



COMP390

2023/24

Write like classic Poets

Student Name: Shuimu Zeng

Student ID: 201677429

Supervisor Name: Prof. Paul Dunne

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Dedicated to my parents

Acknowledgements

I dedicate this project to my parents, whose unwavering support and sacrifices have been the driving force behind my academic journey. Their love, encouragement, and belief in me have been my greatest inspiration. This project is a tribute to their enduring faith and commitment to my success.

I would like to express my sincere gratitude to Prof. Paul Dunne , my project advisor, for his invaluable guidance and support throughout this journey. I am also thankful to my flatmates for their assistance and passion in evaluation of my project. Additionally, I extend my appreciation to my family and friends for their unwavering encouragement and understanding.



COMP390

2023/24

Write Like a Classic Poet

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Abstract

This research project delves into the application of stylometric analysis techniques within the realm of poetry generation. Drawing upon computational linguistics and machine learning methodologies, the study endeavors to emulate the stylistic nuances characteristic of renowned poets, employing not only n-grams but a broader array of analytical tools. Methodologically, the project utilizes n-grams and neural networks to discern syntax, rhythm, and meter from a curated corpus of poetic works, subsequently employing this analysis for poetry generation.

Central to the project's approach is the employment of deep learning models, particularly recurrent neural networks (RNNs), tasked with discerning the inherent patterns and structures present within poetic texts. Through iterative training and optimization, these models are honed to generate new poetry reflective of the stylistic traits identified within the original corpus. To ascertain the fidelity of the generated poetry, a qualitative analysis is undertaken by human evaluators, culminating in a comprehensive assessment of the algorithmically produced compositions.

The findings of this study underscore the viability and efficacy of utilizing stylometric analysis in the domain of poetry generation, yielding algorithmically generated verses that encapsulate the essence and aesthetic of the source poets. Furthermore, the project's dissertation serves to fulfill the requisite criteria set forth by the BCS Project Guidelines, accompanied by a comprehensive self-reflection.

In sum, this research contributes novel insights into algorithmic poetry generation, particularly within the realm of stylometric analysis, where a multifaceted approach encompassing n-grams and neural networks is employed to advance the field.

Statement of ethical compliance

Data Category: A

Participant Category: 2

I confirm that I have followed the ethical guidelines during this project. Further details can be found in the relevant sections of this dissertation.

Table of contents

Abstract-----	5
Statement of ethical compliance-----	5
Introduction&background-----	7
Design-----	8
Implementation-----	10
Testing&Evaluation-----	23
Project Ethics-----	26
Conclusion&Future work-----	27
BCS project Criteria & Self-reflection-----	27
Reference-----	29
Appendix-----	30

A table of figures

Chain rule of probability-----	11
Bi-grams-----	11
Sonnet example-----	12
Import word tokenize-----	12
Bi-gram implemtation-----	13
Sequence count-----	13

Visualization of sequence count-----	13
Dictionary-----	14
Probability count-----	14
Visualization of sequence probability--	14
Word sampling 1-----	15
Word sampling 2-----	15
Tri-grams-----	16
Tri-grams defaultdict-----	16
Random number-----	17
dictionary of character to integer and integer to character-----	17
next character-----	18
Vectorization-----	18
Training labels and targets-----	18
Character level neural network structure-----	19
Tokenizer fit-----	21
Sentence tokenizer -----	21
categorize the y-----	21
Optimizer-----	22
Word level neural network structure-----	22
testing during development-----	24
result from character level neural network---	26

Introduction&background

Driven by an enduring passion for linguistics, this research project holds significant allure as it amalgamates the realms of language and computer science. Within the expansive domain of computational linguistics lies a wealth of tools and methodologies primed for the analysis and comprehension of literary texts. Notably, stylometric analysis has emerged as a potent technique within literary studies, offering profound insights into writing style and authorship attribution. In this endeavor, we embark upon an exploration of stylometric analysis's application in the emulation of a classic poet's style, with two primary objectives:

1. produce a system building on the n-gram language model that generates verse in the style of a classical poet. Notably, this pursuit is characterized by an openness to explore alternative methodologies beyond the n-gram framework.
2. To identify deficiencies in the output and discuss/develop methods by which these can improved. Through a systematic and rigorous evaluation process, we endeavor to cultivate a nuanced understanding of the underlying factors contributing to the observed deficiencies, thereby paving the way for the refinement and optimization of our methodology.

Fictional narratives, such as Sherlock Holmes, have long hinted at the potential of stylometric analysis, as exemplified in "A Scandal in Bohemia." Fiction has recently become fact with the improving science of stylometry, the study of writing style[1]. Stylometric analysis entails the quantitative examination of linguistic attributes within texts to discern discernible patterns and stylistic traits. Prior research has underscored the efficacy of stylometric analysis across diverse domains, encompassing authorship attribution, genre classification, and literary emulation. Notably, software tools have been developed to emulate text in the vein of specific authors or literary movements, thereby illuminating the underlying structures and conventions of literary expression. Leveraging these tools, our project endeavors to navigate the intricacies of stylometric analysis to emulate the poetic style of classical authors.

Imitating the poetic style of classical authors poses formidable challenges, necessitating the adept capture of their linguistic intricacies, imagery, and thematic preoccupations. To mitigate these challenges, our focus will be directed toward specific works of renowned classical poets. Despite the inherent complexities, our project's impetus stems from a fervent interest in computational linguistics. By devising models capable of generating poetry reflective of classical poets' styles, we aspire to foster deeper engagement with their oeuvres while simultaneously exploring the multifaceted realm of computational linguistics.

The overarching objective of this endeavor is to emulate the literary output of classical poets, underpinned by the application of text analysis techniques, particularly stylometric analysis. To this end, an algorithm will be developed to analyze the works of classical poets, subsequently generating text based on the acquired language model. The resultant texts will exhibit stylistic congruence with the respective poet. Evaluation of the generated output will

be facilitated through human participant feedback, supplemented by the potential application of additional scientific methodologies for comprehensive assessment.

Design

The project encompasses two principal approaches: the utilization of n-grams, as stipulated, and the employment of neural networks, deemed as a viable methodology.

N-grams design:

N-grams have long served as a pragmatic method in stylometric analysis. An n-gram denotes a sequence of n words: a 2-gram (which we'll call bigram) is a two-word sequence of words like "I am", "am a", or "a student", and a 3-gram (a trigram) is a three-word sequence of words like "I am a", or "am a student". At present, we possess the latest advancements in language and environment conducive to the development of the N-grams model. Hence, the identification of pertinent dependencies is imperative for effective utilization. A robust data structure is essential for storing n-grams models and their corresponding analyses. As delineated below, the focus is directed toward two n-grams models, from which conclusions can be drawn:

1. Bi-gram model

2. Tri-gram model

Upon the construction of the n-grams models, the generation of sampled text assumes paramount importance. This task presents considerable challenges, as maintaining the syntactic integrity of verses proves daunting even for human poets. Moreover, due consideration is accorded to the rhyme scheme. The user interface is designed to be interactive, facilitating user input of words or phrases as tokens, whereupon the program generates subsequent corpus based on analytical data. Error handling mechanisms are incorporated to address triggered errors. Testing is integral throughout the project life-cycle, with refinements iteratively implemented and results scrutinized. Human participants are engaged in the final testing phase, providing feedback based on the outcomes yielded by the refined program.

1. Utilization of Python on Google Colab environment
2. Data collection from reliable sources
3. Preparation for capable data structure
4. Implementation of Bigrams and Trigrams models
5. Text Sampling methodologies
6. User interface design considerations
7. Test and Refinement
8. Comprehensive evaluation procedures

Neural network:

In recent years, the realm of deep learning, underpinned by neural network architectures, has witnessed substantial advancements, particularly within the domain of Natural Language Processing (NLP). Motivated by seminal models like ChatGPT, our project endeavors to leverage neural network methodologies. We opt to conduct our experiments within the Google Colab environment, owing to its Python 3.0 compatibility and facilitation of seamless dependency and framework integration. TensorFlow emerges as the primary framework driving our models, chosen for its robustness and versatility. We embark on the development of two distinct models based on differing granularities: character level and word level.

1. Utilization of Python on Google Colab as the development environment
2. Methodical data collection procedures
3. Utilization of the TensorFlow framework for model implementation
4. Preprocessing of data prior to model training
5. Development and evaluation of models at both character and word levels
6. Consideration of user interface design principles
7. Implementation of fine-tuning techniques for model optimization
8. Rigorous testing and comprehensive evaluation methodologies

Acknowledging the inherent limitations and challenges encountered during the project, we engage in substantive discussions to delineate these shortcomings and outline avenues for future research and development endeavors.

Implementation

Theory preparation:

The project under consideration places a significant emphasis on the n-grams model, prompting a comprehensive examination of its conceptual underpinnings. Initial scrutiny is directed towards the statistical language model, a fundamental construct within natural language processing frameworks. This model operates as a probabilistic apparatus, tasked with assessing the likelihood of encountering a particular word based on the contextual cues provided by preceding words. However, the process of probabilistic estimation of the preceding words poses notable challenges.

A conventional methodology entails tabulating the frequency of occurrence of preceding word sequences and subsequently computing the frequency with which these sequences are followed by the target word. While efficacious, particularly when applied to voluminous corpora, this approach is computationally intensive, rendering it impractical within the confines of the project's computational resources. Consequently, alternative methodologies for probabilistic estimation are sought.

One such alternative involves estimating the probability by enumerating the frequency of occurrence of the specific sequence of preceding words relative to the total number of possible sequences. This approach necessitates the computation of the count of the target sequence and its normalization against the exhaustive set of potential sequences. However, this method is encumbered by computational overheads, prompting the exploration of more parsimonious strategies.

In response to these exigencies, the concept of the chain rule is introduced. This principle, rooted in probability theory, facilitates the decomposition of the joint probability distribution of a sequence of words into a series of conditional probabilities. By leveraging the chain rule, the estimation of a word conditioned on a historical context of preceding words can be approached in a more nuanced and computationally tractable manner.

In light of the aforementioned considerations, there arises a compelling imperative to explore more sophisticated methodologies for the estimation of a word within the contextual framework provided by preceding words. Consequently, the introduction of the chain rule emerges as a judicious recourse within this scholarly pursuit.

$$\begin{aligned}
 P(X_1, X_2, \dots, X_n) &= P(X_1)P(X_2|X_1)P(X_3|X_1, X_2) \dots \\
 &= \prod_{i=1}^n P(X_i|X_1, \dots, X_{i-1})
 \end{aligned}$$

Figure: Chain rule of probability

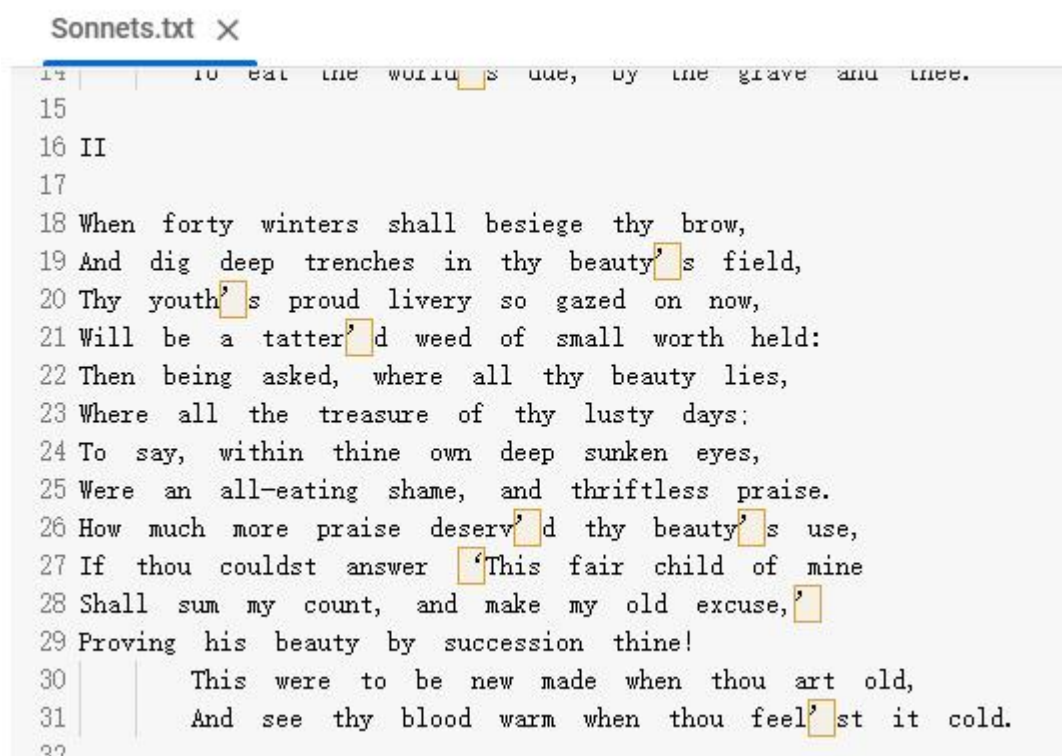
The chain rule of probability links the entire sequence and produces a final probability based on every component. However, problems have emerged that languages are too creative to be analyzed using long strings. To avoid too much computation and to reach achievable analysis with small corpus, n-grams model has risen in our mind. The intuition of the n-gram model is that instead of computing the probability of a word given its entire history, we can approximate the history by just the last few words.(Speech and Language processing).The Markov assumption indicates that we can estimate a new word only depends on the word that it follows. [2]

$$\begin{array}{c}
 \text{next state} \\
 \hline
 P(q_i = a \mid q_1 \dots q_{i-1}) = P(q_i = a \mid q_{i-1}) \\
 \hline
 \text{history of states} \\
 \text{traversed} \qquad \qquad \qquad \text{current} \\
 \qquad \qquad \qquad \qquad \qquad \text{state}
 \end{array}$$

Figure: bi-grams

Based on the assumption, we can build our bi-grams model. in a comparable manner , we can produce 3-grams model and even models with greater n values.

Accessing the literary oeuvre of classic poets is facilitated by repositories such as Project Gutenberg, thereby affording ready availability of works. In the initial phases of implementation, due diligence in data preprocessing is imperative, whether in its entirety or selectively. For the purposes of this endeavor, the corpus of Shakespearean Sonnets has been selected as the focal point for attainment and primary source for subsequent analysis. Subsequently, a comprehensive examination of the acquired textual data is conducted.



```
Sonnets.txt X
14 |         To eat the world's due, by the grave and thee.
15 |
16 | II
17 |
18 | When forty winters shall besiege thy brow,
19 | And dig deep trenches in thy beauty's field,
20 | Thy youth's proud livery so gazed on now,
21 | Will be a tatter'd weed of small worth held:
22 | Then being asked, where all thy beauty lies,
23 | Where all the treasure of thy lusty days:
24 | To say, within thine own deep sunken eyes,
25 | Were an all-eating shame, and thriftless praise.
26 | How much more praise deserv'd thy beauty's use,
27 | If thou couldst answer 'This fair child of mine
28 | Shall sum my count, and make my old excuse,'
29 | Proving his beauty by succession thine!
30 |         This were to be new made when thou art old,
31 |         And see thy blood warm when thou feel'st it cold.
32 |
```

Figure: Sonnet example

The corpus is sourced directly from Sonnet 116, as retrieved from the Project Gutenberg website. Notably, the original syntax entails the utilization of Roman numerals preceding each sonnet. Given their marginal significance in contributing to the overall semantic content of the sonnets, a decision was made to expunge these numerals from the corpus as a preliminary step.

For tokenization and series of other natural language processing functions, we import natural language toolkit [3]. We introduce word_tokenize to make tokenization of our text in string format.

```
from nltk.tokenize import word_tokenize [3]
```

Figure: Import word_tokenize

Subsequently, we employ a function, as depicted in the following snippet, to generate word pairs, commonly referred to as bigrams, from a given list of words. Upon attempting to visualize the resulting bigram structure, it becomes evident that direct printing or visualization is unfeasible. Fortunately, a viable workaround is available: the transformation of the bigram structure into a list format. This transformation entails the conversion of each element within the bigram structure into a tuple structure encapsulated within a list.

```
from nltk.util import bigrams [3]
```

```
[('From', 'fairest'), ('fairest', 'creatures'), ('creatures', 'we'),...('not', 'love')]
```

Figure: bigram implementation

Upon obtaining the bigram list, the subsequent phase entails deliberation on the requisite data structure for analysis and data storage. To succinctly encapsulate this requirement, a dictionary-like structure is deemed necessary, wherein a string serves as the key and other dictionary structures serve as the corresponding values. Following a comprehensive review of pertinent references[4], the defaultdict data structure emerges as a judicious selection. In initializing the analysis, the structure is configured to instantiate a dictionary whereby the default value for any absent key is an empty dictionary. The nested dictionary specified within defaultdict(dict) operates as the default factory or data type for the values contained therein.

```
from_token_to_next_token_counts = defaultdict(dict) [4]
```

To enumerate the occurrences of each bigram sequence, it is necessary to iterate through the list of bigrams and implement a conditional statement, as illustrated below:

```
if next_token not in from_token_to_next_token_counts[token]:
    from_token_to_next_token_counts[token][next_token] = 0
    from_token_to_next_token_counts[token][next_token] += 1
```

Figure: sequence count

When scrutinizing the default dictionary structure, a similar obstacle arises where direct printing of the structure is not feasible. To visualize the default dictionary, appending the items() method is necessary. Subsequently, we can inspect the structure accordingly.

```
dict_items([('From', {'fairest': 1, 'his': 1, 'sullen': 1, 'limits': 1, 'hands': 1, 'whence': 1, 'where': 1, 'me': 1, 'this': 1, 'thy': 1, 'hence': 1, 'thee': 1, 'you': 1, 'heaven': 1})])
```

Figure: Visualization of sequence count

Now, we have the count for all bigram sequences at hand. The next step is more challenging that we need to count the probability all every sequences. Thanks to the developers of defaultdict, we are able to utilize some built in functions to help. To easily account for the total number of bigram sequences given a starting word, we can use values() suffix because the number of each bigram sequence is count in a dictionary format.

```
{'fairest': 1, 'his': 1, 'sullen': 1, 'limits': 1, 'hands': 1, 'whence': 1, 'where': 1, 'me': 1, 'this': 1, 'thy': 1, 'hence': 1, 'thee': 1, 'you': 1, 'heaven': 1}
```

Figure: Dictionary

An essential function, sum(), is employed to aggregate the values within the dictionary, consequently computing the total count of sequences associated with a specific starting word. Subsequently, it becomes imperative to iteratively traverse each dictionary nested within every key-value pair, subsequently transforming these values into a dictionary representing probabilities.

```
from_token_to_next_token_probs[token] = {
    next_token: count / sum_of_counts_for_token
    for next_token, count
    in d_token.items()
}
```

```
sum_of_counts_for_token = sum(d_token.values())
from_token_to_next_token_probs[token] = {
    next_token: count / sum_of_counts_for_token
    for next_token, count
    in d_token.items()
}
```

Figure: Probability count

We examine the data after a visualization of the structure and go to the sampling part which is tricky as well.

```
dict_items([('From', {'fairest': 0.07142857142857142, 'his': 0.07142857142857142})])
```

Figure: Visualization of sequence probability

The initial strategy for sampling the next word based on a history of words involves extracting the last word in the history and utilizing it to access the corresponding probabilities from the from_token_to_next_token_probs structure. Subsequently, this

information is encapsulated into a list, where each element comprises a word and its associated probability. To conduct a rudimentary test of word sampling, the next word is chosen randomly, without giving significant weight to its probability.

```
next_tokens = list(zip(from_token_to_next_token_probs['From'].items()))
```

```
next_token_sampled = np.random.choice(next_tokens, size=1)
```

Figure: Word sampling 1

At first, we tried to list the zipped itemized structure. It shows no errors and produces a two dimensional datasets. Problem has come when we try to use `np.random.choice()` function. It says that the parameter in the function needs to be one dimensional. So , we tend to use the asterisk (*) to unpack the items of the dictionary `from_token_to_next_token_probs['From'].items()` into separate arguments and assign the two arguments into to variables.

```
next_tokens,next_tokens_probs=list(zip(*from_token_to_next_token_probs['From'].items()))
next_token_sampled = np.random.choice(next_tokens, size=1, p=next_tokens_probs)
```

Figure: Word sampling 2

Consequently, random word sampling was implemented. To generate a corpus resembling a Sonnet, a predetermined number of words is required. Hence, a for loop and a word count variable were incorporated into the text generation process. Finally, the results are examined.

Result example: See appendix 1.

The trigram model can be constructed in a manner analogous to the bi-gram model. However, in contrast to the bi-gram model, it is proposed to retain the delimiters. This decision stems from the observation that the trigram model constitutes a more robust chain, which can effectively regulate the behavior of delimiters, thereby enhancing model performance.

$$\text{Unigram LM : } p(w_1^N) = \prod_{n=1}^N p(w_n)$$

$$\text{Bigram LM : } p(w_1^N) = \prod_{n=1}^N p(w_n | w_{n-1})$$

$$\text{Trigram LM : } p(w_1^N) = \prod_{n=1}^N p(w_n | w_{n-2}, w_{n-1})$$

Figure: tri-grams

When introducing the defaultdict, we need slightly change it and make the key to be a tuple.

```
for trigram in trigrams:
    trigram_model[(trigram[0], trigram[1])][trigram[2]] += 1
```

Figure: tri-grams defaultdict

We then count for probabilities for each tri-grams pair in a similar manner as what we do with the bi-grams pairs.

During sampling, we choose to pick randomly from the probabilities. This does show a noisy result. To improve the scene, we pick the most common results for every sequence every time. To make the result in a certain syntax, we separate the string at every delimiter

It's clear to see that the corpus is not in the syntax of a Sonnet. We have now encountered one of the biggest issues. Other problems have also shown their existence. Such as , the delimiters are not acting correctly and the rhyme scheme is not maintained as well. More for this, the verses are in a weak grammar accuracy. We are now breaking this discussion into several parts.

Subsequently, probabilities are computed for each tri-grams pair in a manner akin to that employed for bi-grams pairs.

During the sampling process, random selection from the probabilities is initially employed, resulting in a somewhat noisy outcome. To ameliorate this issue, a strategy is adopted wherein the most common results for each sequence are consistently chosen. Furthermore, to enforce a specific syntax, strings are partitioned at each delimiter.

It becomes evident that the corpus does not adhere to the syntax of a Sonnet, highlighting one of the primary concerns. Additionally, other issues have emerged, including improper delimiter behavior, lack of maintenance of the rhyme scheme, and weak grammar accuracy within the verses. Consequently, the discussion is subdivided into several parts to address these issues comprehensively.

For syntax maintenance, a brute force method is initially considered. This method entails counting the syllable number for each line and transitioning to a new line once the syllable

limit is reached. However, this approach presents limitations, such as the possibility of encountering single-syllable words when only one syllable is needed, potentially leading to difficulties in word selection. Moreover, the brute force method tends to fragment sentences, hindering readability. To enhance performance, attention is directed towards text preprocessing.

We decided to remove punctuation marks, but keep the ones that have an impact on semantics. At the same time, we also innovate on text sampling. When sampling for a word we map the probability of every sequence into a structure in a descendant order. Then we use a random unit to generate a random number between 0 and 1.

```
random_number = random.random() [9]
```

Figure: random number

To decide which word to choose, we loop through the structure and accumulate every probability first and compare the random number with the accumulation. Once the accumulation goes beyond the random number, the words whose probability is added at last will be the sampled word. By using this method for word selection, the correlation between the words of the resulting text is significantly strengthened.

Neural network:

Character-level neural networks are suitable for tasks requiring fine-grained text generation and analysis, while word-level neural networks are well-suited for tasks requiring semantic understanding and language generation at the word level.

Character level:

The implementation process is divided into several key stages, including data preparation, model architecture design, training, and text generation. The dataset used for training the text generator consists of Shakespearean sonnets extracted from the file "Sonnets.txt". The raw text data is loaded and converted to lowercase to ensure uniformity in character representation. Each sonnet is split into individual characters, and unique characters are mapped to numerical indices using dictionaries. This tokenization process prepares the data for input into the neural network model.

```
{'\n': 0, ' ': 1, '!': 2} [5]
```

```
{0: '\n', 1: ' ', 2: '!'} [5]
```

Figure: dictionary of character to integer and integer to character

The tokenization is just the beginning of our data processing. We are now going to prepare training data from the corpus. We first need to generate an organized corpus list which serves an intermediate state. The organized corpus was implemented using a sliding window approach to extract sequences of fixed length from the raw text corpus. With a specified maximum length and a step size, the algorithm iterated over the characters of the corpus, extracting sequences at regular intervals. These sequences, along with the subsequent character following each sequence, were stored in separate lists.

```
maxlen = 40
step = 3 #the step indicates how many characters to skip every time like from -> om which
means to skip two charaters
sentences = []
next_chars = []
for i in range(0, n_chars-maxlen, step):
    sentences.append(raw_text[i:i+maxlen])
    next_chars.append(raw_text[i+maxlen])
```

Figure: next character

This methodology ensured that the model had a diverse range of input sequences to learn from, enabling it to capture complex patterns and dependencies within the text data. Subsequently, the model could generate coherent and contextually relevant text sequences based on the learned patterns. To attain usable data for training. We are now heading into vectorization part.

In the vectorization stage, the input data was encoded into a suitable format for training the neural network model. This involved converting the raw text sequences into binary matrices using one-hot encoding. A binary matrix \mathbf{x} of shape `(num_sentences, maxlen, num_unique_chars)` was initialized, where `num_sentences` represents the total number of input sentences, `maxlen` denotes the maximum length of each sequence, and `num_unique_chars` signifies the total number of unique characters in the corpus. Similarly, a binary matrix \mathbf{y} of shape `(num_sentences, num_unique_chars)` was created to represent the target output.

```
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=bool)
y = np.zeros((len(sentences), len(chars)), dtype=bool)
```

Figure: vectorization

For each sentence in the dataset, the corresponding binary matrix entries were set to 1 based on the presence of characters. This was achieved by iterating over each character in the sentence and setting the corresponding entry in the binary matrix to 1.

```
for i,sentence in enumerate(sentences):
    # i is the index of every sentence in the sentences list,
    #snetence is every sentence in the sentences list
```

```

for t, char in enumerate(sentence):
    x[i,t,char_to_int[char]] = 1
    #the x represents: i index of every sentence , t index of every character,
    char_to_int[char]
    # is the corresponding value for every character in the sentence
    y[i,char_to_int[next_chars[i]]] = 1
    # i is explained, char_to_int[next_chars[i]] is the value of next character given the current
    sentence

```

Figure: training labels and targets

'i' represents the index of each sentence in the dataset, t denotes the index of each character within the sentence, and char_to_int[char] retrieves the integer representation of the character based on the pre-defined character-to-integer mapping. Finally, the target output y was encoded in a similar manner, with the value corresponding to the next character set to 1 for each sentence.

The successful vectorization marks to end of data preparation. We now head into the neural network structure design. In this stage, a Long Short-Term Memory (LSTM) neural network model was constructed using the Keras library(Tensorflow Keras)[6]. The model architecture consisted of a sequential stack of layers.

An LSTM layer with 128 units was added to the model, specifying the input shape as (maxlen, len(chars)), where maxlen represents the maximum length of each sequence and len(chars) denotes the total number of unique characters in the corpus.

Subsequently, a dense layer with softmax activation function was added to the model to produce the output probabilities over the characters.

The structure set up is done and to optimize the training , an Optimizer needs to be introduced. The RMSprop optimizer with a learning rate of 0.01 was utilized, and the model was compiled using categorical cross-entropy as the loss function.

According to our test principle, we need to visualize the structure of the network. Thanks to the built in functions, we make it easily.

```
model.summary() [7]
```

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		

lstm (LSTM)	(None, 128)	86528
-------------	-------------	-------

dense (Dense)	(None, 40)	5160
---------------	------------	------

=====

Total params: 91688 (358.16 KB)

Trainable params: 91688 (358.16 KB)

Non-trainable params: 0 (0.00 Byte)

Figure: Character level neural network structure

Finally, a summary of the model architecture, including the number of parameters in each layer, was displayed.

The training begins after the creation of datasets. The model is trained on the preprocessed data using a batch size of 128 and for 50 epochs. During training, input-output pairs are generated by sliding a window of fixed length (40 characters) over the sonnets with a step size of 3 characters. The model learns to predict the next character in the sequence given the previous characters. The training process aims to minimize the cross-entropy loss between the predicted and actual next characters. After training, the trained model is used to generate new Shakespearean sonnets. A seed sequence of characters is provided as input to the model, and the model predicts the next character based on the learned probabilities. The predicted character is appended to the input sequence, and the process is repeated iteratively to generate text of the desired length. Temperature sampling is applied during text generation to control the diversity of the generated text.

Word level:

The word level neural network implementation shows great difference than character level network but the two remain similarities. We have to encounter new challenges and try new approaches.

We outline the process of preparing text data using the Tokenizer class from the tensorflow.keras.preprocessing.text module in TensorFlow. The implementation includes initializing the tokenizer object and fitting it on the provided text data. First, we initialize a tokenizer object using the Tokenizer[7]. This creates an instance of the Tokenizer class, which will be used to tokenize and preprocess text data for further analysis or modeling.

Next, we fit the tokenizer on the provided text data. This step involves tokenizing the input text data into individual words or tokens and building the vocabulary based on the unique words present in the text:

```
tokenizer.fit_on_texts([text]) [7]
```

Figure: Tokenizer fit

The 'text' represents the text data that we want to preprocess. The `fit_on_texts()` method tokenizes the input text data, builds its vocabulary, and updates the internal state of the tokenizer based on the provided text. Each unique word in the text is assigned a numerical index, and the tokenizer maintains a mapping between words and their numerical indices.

With the tokenized corpus at hand, we can do further data processing at ease. We are now beginning to vectorize the string data. We iterate over each sentence in the provided text data, which is split into individual sentences using the newline character ('\n'). For each sentence, the `texts_to_sequences()` method of the tokenizer is used to tokenize the sentence into numerical indices representing each word.

```
seq=[]
for sentence in faqs.split('\n'):
    tkn_sent=tokenizer.texts_to_sequences([sentence])[0]
    for i in range(1,len(tkn_sent)):
        seq.append(tkn_sent[i:i+1])
        #here we loop through the length of every tokenized sentence
        #then we make a seq which is a list that contains every subset whose length is bigger
        # for every tokenized sentence
```

Figure: Sentence tokenizer

The sentences are vectorized into arrays in different length and we got to have a uniformed length for further analyzation. Thus, we introduce `pad_sequences` dependency to pad sentence with zeros for arrays that are shorter than the max array length. Inside the padded sequence, we need to split the training lables and targets into x and y datasets. This is an easy task that we only need to cut off the final column and assign it to a y variable and assign the remains into a x variable. Last part for the data preparation, we introduce `to_categorical` that converts the target labels y to one-hot encoded vectors, where each vector represents the class labels in a binary format suitable for multi-class classification tasks.

```
from tensorflow.keras.utils import to_categorical [7]
y=to_categorical(y,num_classes=3201)
```

Figure: categorize the y

With datasets at hand, we now start to set up the neural network. The model architecture consists of an embedding layer followed by a Long Short-Term Memory (LSTM) layer and a dense layer. The neural network model is implemented using the Sequential API from TensorFlow Keras. The model architecture comprises the following layers: The first layer of the model is an embedding layer, which is responsible for converting input sequences of word indices into dense numerical vectors of fixed size. In this implementation, the embedding layer has an input dimension of 3201 (vocabulary size) and an output dimension of 200. Each word index in the input sequence is mapped to a dense vector representation. The final layer of the model is a dense layer with a softmax activation function. This layer consists of 3201 units, which corresponds to the vocabulary size. The softmax activation function computes the probability distribution over the vocabulary, producing the likelihood of each word in the vocabulary being the next word in the sequence. This layer outputs the probability scores for predicting the next word.

The optimizer looks quite similar to the one in character level model.

```
model.compile(loss='categorical_crossentropy',optimizer='adam',metrics=['accuracy']) [8]
Figure: optimizer
```

As our testing principle, we visualize the structure of the network.

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
embedding (Embedding)	(None, 10, 200)	640200
lstm (LSTM)	(None, 350)	771400
dense (Dense)	(None, 3201)	1123551
=====		
Total params: 2535151 (9.67 MB)		
Trainable params: 2535151 (9.67 MB)		

Non-trainable params: 0 (0.00 Byte)

Figure: Word level neural network structure

The text generation process iterates over a fixed number of iterations which can be customized (100 in this case) and predicts the next word in the sequence based on the previous words. It starts with a seed text ("fairest" in this case) and continuously appends the predicted words to generate a sequence of text. The prediction is made using the trained word-level text generator model, which predicts the index of the next word based on the context provided by the seed text. Finally, the predicted word is retrieved from the tokenizer's vocabulary and appended to the seed text, continuing the generation process.

By following this implementation, we can generate text using a word-level text generator model, allowing for the creation of coherent and contextually relevant sequences of words based on a given seed text.

Example: see appendix 2.

The result of word level network is clearly better than the character level network, but it is still in a poor syntax.

Testing&Evaluation

Testing has been integral from the inception of the development process. With each new manipulation, the entire program undergoes rigorous testing to evaluate its overall efficacy. This includes testing different data structures and implementations to ensure optimal performance.

In the context of n-grams development, testing occurs following the implementation of new decisions. This encompasses the introduction of new data structures as well as the examination of built-in functionalities within established data structures and imported dependencies. For instance, upon integrating the defaultdict, a comprehensive visualization of its internal data is conducted to assess its functionality.

```
print(from_token_to_next_token_counts.items())
print(type(from_token_to_next_token_counts.items()))
print(list(from_token_to_next_token_counts["I"].items()))
```

During the sampling session, we conduct tests using a limited range of word numbers for sampling purposes. This involves printing and analyzing every value that influences the selection of the subsequent word.

```
next_tokens, next_tokens_probs =  
list(zip(*from_token_to_next_token_probs['From'].items()))#we unpack here  
dict = {}  
for i in range(len(next_tokens)):  
    dict[next_tokens[i]] = next_tokens_probs[i]  
    sorted_dict = sorted(dict.items(), key=lambda item: item[1], reverse=True)  
    dict_sorted = {}  
    for pair in sorted_dict:  
        dict_sorted[pair[0]] = float(pair[1])  
accumulation = 0  
random_number = random.random()  
print(dict_sorted)  
for key,value in dict_sorted.items():  
    accumulation = accumulation+value  
    if random_number < accumulation:  
        next_token = key  
        break  
print(random_number)  
print(accumulation)  
print(dict_sorted[next_token])  
next_token
```

Figure: testing during development

When generating the whole corpus, we collect different output from every version of the program and evaluate them by the same participants. Participants will be acknowledged about the results from different versions , thus we can have feedback that can help to enhance performance.

Version 1: see appendix 3.

Version 2: see appendix 3.

I chose my flatmates as beta testers, not just because they were supportive, but because, well, some of them have a literary background and some of them don't. They can provide valuable feedback on different aspects. They will type words and click buttons to initialize the model if they want as these are not the essential task for them.


The first version is presented to them after the results are out. After my roommates' reading, they agreed that the generated text did have a poetic style, and when they compared it to the original Shakespearean poem, they clearly felt the similarities between the diction and Shakespeare's. At the same time they felt that even though the generated

verses, were not entirely grammatically accurate and in a freer syntax. However the verses embodied the theme, as a whole, very well. Those who have a literary background tell that the rhyme scheme is not well maintained or not in correct rhyme scheme at all. They also told me that the punctuation markers are so in disorder for bigrams model. Thus, the removal of delimiters in bigrams model is processed. For trigrams ,they consider the delimiter are in a order. So we did not remove delimiters in trigrams model. For neural network models, they clear stated that the character level is doing really bad that it generates words that are not English.

As far as we know the n-grams model can not understand the semantic well. We apply other methodology to enhance the syntax presentation. For bigrams , we use every capitalized word as the beginning of a new verse. When the word is capitalized, a new line starts and the word becomes the initial word. For trigrams, every delimiter with a punctuation sense marks the end of the verse and a new line will follow. With this change, we have conducted a new test. All participants reported a significant increase in the readability and formatting of the verses. However , the rhyme scheme remains mostly the same and has no significant improvement.

The testing of the neural network model was slow, as the updating of the hyperparameters of the neural network took time to deliberate. In addition to adjusting the hyperparameters, we can also change the input structure of the data, however the input structure is abandoned for modification due to the lack of a clear direction of the effect.

Temperature is an important hyperparameter for letter-level neural networks, however, after exhausting the range from 0 to 1, which is the range of this hyperparameter, the model still did not perform well. Participants all commented that the generated text was very cluttered, but maintained a nice format.



mime.
 lxxxviii

 loo you love thee, of you are you troull love hild,
 that mine is toon to hold thy praise.

 lxx

 the sicpering with as truth bring woodd,
 when my bod the excest, and i evernce approve sheet word,
 thy youthere his grace which thoughts all will.

 xxv

 then lof your some i love that so reep wound:
 thy had my love should parts of thy dear with withit
 det but you some glory thy dear tike,
 though i made be be tume doth find wow,
 or those varts thy poor doth with submery.

 lxx

 then some from her have thy sweet live younds
 for if thou dart which the surm' dess an ellight
 and you love th

Figure: result from character level neural network

For the word-level neural network model, participants still noticed the disruption of syntax, but they all noticed an improvement in both the rhyme scheme and the metres of the text. However, these improvements were still far from the human level.

To summarise, the n-grams model and the neural network model, have a good representation of both the diction and themes of Shakespeare's Sonnets, however, there is a lack of control over the meters and rhyme scheme. In conclusion, the rhyme scheme has become the hardest issue to overcome. This is because the need for semantic understanding of the rhyme scheme is very high. In the future I would like to apply big language modelling to train Shakespeare's text. Since the datasets we used are all from the original electronic text, they lack the required format for training. In the future, I hope that we will be able to use better datasets!

Project&Ethics

In the N-grams text emulation project, the involvement of human participants plays a pivotal role in the evaluation process, where their assessments and critiques of the generated outputs are integral. Participants can decide freely whether to select the initial token word or sentence from which the emulation process commences or just evaluate the

result. Their task entails a meticulous examination of the generated texts vis-à-vis the original author being emulated. The anonymity of participants is rigorously maintained, unless explicit consent is obtained for divulging their identities, particularly in circumstances necessitating the substantiation of evaluative rigor, such as the verification of expertise in English literature.

Conclusion&Future work

In conclusion, while the n-gram model is a valuable tool for many natural language processing tasks, it falls short of perfectly imitating poetry works due to several inherent limitations. Poetry, with its rich use of figurative language, creative expression, and nuanced meanings, presents a complex challenge for any computational model. The n-gram model's reliance on statistical patterns and lack of semantic understanding hinder its ability to capture the depth, originality, and emotional resonance that characterize authentic poetry. Additionally, poetry's subjective nature, reliance on literary devices, and intangible qualities further complicate the task of generating poetic text using traditional language models. While the n-gram model may produce text that superficially resembles poetry, it ultimately struggles to capture the essence of poetry's creativity, imagination, and artistic expression. To truly appreciate and understand poetry, we must turn to the human capacity for creativity, interpretation, and emotional connection, which cannot be fully replicated by computational models alone. Though the character level neural network performs the worst of all the models at hand, it shows great potential for it has shown the best syntax. Due to the lack of computational power and proper datasets, we can not exhaust the performance of character level RNN.

In the future, we want to apply large language model and larger more formatted datasets. Thus, our new model can learn to understand and generate coherent and contextually relevant text. By leveraging deep learning techniques, they can capture intricate patterns and nuances of language, producing output that often mimics human writing styles. So, we can have a more accurate handle of the meters , rhyme schemes and context. Despite these challenges, the development and refinement of large language models continue to drive innovation in artificial intelligence and reshape how we interact with and generate textual content.

BCS project Criteria & Self-reflection

Meeting BCS Project Criteria:

Throughout my dissertation, I have demonstrated the application of practical and analytical skills acquired during my degree program in computer science. For example, in the implementation phase of my project, I utilized advanced programming techniques and statistical analysis to conduct stylometric analysis on classic poetry texts.

The project exhibits innovation through the application utilization of data structures, sampling techniques and theoretical analysis on the literary aspect of classic works. By

leveraging machine learning algorithms and linguistic features, I creatively maintain proper syntax base on semantic analysis and successfully control the theme of Shakespearean Sonnets.

My dissertation integrates diverse sources of information, including literary theory, computational linguistics, and machine learning, to develop a robust solution for stylometric analysis. Moreover, I conducted a comprehensive evaluation of the proposed methodology, critically assessing its effectiveness in identifying authorship patterns and discussing its implications for literary scholarship.

By applying stylometric analysis to classic poetry, my project indicates the potential for text analysis in the digital age. Stylometric analysis is shown valuable for understanding language usage and behavior, with applications across disciplines like linguistics, literature, forensic science, and computational social science.

Self-Reflection:

In managing this project, I have demonstrated effective self-management skills by:

Setting clear project objectives and timelines, facilitating efficient progress tracking and task prioritization.

Flexibly adapting to evolving project requirements and methodological challenges, demonstrating resilience and problem-solving abilities.

Collaborating with academic advisors and domain experts to refine research methodologies and validate findings, enhancing the credibility and rigor of the study.

However, I also recognize areas for personal and professional growth, including:

Enhancing my proficiency in statistical analysis and machine learning techniques, particularly in the context of natural language processing and text mining.

Improving my communication and presentation skills to effectively convey complex technical concepts and research findings to diverse audiences.

Cultivating a deeper understanding of the ethical considerations inherent in digital humanities research, including issues related to data privacy.

Overall, this project has been a transformative learning experience, enabling me to bridge the gap between computer science and literary studies while fostering critical thinking and interdisciplinary collaboration. Moving forward, I am committed to continuous self-improvement and lifelong learning in pursuit of academic excellence and professional fulfillment.

References

- [1] P.Juola. " How a Computer Program Helped Show J.K. Rowling write A Cuckoo's Calling" Available: [How a Computer Program Helped Reveal J. K. Rowling as Author of A Cuckoo's Calling | Scientific American](#). [Accessed: May 5, 2024].
- [2] D. Jurafsky and J. H. Martin, *Speech and Language Processing*, 3rd ed. Upper Saddle River, NJ: Pearson Education, Inc., 2020,pp31.
- [3] NLTK Project, "Natural Language Toolkit," Natural Language Toolkit, [Online]. Available: <https://www.nltk.org/>. [Accessed: May 6, 2024].
- [4] Python Software Foundation, "collections — Container datatypes," Python 3 Documentation, [Online]. Available: <https://docs.python.org/3/library/collections.html#defaultdict-objects>. [Accessed: May 7, 2024].
- [5] TensorFlow, "Text generation with an RNN," TensorFlow, [Online]. Available: https://www.tensorflow.org/text/tutorials/text_generation. [Accessed: May 8, 2024].
- [6] TensorFlow Authors, "Get started with TensorFlow," TensorFlow Documentation, Version 2.7.0, 2023. [Online]. Available: <https://www.tensorflow.org/tutorials>. [Accessed: April 11, 2024].
- [7] TensorFlow, "Keras: The Python Deep Learning Library," TensorFlow, [Online]. Available: <https://www.tensorflow.org/guide/keras>. [Accessed: May 7, 2024].
- [8] Keras, "Optimizers - Keras Documentation," Keras, [Online]. Available: <https://keras.io/api/optimizers/>. [Accessed: May 7, 2024].
- [9] W3Schools, "Python random.random() Method," W3Schools, [Online]. Available: https://www.w3schools.com/python/ref_random_random.asp. [Accessed: May 8, 2024].

Appendices



Department of Computer Science
Ashton Street, Liverpool, L69 3BX

Computer Science Project Modules Participant Consent Form

Project Title: Write like classic poets
Student: XXX
Supervisor: XXX
Coordinator: Dr Stuart Thomason (s.thomason@liverpool.ac.uk)

Instructions to student: Mark the boxes that apply to your project so the participant knows what you will be asking them to do. If you later decide to ask the participant to do something that isn't marked, fill in a new form.

You are inviting the participant to take part in... (Copy/paste the ☒ symbol where necessary)

☐ Requirements Analysis ☒ Software Evaluation

You plan to use the following methods as part of this activity...

☒ Interview ☐ Discussion ☐ Observation
☐ Narration ☐ Questionnaire

You will be using the following techniques to record what the participant is saying or doing during the activity...

☐ Audio Recording ☐ Screen Recording ☒ Written Notes
☐ Apple TestFlight or Google Console (for which the participant's Apple or Google ID will be required)

Instructions to participant: Please read the Participant Information Sheet carefully. If you agree to take part in the activities indicated above, enter your initials in the box below and return this sheet to the student.

- 1) I confirm that I have read and understood the information sheet attached to this form, or it has been read to me. I have had the opportunity to consider the information and ask questions. My questions have been answered to my satisfaction.
- 2) I understand that my participation is voluntary and that I am free to stop taking part at any time without giving a reason. In addition, I understand that I do not have to answer all questions or take part in all listed activities.
- 3) I understand I can ask to see the information I have provided, and I can ask for it to be destroyed before it is aggregated and anonymised. I understand that after the information has been anonymised, I can no longer ask for it to be destroyed.
- 4) I understand that the information I provide will be stored in line with the data protection policy of the University of Liverpool until it is destroyed.
- 5) I understand that my personal contribution will be destroyed when it has been aggregated and anonymised.

- 6) I agree that this consent form will be retained by the project supervisor for a period of two months following submission of the dissertation of which it forms part, after which it will be destroyed.

XXX

I agree to take part in these activities
(enter your initials into the box)

1.

from myself I haste . And you , ' s scythe and eyes . Nor services to be gone , When lofty trees I love well shows now . Yet seem woe . That is to make , another youth and therefore love and men prove ; And your memory death , Save that I loved her mournful hymns did canopy the time-bettering days , Spending again , Me from thee I love doth good turns my life in for the lease Find no truth ; The wrinkles strange shadows on the night , Though not you , to thee , nor despis ' s decease , Although I that putt ' d it be ; And only herald to thee

2.

text="fairest"

```
for i in range (100):
```

```
    token_text=tokenizer.texts_to_sequences([text])[0]
```

```
    pad_token_text=pad_sequences([token_text],maxlen=10,padding='pre')
```

```
    prd=np.argmax(model.predict(pad_token_text))
```

```
for word,pos in tokenizer.word_index.items():
```

```
    #if the value of the word index in the structure is the same as the predicted value
```

```
    if pos==prd:
```

```
        text=text+ " " + word
```

```
        #print(text)
```

```
fairest in their garments sap new know on me more much of more rare
greater desire decays vexed sun this private sun staineth pray
triumphant plea grew still of his heart ' live it not live in him that
your worth greater greater doth heart greater dead heart cruel heart
eye of this heart live it not worth your worth ' thee are worth it see
worth it not greater more heart greater worth still shallowest mind
wear pine issueless sun eye outward treasure issueless lour'st outward
'twixt lour'st of weeds weeds or heart or dead outward end gems faint
accusing
```

3.

Version 1:

```
fairest in their garments sap new know on me more much of more rare
greater desire decays vexed sun this private sun staineth pray
triumphant plea grew still of his heart ' live it not live in him that
your worth greater greater doth heart greater dead heart cruel heart
eye of this heart live it not worth your worth ' thee are worth it see
worth it not greater more heart greater worth still shallowest mind
wear pine issueless sun eye outward treasure issueless lour'st outward
'twixt lour'st of weeds weeds or heart or dead outward end gems faint
accusing
```

Version 2:

```
From hands to sell
```


My sinful loving thee partake

Do in niggarding

Pity me sin there reigns Love

A bliss in thine eyes delight

How would show

Then do offend thine annoy If the fools of altering things of
thy heart think that which I am thine eyes

Feedst thy sake lay But thou consumst thyself thou wilt thou
taste nor outward thus did proceed

O cunning want to be seen

Without this rage shall live Look in worth

That barren of you have what I am a journey in this false eyes
due by feeding

And in some in love

Be where is thy lays upon thy sins more strong but one ever
yet unset With means more delight

Drawn