a Deterministic Finite Automata (DFA) to produce every possible sentence in the English language. This can be explained by the concept of "plurality". Plurality of objects not only change the representation of their word, but the words that surround them. This is an interesting property of language, but is shown to be solvable by increasing the layers to the grammar, such that a "noun phrase" can either contain a singular or plural noun and the rest of the sentence would be modulated.

Chomsky's structure of grammar may not work particularly well as a formal definition for all languages, due to the nature of English being affected by context. Chomsky also notes that "noun phrases" may contain "verb phrases" and vice versa. These use cases are plausible, but can be assisted through the approach of using a grammar when utilised alongside postprocessing and a degree of Natural Language Understanding(NLU) to aid in generation.

### 3.2 ChatGPT

ChatGPT is a chatbot that works using the GPT-4 NLP model developed by OpenAI and backed by Microsoft (OpenAI, 2023). GPT-4 works through the use of transformers, having been pre-trained to predict the next word in a document by paying attention to the words before. Transformers are models which utilise an encoder and decoder to make a note of, and then utilise, the relationships between words to help the model generate coherent sentences (Thompson & Zachary, n.d.). The model was then fine tuned using user reinforcement. GPT-4 is the current market leader in NLP solutions.

Being a large multimodal model it has many use cases. It is capable of producing text for a wide array of purposes, including those outlined in the project motivation. The application takes a prompt, thinks for a moment, and then produces an output based on knowledge it has learnt through training. As it is a multimodal model, it is too large to be viable as simply a text production model and so is not appropriate for the use case. However, as the leader of the market it is important to analyse the structure of the output.

As mentioned previously, the cost to train an updated GPT iteration would be billions of dollars. (Brown et al., 2020). This means that it is prohibitive to the average independent programmer / software development studio to produce. The training cost corresponds to the amount of information that GPT is trained on. For example, GPT-3 is trained on approximately 45TB of internet data (Floridi & Chiriatti, 2020) and is capable of understanding and producing text in many languages, even programming languages. Whilst this collection of data will be incredibly useful to produce a model that is good at everything, a lot of this data will be ‘irrelevant’ to most use cases. This is not to say that the patterns learned by reading the text is irrelevant but it would affect the word choices used. Striking an appropriate middle ground would help my model learn both how to structure a sentence and also what word choices should be used.

Using ChatGPT online through the OpenAI portal, a user might notice that text is produced rather quickly and so may not see the issue. Were ChatGPT being run on an average user’s machine, it would run significantly slower. The server is an incredibly high-specification machine (Floridi & Chiriatti, 2020) and so is able to compute the information incredibly fast.

GPT is capable of being leveraged by other applications through the use of an API provided by OpenAI. The API is both powerful and easy to use, requiring only limited setup and prompt engineering by the user. This means that for a specific application, you can have your own pre-trained GPT model to provide output according to a certain prompt. This is an extraordinarily powerful tool to use in your applications. The main issue with this is that the software using the custom GPT model would likely be paid software, which could be an issue for many. Considering the scenario of a Role-Playing Video Game, where the developers want to use a language model to produce random text for each NPC in order to make the world feel more integrated, the number of tokens that could be generated by players all over the world could be prohibitively high. It would also require such a game to have a constant internet

connection to send and receive requests. Therefore, in this situation it is justifiable to program a smaller, more specific model into the game.

### 3.3 Tracery

Tracery is a lightweight replacement grammar based model written by Dr Kate Compton (2015). It allows for random language generation by method of replacement grammar. Replacement grammars are simple rules (called productions) that dictate the structure of a valid element of a language by replacing symbols with others until all "non-terminal symbols" in the element are "terminal symbols" and cannot be reduced any further (Engelfriet & Rozenberg, 1997). When using Tracery, the user defines a template for text to be generated, and then through either a deterministic or non-deterministic process (user specified), the template will be filled in with the user-defined words. This approach allows the user full control over the structure of the text produced. Anything that the user wants to remain the same can be hard coded. This is suitable for instances whereby you want to interact with a user directly using a prewritten greeting message. In certain cases, this may be useful. However, due to the small number of parameters for each type of production the text will always be very similar.

The main reason that this method works so well is that all components of the grammar are written up by a programmer or even end-user and then the algorithm chooses which ones to use. This means that there is a level of user control allowing all sentences to be themed the same, and provided that the text is written properly, it should always make sense. This has the drawback of requiring all text to be written by a user or otherwise carefully extracted, increasing training time. However, the effectiveness cannot be disputed. For a general model, this method would be insufficient as you would need to write full sections of text, like a whole noun or verb phrase. For generating "adventure hooks" for a D&D game, the general format of "tokens" could be:

[Description of Setting] [Description of Problem] [Call to adventure] [Rhetorical]

For example:

"*The town of riverside has many farms, each with bountiful crops. Recently, the crops have started to die. A farmer has called upon you brave adventurers to investigate. Can you find the source of the problem and put a stop to it before the crops run out?"*

And so a user would need to pre-write multiple sentences for each "token". Each of these tokens can take many forms, and so if you aim for an increase in variation it would be beneficial to generate each sentence separately. Therefore, breaking down the replacement grammar to individual sentence types would allow for greater variation. Training on the patterns in text would allow both the sentence and paragraph structures to emulate the training data.

The simple productions of a replacement grammar are positive when regarding the aims of this project, specifically to generate sentence structure in a computationally simple way. However, it may be

useful to extend Compton's method to allow sentences to be constructed from the ground up instead of filling in blanks. By extending Tracery's use of replacement grammar to work with a sentence entirely made up of irreducible tokens is an approach that may be quick and of minimal viable complexity.

### 3.4 LLaMA Models

LLaMA Models are another method of Natural Language Generation that was developed by Meta (*Llama 2: Open Foundation and Fine-Tuned Chat Models | Research - AI at Meta*, n.d.). The main selling point of the LLaMA model is that an instance is able to be finetuned and hosted by an individual or company, and as it is open source it is free to use in industry. LLaMA has two variants, one for language and one for code understanding. LLaMA-2 is optimised for dialogue use cases. The models have been found to outperform open-source chat models on most benchmarks we tested. Meta claims that they could be a suitable replacement for closed-source models.

LLaMA models are similar to ChatGPT, in that they work using Artificial Neural Networks (ANNs) that have been trained on between 7 and 70 billion parameters (Touvron et al., n.d.) and reinforced by human interaction. A set of human users have been asked to reinforce the prompts via a process called *supervised fine-tuning***.** This allows the model to reinforce itself with the "correct" choices.

The customisation and human fine-tuning features characteristic of LLaMA models make them a viable solution for the project. In order to reinforce good tagging and ensure that words are counted correctly, it will be recommended that the user can access the configuration file to alter information manually, or update a list of commonly mistagged words to override the process. It should be noted that these features won't directly influence the generation algorithm, but auxiliary software could be constructed to apply reinforcement learning in order to influence the types of words that are used. Users could then apply the highly rated sentences to the training document to reinforce the influence that those sentences have over the configuration file.

### 3.5 POS Tagging

Parts of Speech (POS) tagging is a method in NLP that uses datasets and sentence ordering to tag words based on where they occur in a given text (Martinez, 2012). For example, POS tagging would take the sentence "The dog ate chicken" and return a list similar to ["Determiner", "Noun", "Verb", "Noun"] to represent the categories of the words in the sentence. This is a relatively simple task but can be conquered in many different ways. Python contains its own methods for POS tagging in the NLTK

TextBlob is another Python library for NLP (Loria et al., n.d.). It provides not just POS tagging but also noun phrase extraction, sentiment analysis, and classification among others. This implementation is able to tell you more context about the words, however for the implementation of my algorithm this data might not be useful.

Stanford Core NLP is a Java library for NLP which allows for comprehensive breakdown of written text (Toutanova et al., 2003). Particularly, The Core NLP POS tagger is able to determine extra information about semantics from sentences. Stanford Core also provides details about the semantic field of a word, or the tonality of a sentence, just like TextBlob. Whilst this extra information is interesting, it could ultimately be unnecessary.

Through experimentation, it can be shown that CoreNLP is noticeably slower than the standard Python NLTK POS tagger. However, due to its accuracy and the depth of information which it provides it is certainly not to be overlooked as a candidate for POS tagging. While both NLTK and CoreNLP uses the 45-tag Penn Treebank tagset (Marcus et al., 1993), which breaks sentences down into 45 specific tokens and is generally considered the standard tag set for NLP, CoreNLP is able to return much more information about a text, such as the dependencies between words. This is an idea that is able to effectively combat the fact that English is not a context free language; CoreNLP is able to determine which words are reliant on others. This dependency recognition may be important to extend the algorithm should it prove to be unreliable.

### 3.6 Markov Chains

Markov Chains (MCs) are a versatile stochastic modelling approach representing the probabilities of transitions between current and future states (Mahfuz, 2021). Their core function is to illustrate a set of all possible states and give the probabilities of the transitions between them. They are typically characterised by a transition matrix, where each element captures the conditional probability of transitioning from one state (e.g., a word) to another. MCs have applications in various domains, including text prediction, where a first-order chain might predict the next word based solely on the current one. Markov Chains have a property called order. The order of a Markov Chain represents how many preceding states the current state is based on (Nth Order Markov Chains consider the N previous states) and therefore higher-order chains require a larger context. This increase in space could potentially improve prediction accuracy because considering more previous states may direct the selection to be more specific. Within the project, it is unlikely that many triplets of words occur particularly frequently and so the data gained could possibly be redundant aside from knowing that the words appear at some point in the text.

Beyond future state prediction, MCs are valuable tools for tasks like text segmentation, information retrieval, and modelling biological sequences (Mahfuz, 2021).

Within the context of the project, states will be considered to be the different word choices in a sentence. This will aid in the calculations for the optimal choice for the next word. A first order Markov Chain will be used to help with my text prediction. The use of second and third order Markov Chains was considered, however they could potentially complicate the selection process. These higher-order chains would each require more context and therefore a larger amount of data. It is more likely for a first order word pair to occur over a second order word pair, because there are more possibilities. It may be useful to

extend the algorithm to use higher order chains on a case-by-case, should the model be shown to be ineffective.

### 3.7 Summary of the Literature

There are many interesting methods for NLG, many of which require large data samples and intense pattern recognition. These ideas will inspire the project. ChatGPT's recognition of context and the lack of specificity in training data is certainly intriguing because it is able to recognise all patterns in data and call upon it when required, due to the use of transformers. This is beneficial as all of the data will (at least minimally) influence all outcomes. The scale of the algorithm I wish to create renders utilising a whole transformer unfeasible. Due to the fact that the project aims to run on low power machines, minimal context is required. However, considering further heuristics for analysis may be required should the algorithm fail to achieve its task in its finished state.

It is an important distinction between LLMs such as GPT and the algorithm that I wish to create that GPT is pre-trained to perform all tasks and can therefore respond to different prompts. The intention is not for my algorithm to intrinsically work based on prompts, instead it will simply generate text based on what it has read and understood. However, in order to direct the flow of the sentences, it may be beneficial to seed the data before generation in order to artificially influence the probabilities of word selection in order to direct towards what a user wants.

LLaMA models appear to be the closest implementation to the desired product. They are able to be adapted by a user to a given task and are trained with human reinforcement. However, the configuration process for an instance of the LLaMA model will be more difficult than the desired algorithm will take to implement. This said, for more intensive generation tasks or language processing, LLaMA models or GPT will be preferred. My aim is to make an algorithm that is efficient for producing short text without a user-passed prompt.

Likewise, the work done by Compton on replacement grammars applied to NLG has provided insight on a method to produce text. By taking it a step further and decomposing each part of a sentence into its own atomic components, the algorithm will be able to generate a greater variety of sentences than using user-defined components. Whilst the method in question would work well for some formatted sentences, the grammar used will be able to be subclassed in order to override certain aspects of my algorithm depending on use case. This not only allows me to make software that is general enough to be useful in most cases, but able to be trained to be specific enough for certain cases - developers can further expand on the work to add or remove complexity of the grammar as they see fit.

POS tagging will be required as a part of the project, for preprocessing the data before the algorithm is performed. POS tagging is the simplest way to tag all of the information (Martinez, 2012) and thanks to its implementation in Python,  is suited to a low-powered machine that an average user may have, rather than a fully defined server. The algorithm will use NLTK, Stanford Core and BLOB methods

for POS tagging. It will be recommended to the user that they should strengthen the dataset by using all three methods on a document, should one method mistakenly tag certain words.

## 4. Description of the Work

The overall aim of the project is to explore how feasible an NLG algorithm based on formal grammars and Markov Chains is in the modern day computing ecosystem. If my algorithm is found to be feasible, it could have the potential to be a suitable alternative to quick and NLG with a low computational cost whilst maintaining accuracy. This, in turn, would broaden the scope of possible use cases for NLG.

In order to achieve structurally sound sentences, the algorithm will use a custom recurrent grammar object which uses precomputed sentence structures with an element of random choice that is directly influenced by analysis of how often sentence types occur. This means that all sentence structures generated are not only correct, but that the frequency in which they occur is reflective of human written language. The grammar will use tokens from Python's Natural Language Tool Kit (NLTK) as the different POS taggers that as the data population algorithm already return the data in this format.

My algorithm will use this grammar to produce a tokenizsed sentence, and then with knowledge of the current word and token that will follow, the algorithm will use probability in order to aid selection of an appropriate following word. The core concept of the algorithm is similar to applying random, though directed, word choice to properly formulated sentence structure.

Once the grammar is working, a probabilistic selection process will be implemented to select the appropriate word. This will allow the word choices to be broader but more semantically directed. This will aid in producing consistent text and stopping "hallucinations" that have been described in models such as ChatGPT (OpenAI, 2023) by keeping track of the words that have already been used to describe it. The hallucinations described are errors that the model makes where information is fabricated and untrue.

The algorithm is intended to work as follows:

1. The algorithm will generate a series of tokens using a grammar object, for example:

   [DT, JJ, JJ, NN, VBD, IN, DT, JJ, NN]

2. The algorithm will consult the configuration file that it has loaded. The configuration file will have been populated by POS tagging text. An example file may have the following data:

   *The,DT,quick,JJ,1,*
   *quick,JJ,brown,1,*
   *brown,JJ,fox,NN,1,*

*fox,NN,jumped,JJ,1,*
*jumped,VB,over,IN,1,*
*over,IN,the,DT,1,*
*The,DT,lazy,JJ,1,*
*lazy,JJ,dog,NN,1,*

3. The tokens will be processed in a queue fashion.
4. As there is no data for the first token, a word will be selected randomly. The only suitable DT is "The", and so that word is chosen, and added to the return text.
5. Following "The", the algorithm looks for all instances of a JJ token that follow "The", in this case, the lookup vector becomes ["quick", "lazy"] and the transition vector is [1,1]. This represents that both "quick" and "lazy" have equal chance of being selected as their values in the transition vector are the same.
6. The algorithm chooses a word from the vector including the bias of probability. On this occasion, we have selected "quick". This word is then added to the return text.
7. The process repeats until the end of the queue.
8. Then the algorithm processes the return text to ensure that the punctuation is correct. The output is "The quick brown fox jumped over the lazy dog".

**Algorithm Requirements and Specifications:**

| Requirement | Specification |
|---|---|
| Allow for a dictionary of word pairs and their frequency to inform the selection process of future words | ● Create a data structure that will save a word pair and how many times it has been used<br>● Create a script that reads text and returns all word pairs in the script<br>● Save this information, as well as how frequently the pairs occur, to the aforementioned data structure<br>● Write the contents of the data structure to the user machine |
| Generate syntactically structured list of tokens | ● Program an object based on the designed grammar<br>● Write a script that will use this grammar to produce a queue of tokens<br>● Use the queue of tokens to randomly select a series of words |
| Generate text that makes sense | ● Using the tokens, implement a script that reads the aforementioned structure and |

| | |
|---|---|
| | computes the best choice for the next token based on frequency and probability.<br>● Use “context” structures to keep track of the subjects of sentences to ensure the sentences are coherent as a set |
| Be implementable | ● Be organised into modules that are wrapped in a class, which can be implemented and used easily. |

The plan is to implement the completed algorithm into a library, where each component is fully modular. All of the library functions will be wrapped inside of a single class which can be instantiated as a user-friendly way to call the algorithm functions. Setup of the object will be simple, as it will only require information about where to look for the data.

A modularity-based approach to development was chosen to provide the freedom to test, document and update each section independently of any others. Modularity and use of objects will also provide the added benefit of being overridable by other programmers through subtype polymorphism. Each component of the software will be tested independently before testing as a whole. The relevant tests will be run after each stage of development has been completed and any mistakes will be corrected.

Unit tests will be used to ensure that my library responds correctly to all regular, erroneous and boundary cases in the provided software. As mentioned, the tests will be written independently of the code to prevent the tests being written to pass the code and ensure that the implementation is truly robust. The software will also be tested using an adapted “Turing Test” (A. M. Turing, 1950) that will allow me to understand if the algorithm is suited to generating natural sounding language.

After the library is complete, it will be implemented into some small software demonstrations of ways in which the algorithm could be used to show its specific use cases that could be prohibited by larger language models.

## 5. Design

Initially, the implementation of this system was going to be carried out using a series of objects that represent nodes in order to emulate a series of Deterministic Finite Automata (DFA) which would work together to create sentences. A DFA is an abstract mathematical model that accepts or rejects symbols based on its current state (Vayadande et al., 2022). This property was to be used to link each possible state for a sentence to everything that could come after to create sentences. A DFA would have been created for each type of sentence and then linked together from the bottom up to form a larger DFA for every possible sentence structure. Each node in each DFA would have been an object of a Node class and each DFA would hold a reference to the start and end nodes, similar to an implementation of a linked list. However, research showed that it would be more efficient to define and create a computable grammar

for a subset of the English language because all DFA can be converted directly into a grammar (Indu et al., 2016).

Research also showed that the Python NLTK is a useful library that would help to simplify data collection methods. I planned to transfer development to the language “Mojo” once I had programmed most of it in Python. This is because Mojo claims to “combine the usability of Python with the performance of C” (Modular, 2023). As Mojo is a superset of Python, it was my understanding that only minimal changes to the code would be required in order to allow it to run inside of a Mojo wrapper. This means I can quickly program the majority of the project and then optimise a working version. Mojo uses CUDA, a parallel computing platform developed by NVidia that runs using the GPU (NVIDIA, n.d.) ideal, especially for training on long files.

Given the aim that the algorithm be easy to import and implement into a Python project, the project functions were wrapped inside of an object with simple user-accessible functions, which can then be imported and set up in around 3 lines of code. The object should allow a user to:

- Load a configuration file (.lcfg) path
- Populate the loaded configuration file with data scraped from an input (.txt) file
- Clear the data contained within the loaded configuration file
- Generate a number of new sentences using the loaded configuration file

The data is taken from the text file by splitting the text into sentences and POS tagging each word in the sentence. Then, for each pair of consecutive words in the sentence an entry into the .lcfg file is created. The .lcfg files are a simple formatted file type that looks similarly to a .csv file. The file stores two words, which token the words are and the frequency of the pair occurring. The token is the tag that is produced by the POS tagging process. This file is structured where each line contains a new entry, and each entry consists of 5 components:

WORD, TOKEN, WORD, TOKEN, FREQUENCY,

The object should be able to be imported and used immediately with only the setup of the configuration file. The object will act as a layer of obfuscation between the interface and the rest of the code. All scripts that are related will be put into their own module and referenced in the main object. There are four separate modules:

- Training Module to wrap all training functions and objects
- Stochastics Module to handle all probability and selection functions
- Context Module which will wrap the context handler class
- Grammar to wrap the grammar object

The items were modularised to make development simpler; modularity will help me to make each function do only one thing, and for each module to be used wherever necessary through imports. The project will be developed in PyCharm, an Integrated Development Environment (IDE) for Python.

In order to make the project work as simply as possible, it will use two auxiliary data structures. These are the TrainingStructure and WordPair. The TrainingStructure will be a 2D-array of type [[string]]. Considering the 2D-array as a matrix, we can visualise the contents as follows:

```
TrainingStructure = [{the, DT, quick, JJ, 1}, {quick, JJ, brown, JJ, 1} … {lazy, JJ, dog, NN, 1}]
```

It can be converted directly to and from an lcfg file.

Similarly, the WordPair data structure will hold a word and its relevant POS tag. This data structure will be used when adding to the TrainingStructure. It will act as a way for myself as a programmer to keep the implementation clear, and also allow for overriding by any future developers.

Chomsky writes that it is impossible to construct a deterministic finite automaton to compute every word in the English language (Chomsky, 1957). It is because of this that I adapted my original idea to construct a large DFA network to instead formal rules of grammar to a list of possible words based on context. The grammar will obey all rules and use a recursive-descent method similar to parsers to construct a list of tokens. This will ensure that no incorrect sentence structures can be generated. These syntactical patterns have been helpfully broken down by Libron Kelmendi (2020). It should be noted that this is similar to Chomsky's work on the same subject. Subjects in Kelmendi's work are equivalent to Noun Phrases, and alike with predicators and verb phrases (Chomsky, 1957)

As illustrated by these simple patterns, each successive sentence type will be chosen by a function that calculates which sentence type is most appropriate given the structure of the paragraph. This will be done using the first instance of Markov Chains.

Figure 1. Kelmendi's Sentence Structures

Simple Sentences (Independent Clause):
- **Pattern I** : Subject – Predicator – (Adjunct).
- **Pattern II** : Subject – Predicator – Subject Complement
- **Pattern III** : Subject – Predicator – Object
- **Pattern IV** : Subject – Predicator – Indirect Object – Direct Object
- **Pattern V** : Subject – Predicator – Object – Object Complement

Compound Sentence:
- **Method I** : Independent Clause + Comma + Coordinating Conjunction + Independent Clause
- **Method II** : Independent Clause + Semicolon + Independent Clause
- **Method III** : Independent Clause + Semicolon + Coordinator + Comma + Independent Clause

Complex Sentence:
- **Method I:** Coordinating Conjunction + Dependent Clause + Comma + Independent Clause
- **Method II:** Dependent Clause + Comma + Independent Clause + Comma + Dependent Clause

Compound-Complex Sentence:
- **Method I:** Dependent Clause + Comma + Compound Sentence
- **Method II:** Compound Sentence + Comma + Dependent Clause

Once the type of sentence is chosen, another function will call any other lower functions that are required. For example, in Method I for "Compound Sentences", the production would consequently call the function to generate an "Independent Clause", then add a "Comma" token, then a "Coordinating

Conjunction" token and then produce another independent clause. The Independent Clause will produce a series of tokens in the same way. Eventually, this will produce a queue of tokens to be processed. These tokens will represent core elements in the grammar shown in figure 2:

Figure 2. The Designed Formal Grammar

```
sentence ::= sentence sentence | compound-complex | complex | compound | simple

compound-complex ::= dependent clause, comma, compound | compound, comma,
                     dependent clause

complex ::=  coordinating conjunction,  dependent clause, comma, independent clause |
             dependent clause, comma, independent clause, comma, dependent clause

compound := independent clause, comma, coordinating conjunction, independent clause |
            independent clause, semicolon, independent clause | independent clause,
            semicolon, coordinating conjunction, comma, independent clause

simple ::= independent clause

independent clause ::= subject, predicator, adjunct | subject, predicator, subject
                       complement | subject, predicator, object | subject, predicator,
                       indirect object, direct object | subject, predicator, object,
                       object complement

dependent clause ::= subject, predicator
```

In the grammar above, tokens such as "subject" are non-atomic, meaning that they are composed of at least a Noun, but potentially a set of Adjectives and Nouns or even containing "The". These elements are broken down further so that each component of the token is ensured to be authentically generated. In the generation script, the token queue is read item by item and, through the use of Markov Chains, a sentence will be produced. By applying probabilistic generation to structure, it is my hypothesis that the program will produce well structured sentences that make sense. After the text has been generated post-processing functions will apply correct capitalisation and will replace any full stops with exclamation marks and question marks where appropriate.

The grammar may, in some cases, need to be overridden. For example, should a developer choose to implement the algorithm into a joke writing program it would require a different set of grammar rules. For this reason, the grammar will be wrapped into an object that calls through the levels of the grammar rules and returns the token queue. The user will only be able to see the generation function unless they want to override the class.

The NLG object will contain a reference to the grammar object so that the grammar used can be changed inside of the program. It will also have functions to perform the tasks outlined above. The other

modules will be used throughout the program. This will abstract all of the processing away from the user, streamlining the setup process.

To illustrate the software's ease-of-use, some software demonstrations will be produced that use the package. The demonstration software will show off a range of use cases such as speech generation, prompts for D&D and descriptions of a scene.These demonstrations should also show off the power of the system, should it prove to be feasible.

# 6. Methodology

## 6.1 The Project Structure

The project will be developed in Python using PyCharm Integrated Development Environment (IDE) because of the outstanding refactoring tools that the suite provides which will allow me to quickly correct any mistakes and clean up the codebase towards the end of development. The IDE comes with its own automatically flagged and enforced coding conventions, which will also be used to ensure the software is high quality.

## 6.2 The Algorithm

The algorithm itself, named NLGLite, requires only a list of information about word pairs and their frequency as well as a series of tokens that represent the sentence to be formed.

In order to process the information, first the algorithm reads all of the information and looks at the first token. Because there is nothing before the first token, we check all words that arrive at the start of a sentence of the correct token. For example, a DT token might be replaced with "The". We do this by calculating a transition matrix from the data. Then the algorithm chooses a word, randomly, but with the bias weighted by the frequency of the word. This means that we order the data and select a choice using the frequency of occurrence as the weights for the selection. The data will be stored in the configuration files as specified in section 5.

Then, we repeat the process looking at the previously chosen word and the current token. We look up from the table all words of token XX that follow the previous word. For example, if the token was NN (for noun) it may be replaced with "Dog" if the data shows that "Dog" commonly follows "The". We then repeat this for all tokens in the queue. Finally, the algorithm dictates that we look through the sentence, then capitalise words as appropriate and return the finalised sentence back to the user.

## 6.3 Markov Chains

The core of the computation in the project is the calculation of probabilities using Markov Chains. Markov Chains are statistical models that are used to predict a successor state using only knowledge of the current state. Markov Chains are the industry standard for NLG and are used in

smartphones and such smart devices in their text generation / prediction software (Vermeer & Trilling, 2020). This heavily influenced my decision, as well as the fact that other probabilistic models require information gathered from hundreds of examples of text in order to train them. Using deep learning, the model would need a large set of examples from which to read and interpret text.

Markov Chains were chosen as they do not require knowledge of previous states to influence the decision of the next outcome. This means less context needs to be tracked during the training process and this will reduce the memory that the algorithm takes up when implemented. Markov Chains are still a stochastic model, however, so they incorporate a degree of randomness into the output. This allows for variation in a very simplistic but effective manner.

A human reinforcement approach was considered, whereby the algorithm would produce sentences then a human would classify each sentence based on whether or not they made sense, and the sentences that make sense will be added to the configuration file. This approach would have ultimately been lengthier with minimal impact to the success of the automated data-obtaining methodology.

In order to successfully implement Markov Chains into the software, the matrices that the chain is based on had to be calculated dynamically. Markov Chains work by referencing a transition matrix. In order to calculate the transition matrix for an entire text it would require a matrix of size n*n, where n is the number of words in the text. This will grow rapidly for the size of the dataset. Calculating it dynamically, the maximum size of the vector that will be needed (considering a vector as a matrix of 1*n) is simply n entries. By keeping track of the possible transitions the probability can be calculated dynamically which will further cut down the size of the transition vector, increasing both space and time efficiency.

## 6.4 Language Representation

Formal Languages can be represented in many ways, such as through Deterministic or Non-Deterministic Finite Automata, Regular Expressions, Grammars, Turing Machines and Push-Down Automata (Yan, 1998). When ideating this project, the choice of how to represent the structure of the English Language had to be considered. Initially, the use of a DFA would be the most effective method. After further research, an approach similar to inverse-compilation would be better suited. This works by building a string from tokens instead of tokenizing a string like a compiler would, though using a grammar in the same way. The reason for choosing grammars over DFAs was that instead of chaining together what is effectively a countably infinite set of states, a structured list of placeholders can be created that are then replaced with the calculated word choices afterwards. This will mean that the sentences should generally make sense with little modulation based on tense.

This can be illustrated by using Google's predictive text feature on android phones. If a user repeatedly presses the middle option, they will receive nonsense text generated with poor structure. Whilst regular expressions are theoretically suitable for building the structure of sentences and then populating the gaps, functionally they are the same due to the nature of language theory.

It is important to reiterate that English isn't a context-free language. English requires context in order to produce sentences with consistent tense and plurality. Through the tokens provided in the python NLTK, a larger grammar could be used to produce different types of sentence components, such as plural noun phrase or singular noun phrase. As mentioned in section 6.2, the method of training is expected to influence the context of the sentence.

# 7. Implementation

## 7.1 Creating the Library Functions

The aim of the project is to create an algorithm that can be imported easily and simply into a Python project to produce language. In order to bring my ideas to fruition, I first made a list of the critical components that the algorithm would need should it be implemented. I already knew that my language of choice was Python, due to the incredible amount of resources provided by the NLTK package. The critical components of my library were:

- Training suite, containing a reader and scraper
- Auxiliary Data structures including a wordPair and trainingStructure
- Stochastics suite
- Generation algorithm

The training suite will contain a series of functions to read a text file and convert it into a lite configuration file. The functions used in the training suite include reading the file, processing the word list, outputting the training structure, tagging the text with NLTK, TextBlob and Stanford Core methods.

To populate the configuration file, the scraper will categorise each word in the text file using POS tagging and then create a list containing tuples, which contain the WORD and its relevant TOKEN. Then, it analyses which words come after each other and outputs this information to the file in the order specified above. The process to gather data was bottlenecked by both time and memory. One file that was fed to the algorithm was the complete works of Shakespeare, which was approximately 5Mb in size. When initially scraping the file, after 45 minutes the file was only populated 3.4%. It can therefore be calculated that to scrape the whole file it would have taken approximately 22 hours. The population algorithm was therefore edited to improve the speed by making changes to the memory management of the implementation by deleting lists between loops. After the improvements, when running the algorithm on the file once again, the process terminated when the processing was 10% complete due to lack of available memory.

The first prototype of my algorithm was designed to randomly produce text in a format that was readable and understandable. However, these sentences didn't always make sense as Markov Chain usage and context retention had not yet been implemented into the system. Despite this, the system worked suitably (with minor grammar adaptation) for certain use cases, and could have already acted as a

sufficient method for generating pseudo-random strings applicable to software such as chatbots and idea generators. Further extension of the code base would have involved keeping a custom data structure containing word pairs, their token types, and statistical data about their usage. This statistical data would have been taken from literature from a variety of sources and types.

When implementing the grammar, it became apparent that it was not exhaustive and couldn't accommodate all sentence structures. This was due to the fact that my original design did not produce atomic tokens and so when considering “Subject”, this was not always a noun. Hence, the design of the grammar had to be changed to be more comprehensive.

Figure 3. Improved formal grammar

```
sentence := compound_complex | complex | compound | simple | question | exclaimation

compound_complex := dependent_clause, comma, compound | compound, comma, dependent_clause

complex := prepositon, comma, dependent_clause, comma, independent_clause
        | dependent_clause, comma, independent_clause, comma,  dependent_clause

compound := independent_clause, conjunction, independent_clause
        | independent_clause, semicolon, independent_clause
        | independent_clause, semicolon, conjunction, comma, independent_clause

simple := dependent_clause

exclaimation := verb, independent_clause

question := wh-determiner, verb, subject, verb

dependent_clause := subject, verb_phrase

independent_clause := subject, verb, object | subject, verb, adjectives | subject, verb, adverb
        | subject, verb, verb_phrase

verb_phrase := modal | present tense verb | adverb | verb | verb gerund

subject := predeterminer | noun | noun, conjunction, noun

object := noun

complement := prepositon, noun | "a", noun | adjective

noun := determiner, adjectives, noun | name | "a", adjectives, noun | pronoun, adjectives, noun
        | possessive pronoun, adjectives, noun

adjectives := adjective | adjective | adjective
```

## 7.2 Wrapping the Library Functions

The library functions were all wrapped in the NLGlite object. This object then serves as a central encapsulation point library's functionalities. This encapsulation offers developers a streamlined approach to accessing the library's capabilities with only a few lines of simple code, in that developers can leverage the entire library's functionality with a single import statement. Furthermore, the inclusion of only essential functions within the object ensures a compact codebase, mitigating concerns regarding file size inflation.

In order to wrap the library function, a well-defined project file structure was required. This is because Python modules rely heavily on a specific file structure, which is orchestrated and managed through initialization files.

It became apparent that the grammar module should be wrapped in an object, so that developers can use subtype polymorphism to override the stages should they want to change the way that the sentences are generated. This would, theoretically, allow the implementation to work with different languages aside from English. For example; a grammar for writing code could be produced. Provided that developers published these grammar objects, the implementation could soon grow in scope.

When it came to the design of the object, each of the functions that were necessary to produce text were wrapped in functions that could be called through the object. The object was designed to accept a set of parameters such as a reference to the file intended for loading, the training structure employed, the grammatical rules utilised, the token queue, the desired number of sentences to be generated, and the final text output produced by the object. The object therefore has to manipulate the configuration file n training (of which it has a reference to the file path) such as creating and populating a file and it can also generate sentences. All of these functions reference their own parts of the training suite. As mentioned before, the entire project has been modularised so that components can be changed and updated throughout development to enforce good coding practice. The adoption of this modular design philosophy promotes the creation of clean, maintainable, and well-structured code.

The context tracking object was removed in the final implementation as it was discovered to be redundant. For many nouns, giving the example of “Dog”, the descriptors will usually be the same in a given text. It is unlikely that the “Dog” will be described both as “ferocious” and “docile”, and therefore the trace of the context tracking object showed that the words it had tracked were merely a subset of the possible words. A more suitable solution to this context tracking issue would be dynamic adjustments to the frequency during runtime.

## 7.3 Documenting and Testing the Library

Effective documentation is crucial for any software library, the aim is that users will be able to understand exactly how to effectively use the library. Docstrings were used for every function to appropriately document them with what it is that they achieve, their parameters, and return variable. The notes taken throughout development were used to inform the documentation, which was then used to generate PyDocs, which offered a clear and centralised reference for understanding each function’s workflow and parameter usage in HTML format. These will be published alongside the repository when the library is released.

Unit testing was also critical to the success of the library. If the library wasn't properly tested, then it may cause runtime errors in any software where it has been implemented. Tests were written for each module’s series of library functions, and then a test suite was created for the wrapper object. This iterative process identified areas for improvement, leading to necessary code corrections. Whilst

Test-Driven Development (TDD) was considered early on in the development process, the fact that changes to the methodology were possible at points throughout development meant that the tests could have been redundant, wasting time and resources. However my documentation of each function served as a valuable resource during test creation, as the docstrings were used to inform the tests on the intended behaviour of each function, so that tests were not written to validate the existing code. This focus on comprehensive testing ultimately resulted in a higher quality of the code base and ensured that the project was fully functional as intended. This also means that when developers implement the code, any errors caused by overriding or misimplementation are not due to poor quality code.

## 7.4 Building the Demonstration Software

To demonstrate the versatility and user-friendliness of the NLGLite object, a suite of applications were created to showcase its capabilities in real-world scenarios. This suite highlights the algorithm's potential for integration into various software environments, and is intended to be shown on the demonstration day for the project.

The main piece of software was a software “Engine” application. This is an interactive program that allows users to harness each of the functions that the NLGlite object has. This means that users have the option to manipulate configuration files, as well as populating them with data using the “training” functions. This illustrates the way that the software works in code, which allows users to familiarise themselves with the capabilities of the project before using it in their own development.

Next, a snippet of a game scenario utilising NLGlite in conversations with an NPC was produced. The player is only able to interact with NPCs and read the text. Every interaction would result in freshly generated dialogue based on a training set of the sort of things the NPC would say. This is built to demonstrate the potential that NLGlite has to enhance gaming experiences by creating characters that are engaging and never the same twice. This would enhance immersion as no two conversations would be the same.

Finally, the algorithm was used in the creation of a Discord bot that generates text to send messages on the popular chat platform. Users can interact with the bot by sending text commands within a Discord message and the bot responds using NLGlite to generate some text. This showcases NLGlite's potential for real-time interaction and engagement within online communities.

The ease of building these demonstrations highlights the user-friendliness of the NLGLite object. Its ease of integration allows developers to incorporate text generation capabilities into various software environments without extensive coding requirements that will run efficiently on low power machines. This opens doors for exciting new applications across an array of fields, such as messaging, game development and TTRPG help.

By showcasing NLGLite's potential through these interactive demonstrations, the aim is to highlight the variety of use cases that a lightweight NLG algorithm would provide, and potentially

highlight any shortcomings that may be revealed by the evaluation of the algorithm. This would then allow me to further develop based on real intended situations. Use of the algorithm in the right ways should lead to a series of engaging and dynamic user experiences on a wide range of machines.

# 8. Evaluation

## 8.1 Critical Evaluation of the Project

To evaluate the effectiveness of my library, I used books that are in the public domain to produce text in the style of certain authors. I then ran an online “Turing Test” using a quiz implemented in Google Forms, a questionnaire site, that provided me insight into the success of the algorithm. The Turing Test works where a user speaks to both a human and a machine (A. M. Turing, 1950), though the user doesn't know which conversation is generated by the computer or the human. Therefore, a machine would be considered to have passed the test if the user is unable to distinguish which conversation is generated by the computer. The test had 5 questions, each focused on a particular public domain literary work. Participants had to select the 6 genuine quotes from a list of 12 quotes, half of which had been written by the algorithm trained on the respective novel. To achieve success, the data should show an approximately 50:50 split between picking the correct quotes and the quotes that the algorithm has produced (A. M. Turing, 1950). I adapted this idea to gather the most data in the shortest time. All participants have been informed exactly the task they are to perform and told exactly why at the top of the form, and all data was provided anonymously. This allowed me not only to test the effectiveness, but also to compare how effective the algorithm was on differently sized training sets.

Whilst considering the method of testing,the originally intended use case was to be demonstrated by attempting to run a game of D&D using generated prompts.In D&D, a player called the “Dungeon Master” (DM) unveils an adventure and ultimately controls the world that the players (who act as their character) explore. This role can require thinking of many types of prompts for players on the spot. An insufficient amount of training data was able to be gathered to produce anything workable and therefore it was determined that even without further study, the algorithm would have performed insufficiently in this use case. Therefore, the testing method was revised.

Were the original test feasible, it would have used multiple configuration files, each trained on a separate sort of text production. For example; adventure hooks (which introduce the players to the wider adventure), character or location descriptions, character speech, and even combat descriptions. The DM would use the relevant configuration files as necessary to produce the text.

The test was distributed online to a random population of participants who were each encouraged to share the test, which received 60 responses. The responses showed that whilst the algorithm is not perfect, of the 60 attempts at each of the 6 questions only 15 questions in total have had a fully correct output. This alone would imply that the algorithm has performed well, however this just means that users can comfortably pick mostly real quotes and then guess trickier quotes. When creating the test, the types of sentences used were chosen carefully, each chosen for their different structures which would allow

further assessment of how the algorithm performs at replicating different structures. This choice has allowed me to compare at which complexity the algorithm fails to perform. After analysis, it was found that this was the case for all 5 questions.

The fact that only 15 of the total answers to the questions were correctly answered lends itself to the idea that the algorithm is able to produce *some* text well enough to fool a user. This means that it performs better at some things than others. By careful analysis of the data provided, it can be determined that the algorithm is able to produce shorter sentences, or parts of a sentence, very well. When generating compound or complex sentences, the algorithm is more prone to failure. This could be explained by the fact that there are simply more points for error, as well as the training datasets being particularly small.

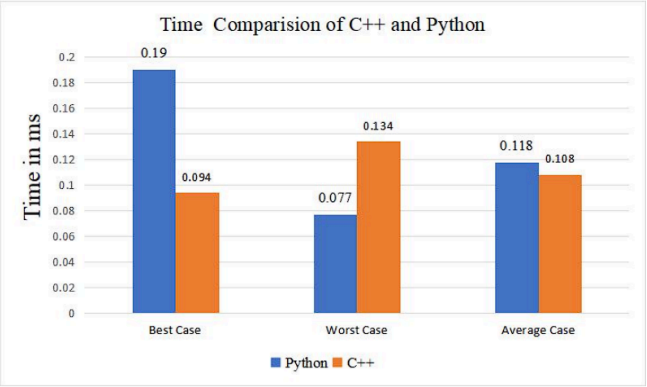
To confirm this, a small script was produced (separate from the main project) to gather frequency data for all of the different words contained within each text, and how many unique instances of each word there are. This data is presented below:

Figure 4. Table showing Number and Frequency of words

| Title of Book | Number of Words | Avg. Frequency 3.d.p. |
|---|---|---|
| Crime and Punishment | 10,259 | 20.139 |
| War of the Worlds | 7,691 | 9.577 |
| Dracula | 9735 | 16.538 |
| Pride and Prejudice | 7,026 | 17.417 |
| A Study in Scarlet | 5,960 | 7.328 |

By looking at this data, it becomes apparent that a single novel is not enough information for the algorithm to be satisfied. This was a bottleneck for the project throughout development. When searching for data, the gathering and preprocessing of training data became difficult. This difficulty stemmed from lack of available works that are in the public domain and that were able to be preprocessed. The algorithm was trained on a file containing the complete works of Shakespeare, which was approximately 5.5MB in size. When training the algorithm on this dataset, the processing consumed memory at a rampant rate and resulted in the algorithm unable to produce a full output for the file. This resulted in the conclusion that in order to retrieve and process an appropriate amount of data, you will require a very fast processor and this may not be possible on a low power machine. The largest of my training documents was “Crime and Punishment”, with a file size of 1,111KB. The configuration file produced for this document was 2,157KB large. The Shakespeare document was 5,214KB large, and so therefore it can be predicted that the training document will be approximately 10,000KB in size. The efficiency of the training algorithm will need to be improved in order to make the algorithm viable in the modern ecosystem so that the users may use a larger dataset. It could be argued that due to the speed of execution (Zehra et al., 2020), Python isn't a good language for the training script to be written in. As the time complexity of the algorithm is

already fairly large, the speed of execution of the Python implementation is much slower than it would be if implemented in a language like C or C++. This is evidenced below:



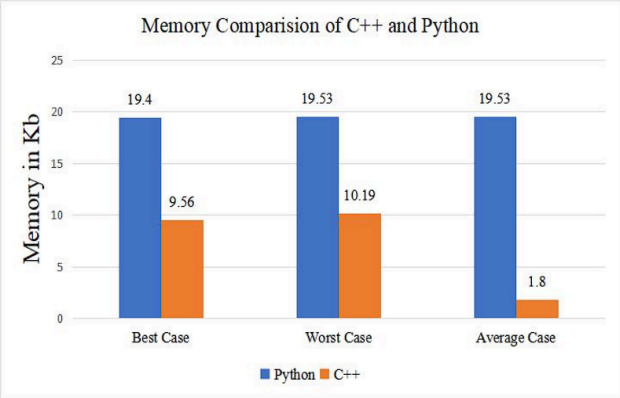
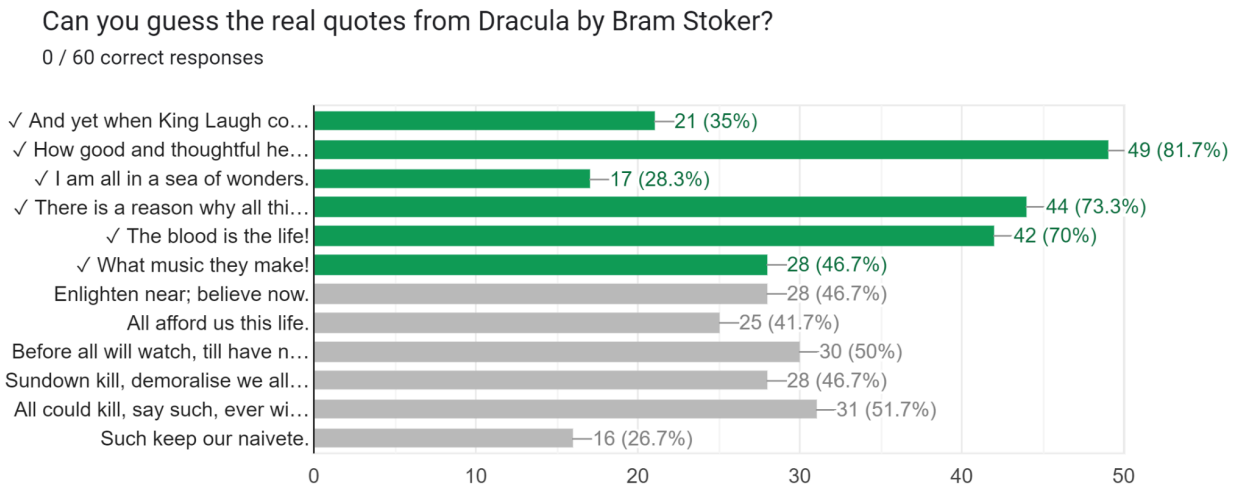




Figure 5. Graphs have been taken from a paper by Zehra et al. (2020).

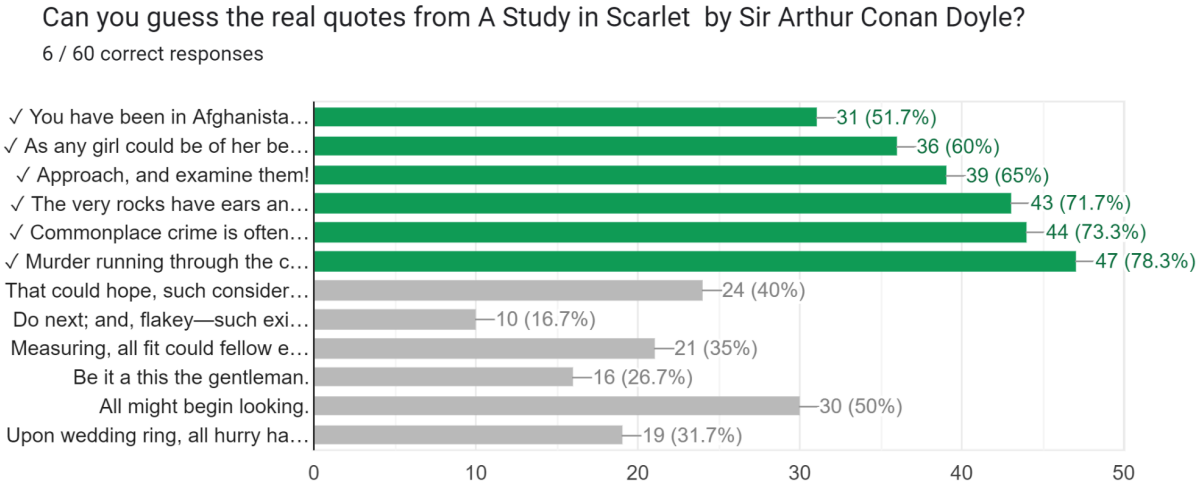
Initially, the aim was to use Mojo to speed up the execution using the increased execution speed and Python-like syntax (Modular, 2023). However, when porting the Python code to Mojo - there was only a minimal speed increase.

In terms of analysis of the success of text generation, the algorithm did not perform to the desired standard for me to confidently claim that it is viable. The 60 responses that the test received was a large enough sample size for trends to start to appear. Generally across the data, the real quotes have been chosen vastly more frequently than those generated by my algorithm, with the exception of “I am all in a sea of wonders” from Dracula and “We can’t have any weak or silly” from War of the Worlds.

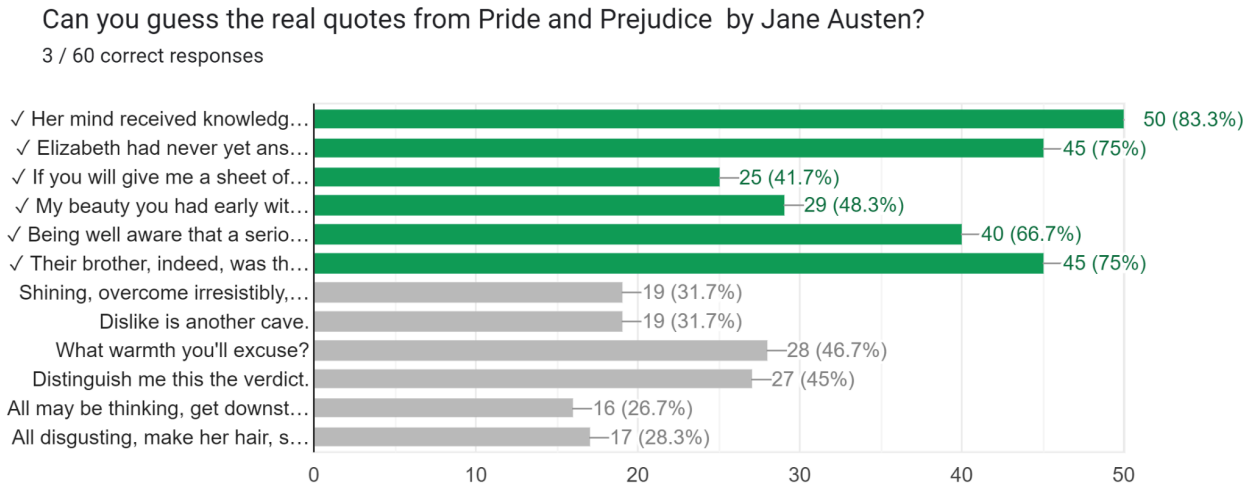
The test uses 12 quotes for each chosen book, half of which are real. Keeping the number of quotes to 12 would lower the cognitive load on the subject and therefore the data will likely be optimally accurate and informative. My prediction for the results was that generally, users will be able to confidently identify under over half of the real quotes, correctly. The ideal result for all of the questions would be approximately 30 selections for each quote in each question. This would show the ideal 50:50 result, which is the desired result for a successful Turing test (A. M. Turing, 1950).



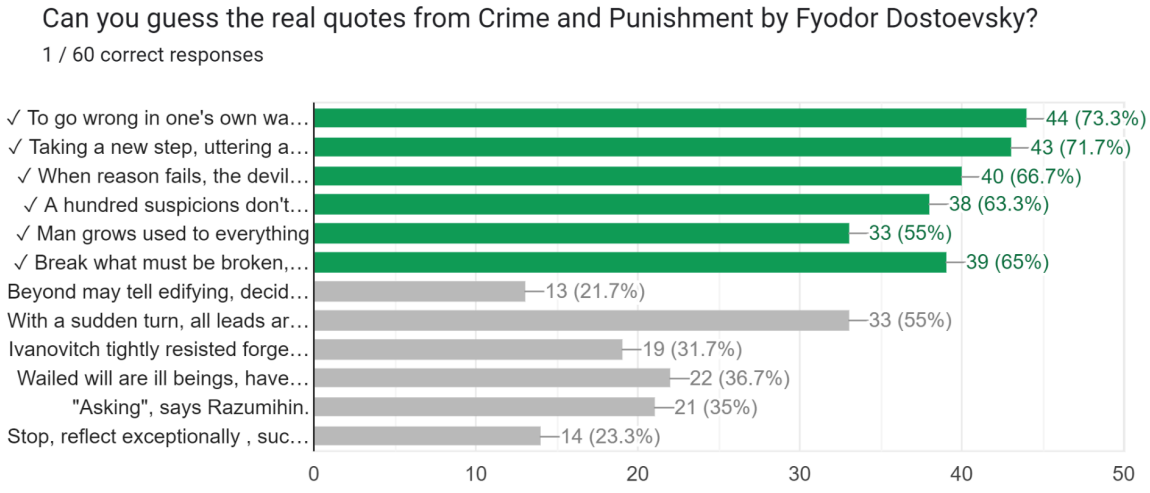

Looking firstly at "Dracula", we can see that generally people found this harder as there is a more uniform distribution of quotes. Dracula's configuration file is 1,689KB in size, the third largest. Generally, most of the quotes were selected around 30 times, as desired. Two of the correct quotes were selected almost all the time. The quote "How good and thoughtful he is; the world seems full of good men--even if there are monsters in it." was identified correctly as a real quote 81% of the time. With the correct quotes, users fit my hypothesis exactly. Users were able to correctly identify two of the quotes, and struggled on some of the more difficult ones. The wrong quotes were almost all picked around 27 times. The correct quotes were either picked almost always or picked rarely. This implies that the dataset is not quite good enough to consistently emulate the writing style of Bram Stoker.



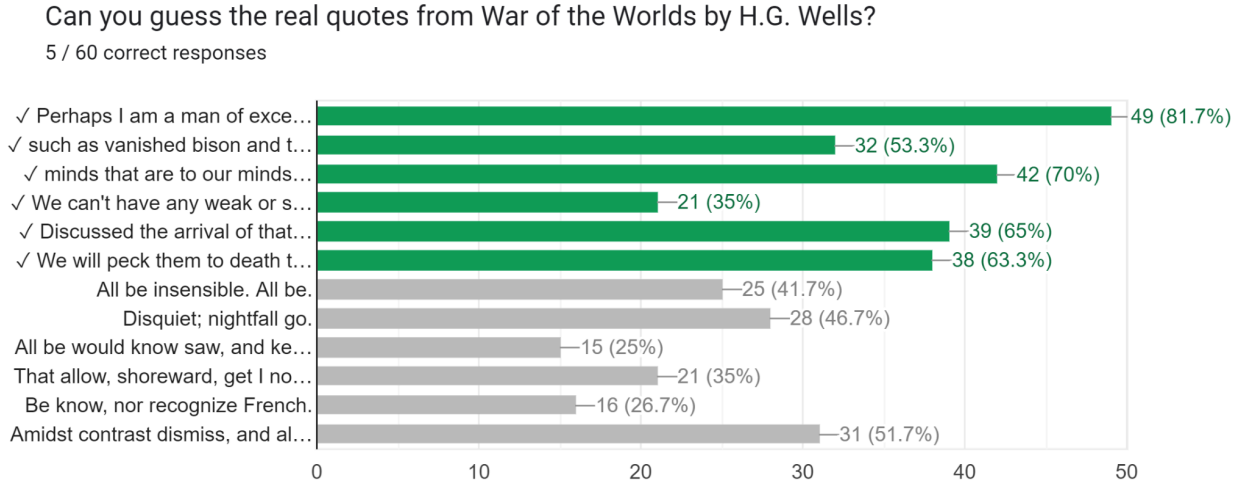
When looking at the results for “A Study in Scarlet", we can see that people were very good at selecting the correct quotes from this novel. The book yielded a configuration file that was only 237KB in size, which was the smallest in the dataset. Each correct answer was chosen over 30 times, and this question had the most fully correct responses of each of the questions. The fact that this had the smallest dataset and the most correct answers implies that the model performs poorly on short texts.




Pride and Prejudice, by Jane Austen, was another interesting test case. The results show once again that the real quotes were chosen confidently a large proportion of the time, with the exception of the shorter quotes. The participants have shown a fair difficulty in selecting shorter quotes that have been generated by my algorithm. The file has a size of 2,154KB, the second largest. It is therefore noteworthy that participants appeared to have such an ease of picking the correct quotes from the second largest file.



“Crime and Punishment” was an interesting book to analyse. This book has been translated from Russian, and so isn't in the original language it was written in, which I thought may influence the writing style. Crime and Punishment had the largest configuration file of 2,157KB in size. We can see that the correct answers were chosen very frequently. The generated quote, “With a sudden turn, all leads are dark.”, was, however, chosen as frequently as the least frequently identified real quote. This sentence is generally well formed and the context appears to have been held well in generation. Given that this is the only instance of this occurring, it could simply be a statistical anomaly. The implication of this is that with more data this will happen more frequently.




“War of the Worlds” also had a small configuration file size, at only 896KB. Analysing the results of the quiz we can see that the majority of the correct quotes were chosen far above the desired frequency. Simply, we can tell that the model does not perform well on smaller files. As predicted, this novel received the second most fully correct responses and has the second smallest configuration file. From this, we could infer that the more data the algorithm gets, the better the sentences are. However, Dracula and Pride and Prejudice do not fit with this rule. Through plotting the data in figure 4 onto a graph, only a weak positive correlation between the number of words and the average frequency of a word can be ascertained. I also plotted a graph between the average frequency of a word and the size of the file. These graphs can be found in the appendices.

Analysing the types of generated sentences that were chosen, we can see that the model is better at producing shorter sentences. This is likely because of the fact there are simply less tokens, and therefore it is easier to choose a sentence that might make sense from the data provided as there are less choices to make. Equally, context and the selection of words is based on chance and not the grammar. Implementing a tiered selection process for the grammar, in that the subject is chosen first and the rest of the sentence is filled in around that, may be a way to improve the structure of the output sentences.

While the current evaluation paradigm focuses on assessing Natural Language Generation (NLG) algorithms' ability to produce grammatically correct and coherent single sentences or sentence fragments,

this approach presents limitations in comprehensively evaluating an NLG model's capabilities. NLP in real-world scenarios often involves generating coherent and cohesive paragraphs that fit in well with the surrounding discourse. Focusing only on single or partial sentences fails to capture the algorithms's ability to maintain coherence and context over extended paragraphs. In order to evaluate the success of generating paragraphs, users would need to perform a similar test but select the correct paragraph. Such a method would necessitate a reduction in the number of response choices offered to human evaluators to ensure that they are able to complete the test with enthusiasm as the increased cognitive load associated with evaluating more then just single sentences could lead to decreased user engagement, which may invalidate the test results.

An aim of the project was to ensure that the algorithm can run on a machine with low computing power. Unfortunately, due to the computational complexity of the algorithm, it is infeasible to be able to run the algorithm on a machine with a fairly high benchmark, even in its current state. This could be mitigated by porting the software to another language that is able to make better usage of CPU resources. Python runs on a single thread of execution therefore will only be able to be run on a single core at a time. Python still allows the program to run on low powered CPUs when disregarding the time taken. When obtaining the data, the memory usage of the program grows rapidly and although attention has been paid to this issue, it is outside of the current scope of the project to develop a full solution to the efficiency of the training process.

The results indicate that Markov Chains applied to formal grammars are not feasible as a method of natural language generation in the modern day computing ecosystem with the method that I have used outside of short sentences. The evidence shows that due to lack of pattern recognition and data, the algorithm had limited overall success.

## 8.2 Further Development

The data gained through the study illustrates that there is limited success at sentence generation and therefore success of paragraph generation would likely be equally as limited, if not greater. In order to improve the paragraph generation, matching different factors about the sentences that follow each other over just words could help to improve coherence of writing. Further steps for improvement could include improving the efficiency of the training algorithm with a complete rework and then running a second, more in depth study. I would also like to add to the configuration file with more context, such as sentence structures and relation between words, and how the subject modulates them. It may be interesting to tokenize sentences completely to gain structure, which may remove the need for the grammar completely. However this will necessitate a more in-depth configuration file structure and could reduce the lower threshold for the computing power the algorithm can run on.

Were the project to be rewritten considering the experience gained from the work carried out thus far, I would alter the token queue to become a simple list that I would process in a tree-parsing structure. This would allow me to build the sentence from the subject first and then use the knowledge gained about the relationship between words to fill in the gaps in priority order. This would require intermediary

tokens, because the tokens that represent the subject of a sentence, for example; [“DT”, “JJ”, “NN”], could be confused with another part of the sentence. This intermediary step would then require swapping the subject token for a subject structure, similarly to how the grammar works in this current iteration. I further hypothesise that this would improve the meaning of the sentence in a semantic sense.

By using more in-depth text tagging, deeper relationships between the words in a sentence and its subject can be found and this can improve the selection algorithm by using more calculations. By using second or third order Markov Chains, the quality of the sentences could be improved. With the increase in order, I hypothesise that the semantic field of each word can be focused on and this will make the sentences more coherent. When generating multiple sentences, the subject of the previous sentence would indicate what the currently generated sentence should focus on. My original hypothesis is that this kind of context management is intrinsic to each sentence and so should be reflected with the transitions between sentences.

## 9. Laws, Social, Ethical and Professional Issues

By developing the algorithm, intellectual property in the form of source code and documentation, as well as the idea behind the algorithm itself has been created. I intend to publish the software on GitHub for free under the creative commons licence. From this point, the project will be open source. This is all covered under the Copyright, Designs and Patents Act (1988).

There is a potential that this language model could be used to generate hate speech or explicit content that targets individuals in a harmful, threatening or otherwise negative way. There is a risk, however, that someone else utilising the model may choose to use it to produce negative or harmful content. Due to my intention of releasing the code in this way, I acknowledge that I relinquish control over certain aspects of how the code is used.

Another ethical concern raised by the project is that theoretically my code may be used to write novels, scripts or other forms of widely read literature. This may contribute to a flood of AI driven content on online marketplaces, which can undermine the effort put in by authors when producing a heartfelt piece of literature. However, given that the model is severely limited in terms of producing contextually sound text in its current state, as illustrated by my analysis, this is unlikely to be an issue.

Due to the nature of populating the configuration files, it is assumed that users will be using other people’s text to generate sentences. This raises ethical issues as there is no real way to measure if users are utilising pirated or otherwise stolen text. Users will be encouraged to only build configuration files based on text they have full permissions to access but by releasing the software for free I accept that I am unable to enforce responsible usage guidelines.

There could also be bias in the training data. Despite not being an AI model, the model still needs some statistical data to read from. By training it on classic literature or nonfiction from a biassed source I could involuntarily introduce word combinations that are outdated or one-sided which may skew the

output of all text through the model. Currently, no effective way to mitigate this has been designed as the users will be building their own training datasets. In the documentation for the library, users will be encouraged to use a variety of literature on the training data in the aim of removing bias.

Ethically, the only tricky aspects to the work were the ownership of the generated sentences. Due to the way that training data is likely to be gathered, i.e. a user uploading a copyrighted work, it could be argued that the original content author owns the newly produced content. It could, however, be argued that because the training process merely takes a note of which words are used in which order, it writes freely using a novel as a dictionary. I believe that it would be fair for a user to claim ownership of the text that the model produces, so long as proper accreditation is given to the training data. In the case that the user writes their own text to be analysed, this would be unnecessary. Due to the essentially random nature of text selection, it could be theoretically possible that the model may produce hate speech or other such unsavoury text. This, of course, cannot be protected against in any real sense, as the project will be open source and therefore could be edited by a user to remove any checks that I put into place.

The benefits of my work are mostly directed towards developers who may now have the opportunity to implement a random text engine where they otherwise may not have been able to due to hardware limitations on target systems or expense. Developers will be the most affected and due to the way that the library is implemented to be as customisable as possible, they are able to influence the work. Due to the fact I intend to publish the library as open source software, they can also access and edit the code without cost or permission granted.

In theory, were the algorithm to outperform other NLG programs such as ChatGPT, users may switch from such programs to use my algorithm. This would, in turn, reduce the cost of powering the other applications servers and this would have the impact of a slight reduction of the fossil fuels required. This would, in turn, benefit the environment. However, I see that this is an unlikely eventuality.

# 10. Summary and Reflections

## 10.1 Project Management

My initial work plan was organised into three phases, to help with planning my time. These sections were a research phase, development phase and a testing and refinement phase. Each phase was broken down further to include the individual tasks I had to complete to ensure that the project was successful. This information was then further collated into a Gantt Chart.

In my research phase, the tasks to complete were:

- Reading and understanding how other projects in the field of NLG work.
- Reading about the structure of sentences in the English language and producing a grammar for it.
- Reading and understanding probabilistic processes and how they apply to NLG.
- Selecting an appropriate process that I can utilise in my application.

In my development phase, the necessary tasks were:

- Implement the grammar production rules.
- Implement the generation algorithm, first with a placeholder random generation function.
- Use an appropriate method to generate words and tokens such that the word order and selection is natural.
- Wrap all of the functions in an object.

In my testing phase, it was imperative to:

- A study to assess and evaluate my model
- Export the code to a library and publish
- Develop some software demos that use the library functions.

This information was carefully considered and assisted in the production of a Gantt Chart which I intended to use to organise my time appropriately. The Gantt Chart was chosen as it is an effective way to organise my development and illustrate dependencies between tasks. Using the chart allowed me to keep on time and maintain a consistent development schedule. However, this initial Gantt chart wasn't useful as over the course of my research phase, I found better methods to complete the work and refine my methodology. The initial and updated charts are shown below, as figures 6 and 7 respectively:

Figure 6. Original Gantt Chart

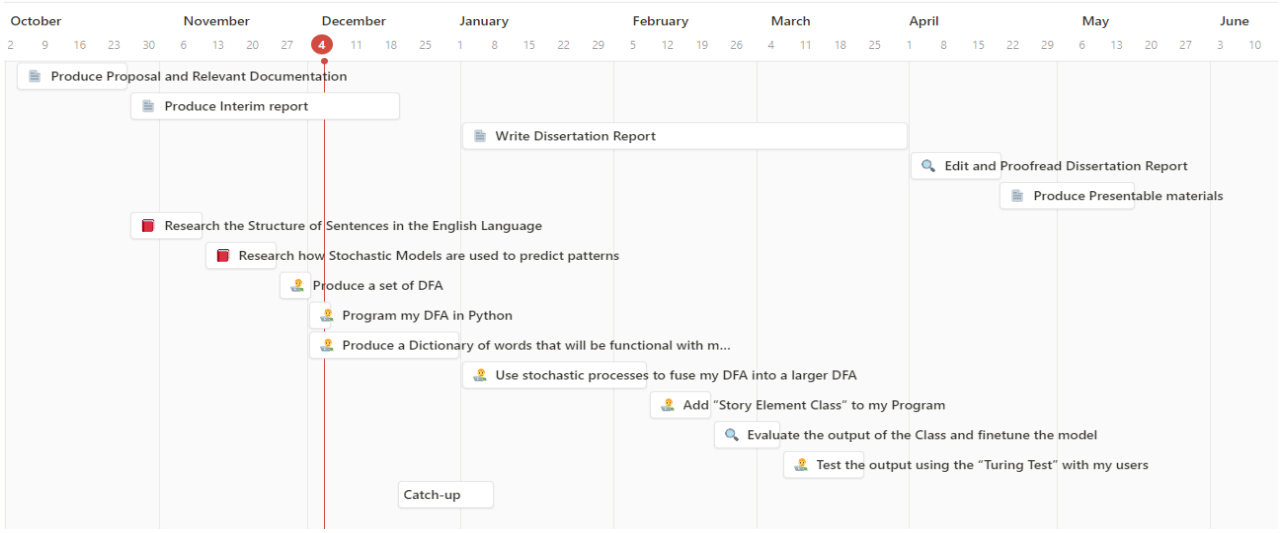
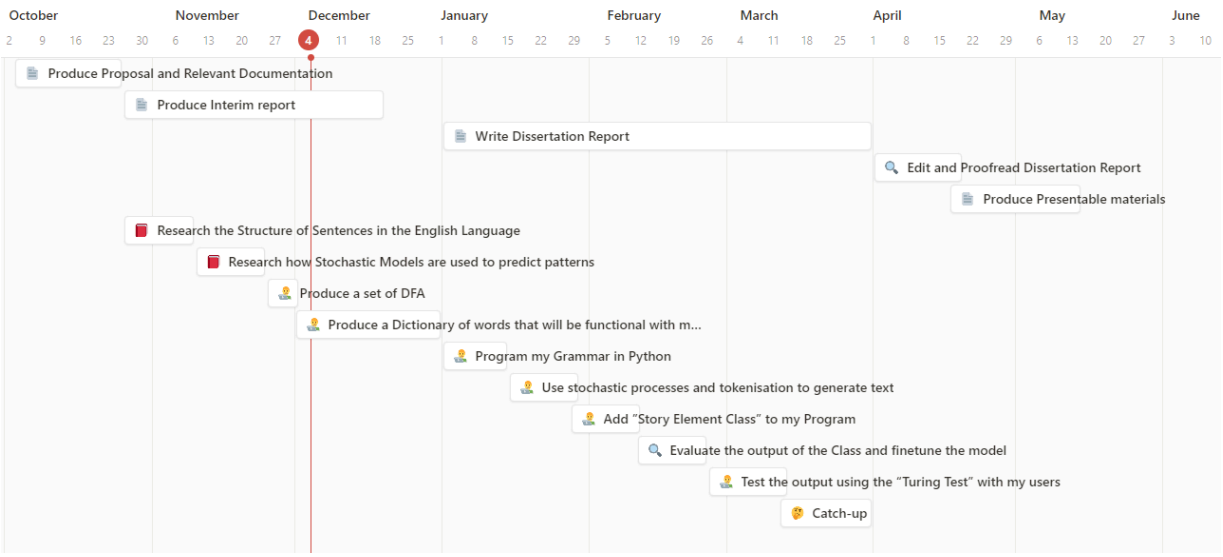


Figure 7. Revised Gantt Chart




To ensure that my tasks were all complete, I reallocated the time I had to match an estimate for the new lengths of the task, and considered the approximate time I should spend on the project regarding credits towards my degree. I found that this approach was appropriate as I was able to complete each task in the allotted time, however I believe that I have been generous with the time that I awarded myself for each task, as I found that I understood the concepts presented to me early into each stage of research. Being generous with the time for certain tasks afforded me more time later on allowed me to start other, more difficult tasks early.

To ensure successful development of my algorithm, I adapted a familiar two-week sprint cycle with a defined goal, and updated my Gantt chart to reflect this. An additional buffer sprint was incorporated to account for unforeseen delays, which was of great help. This meant that I was able to develop the model as well as document my progress to concurrently develop the model and document my progress.. The tasks in the Gantt chart are structured in a cascading dependency, ensuring each completed section informs the writing of the corresponding dissertation segment.

An issue with project management has been the format I have used. Whilst the Gantt Chart was useful to plan my work, I found it impractical for use in organising my time within the sprints. Increasing the granularity of the chart would have reduced its effectiveness. Consequently, I combined the plan made on my Gantt Chart with my notes diary to ensure that during the time period I have set myself for each task, I am able to focus and carefully plan my work around it. This meant that I would set time within each week at the start of the sprint and fill in my notes with what I achieved in that working period. I found that this was extremely helpful in helping me to write my final project report.

Unfortunately, over the course of the project I encountered issues with completing work due to complications in my private life. This prevented me from staying on my schedule as I planned, and I found myself falling behind. Thankfully, due to forethought and planning I was able to use the extra weeks allowed to complete and even make some final refinements to the project.

## 10.2 Contributions and Reflections

The development process of this project presented significant challenges. However, the project's final outcome and the knowledge I have gained from building it has ultimately justified the difficulty. My initial idea was ambitious, and this presented problems in the form of conflicts between my initial understanding of the project and the understanding gained after further research.

As previously mentioned, my initial plan involved employing a large DFA network. This approach would have proven demonstrably inefficient as it would have required me the programming of every possible transition in the English language, with each word linking to every other potential word, hence it was more efficient to construct my own grammar using the grammar object.

Another obstacle emerged when it became apparent that my proposed method aligned poorly with the theoretical framework outlined by Noam Chomsky. Despite this potential incompatibility, I adapted my approach and continued the research.

I believe that the work performed is worth pursuing, despite the response from my evaluation. I showed that the current method is only suitable for short texts, and only when there is an extensive amount of information to generate from. Ultimately, the result tells me that more context than simple word order is required shows that a collection of multiple parameters would be better suited to ANNs. Should the project be taken further, it must be adapted to take more inspiration from ANNs and will be compared to ANNs in the time and quality of text production. My current hypothesis to further the performance of the algorithm is that through improved POS tagging, relationships between the words in a sentence and its subject can be found and this can improve the types of words that are chosen. By using second or third order Markov Chains I could improve the quality of the sentences by honing in on the semantic field of the words used. My original hypothesis is that this kind of context management is intrinsic to each sentence and so should be reflected with the transitions between sentences. Unfortunately, due to lack of pattern recognition and data, the algorithm is unsuccessful. An improved version of the algorithm would use this data to build each sentence in order, but construct the individual sentences using a priority system for given tokens.

I believe that I have achieved my original project goals. I have determined that Markov Chains, when applied to formal grammars, are not entirely sufficient as a standalone method of Natural Language Generation. Further work is required to develop a method in which they can be used.

# 11. Bibliography


Strubell, E., Ganesh, A., & McCallum, A. (2020). Energy and Policy Considerations for Modern Deep Learning Research. Proceedings of the AAAI Conference on Artificial Intelligence, 34(09), Article 09. https://doi.org/10.1609/aaai.v34i09.7123

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., … Amodei, D. (2020). Language Models are Few-Shot Learners. Advances in Neural Information Processing Systems, 33, 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

OpenAI Platform. (n.d.). Retrieved December 7, 2023, from https://platform.openai.com

Menghani, G. (2023). Efficient Deep Learning: A Survey on Making Deep Learning Models Smaller, Faster, and Better. ACM Computing Surveys, 55(12), 259:1-259:37. https://doi.org/10.1145/3578938

Chomsky, N. (1957). Syntactic Structures. Mouton and Co., The Hague.

OpenAI. (2023). GPT-4 Technical Report (arXiv:2303.08774). arXiv. http://arxiv.org/abs/2303.08774

Compton, K., Kybartas, B., & Mateas, M. (2015). Tracery: An Author-Focused Generative Text Tool. In H. Schoenau-Fog, L. E. Bruni, S. Louchart, & S. Baceviciute (Eds.), Interactive Storytelling (pp. 154–161). Springer International Publishing. https://doi.org/10.1007/978-3-319-27036-4_14

A. Markov (1906). Extension of the law of large numbers to dependent quantities (in Russian). Izvestiia Fiz.-Matem. Obsch. Kazan Univ 15: 135–156.

A. M. Turing (1950). Computing Machinery and Intelligence. Mind, 49, 433-460.

Gardner, M., Grus, J., Neumann, M., Tafjord, O., Dasigi, P., Liu, N., Peters, M., Schmitz, M., & Zettlemoyer, L. (2018, March 20). AllenNLP: A Deep Semantic Natural Language Processing Platform. arXiv.Org. https://arxiv.org/abs/1803.07640v2

Modular (2023) Modular Docs – Mojo 🔥 programming manual.. Available at: https://docs.modular.com/mojo/programming-manual.html (Accessed 1st December 2023).

Kelmendi, L. (2020). English Language Sentences According to Structure (SSRN Scholarly Paper 3559057). https://doi.org/10.2139/ssrn.3559057

Loria S, TextBlob Documentation (n.d.). textblob.readthedocs.io/en/dev/

Vermeer, S., & Trilling, D. (2020). Toward a Better Understanding of News User Journeys: A Markov Chain Approach. Journalism Studies, 21(7), 879–894. https://doi.org/10.1080/1461670X.2020.1722958

Marcus, M. P., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: The Penn Treebank. Computational Linguistics, 19(2), 313–330

Frequency Data | Open American National Corpus. (n.d.). Retrieved March 14, 2024, from https://anc.org/data/anc-second-release/frequency-data/

NVIDIA (n.d.) Cuda Zone - Library of Resources; NVIDIA Developer. Retrieved March 14, 2024, from https://developer.nvidia.com/cuda-zone

Llama 2: Open Foundation and Fine-Tuned Chat Models | Research—AI at Meta. (n.d.). Retrieved March 25, 2024, from https://ai.meta.com/research/publications/llama-2-open-foundation-and-fine-tuned-chat-models/

Zehra, F., Javed, M., Khan, D., & Pasha, M. (2020). Comparative Analysis of C++ and Python in Terms of Memory and Time (2020120516). Preprints. https://doi.org/10.20944/preprints202012.0516.v1

Engelfriet, J., & Rozenberg, G. (1997). Node replacement graph grammars. In Handbook of Graph Grammars and Computing by Graph Transformation (pp. 1–94). WORLD SCIENTIFIC. https://doi.org/10.1142/9789812384720_0001

Martinez, A. R. (2012). Part-of-speech tagging. WIREs Computational Statistics, 4(1), 107–113. https://doi.org/10.1002/wics.195

Manning, Christopher D., Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. (2014). The Stanford CoreNLP Natural Language Processing Toolkit In Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pp. 55-60

Yan, S. Y. (1998). An Introduction To Formal Languages And Machine Computation. World Scientific.

Mahfuz, F. (2021). MARKOV CHAINS AND THEIR APPLICATIONS. Math Theses. https://scholarworks.uttyler.edu/math_grad/10

Vayadande, K., Sheth, P., Shelke, A., Patil, V., Shevate, S., & Sawakare, C. (2022). Simulation and Testing of Deterministic Finite Automata Machine. INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING, 10, 2022. https://doi.org/10.26438/ijcse/v10i1.1317

Kristina Toutanova and Christopher D. Manning. 2000. Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger. In Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC-2000), pp. 63-70.

Indu et al. (2016) International Journal of Recent Research Aspects ISSN: 2349-7688, Vol. 3, Issue 2, June 2016, pp. 62-64