Towards Evaluating Creativity in Language

Matey Krastev

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

Department of Computing Science

2022

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: 2022

Abstract

We hypothesize that what we call creativity equates to an expression of a given author's unique character through their work, be it in text, speech, art, and so on. Furthermore, we claim that those can be represented as a certain set of linguistic features that can be found in text, e.g. more of this feature makes this text seem more creative.

To that end, we explore the field of linguistic creativity through the lens of statistical text analysis. We investigate a variety of statistical measures that have been suspected to aid in text classification tasks, genre identification and others. Furthermore, we explore prior work in psycholinguistics aiming to measure the effect of words on the reader, and how those effects can be used by authors to the effect of solidifying an author's style. We also hypothesize a set of novel measures that can be linked with the understanding of structure in text.

In this work, we implement a system for working with text data, as well as methods for applying the measures we have explored. Then, we carry out an experimental evaluation of the system on a set of texts using the implemented metrics, on a broad variety of applicable NLP tasks, such as genre classification, authorship identification, and machine-generated text detection.

We conclude that the measures we have explored can be used to evaluate the creativity of text – creativity in terms of a distinguishing feature for an author – and how this lays out some groundwork for further research into the deep topic of computational creativity.

Acknowledgements

Many thanks to my academic supervisor, Dr. Arabella Sinclair, for constantly being involved with the project, guiding me during my first steps in Natural Language Processing, bearing through my ramblings and entertaining my ideas.

Many thanks to everyone who asked me questions about the project, which served to clarify and potentially solidify my ideas, as well as for giving me the opportunity to better explain my ideas to myself through explaining them to others. I will remember that one time I tried to explain this idea in terms of shepherding, sheep and shepherd dogs.

Many thanks go to the many educators who have taught me throughout the years, teaching me to think critically, rationally, and to put everything under scrutiny.

Contents

Abbreviations

GPT	Generative Pre-trained Transformer.
HPC	High Performance Cluster.
LLM	Large Language Model.
MGT	Machine-Generated Text.
MLM	Masked Language Model.
NLP	Natural Language Processing.
POS	Part of Speech.

Chapter 1

Introduction

1.1 Inspiration

At the heart of the field of artificial intelligence is the concept of reproducing aspects defining human intelligence through rigorous examination and replication of the mechanisms that drive progress. This is a uniquely multi-faceted problem with a multitude of approaches, each tailored to a very specific aspect or manifestations of intelligence. Naturally, intelligence assumes following a logical pathway to arrive at sensible conclusions that interact with the real world beneficially. This can be viewed through the lens of methodical, defined process that always follows a certain formula. An organism, assumed to be intelligent, might always follow such formulaic actions given a set of prerequisite conditions to accomplish a defined goal. Certain schools of thought theorise that there is such an order in every little action, and such thread of logicality interweaves every law of nature, known or not. Then, follows the question, can we recognize and define such a thread for the ambitious field of creativity? We seek not to properly define or constrain the subject of creativity, rather, we explore markers of what could only be a subset of the very broad field of human creativity.

The subject of the present document is exclusively the study of linguistic creativity. Henceforth, we seek to: confirm prior results of the research of psycholinguistics, affirm that hypotheses and conclusions drawn from them correlate highly with certain manifestations or aspects of creativity, and make firm the subject of creativity, that is, provide tools that may be used for exploration and analysis of specific creative features found in text.

1.2 Motivation

Large language models (LLMs) are probabilistic models of language widely used for most tasks in the field of Natural Language Processing (NLP), ranging from machine translation, text classification, sentiment analysis, auto-completion, error correction, or even simple dialogue communication. However, because, fundamentally, LLMs operate on probabilities, a lot of the applications utilizing them tend to struggle with generating logically coherent novel sequences.

Large language models are usually trained on enormous language corpora, mined from books and articles (Gerlach and Font-Clos, 2018), social media posts (Derczynski et al., 2016), or otherwise internet crawls (Gao et al., 2021). Therefore, the underlying assumption would be that, in their attempt to replicate language, they could find some success at least in terms of generating basic structure in creative fields of work such as writing code (Chen et al., 2021) or screenplays (Mirowski et al., 2022), among others.

While LLMs display convincingly human-like text in such areas, we instead intend to examine their ability to produce specifically *creative* text. By its very nature, creativity tends to be a subjective matter, thus, we aim to identify specific aspects of creativity that can be more commonly found within creative text, such as poetry or short stories, as opposed to other less creatively-oriented genres, e.g. news articles or user manuals. Having identified specific linguistic markers within human-written creative texts, we can then begin to evaluate the extent to which LLMs exhibit these properties in generated text. This in turn has the potential to inform techniques to adapt existing text-generation models for applications involving creative language. Therefore, we seek to compile and produce a software package that can efficiently and accurately represent and evaluate linguistic creativity.

1.3 Objectives

We set out to investigate specific markers defining or correlating with conventional creativity in language. As some aspects of creativity have been investigated by research disciplines such as the field of psycholinguistics, we seek to filter and compile a set of measures that have been shown to correlate more highly with creative texts in the literature. Following that, we aim to evaluate the extent to which current LLMs can provide insights into capturing creative elements or patterns within writing. Finally, if time allows, we may use these findings to explore how natural language generation can be adapted to produce texts exhibiting more human-like levels of the creative attributes we have discovered. These can be summarised by the following three questions:

- Can psycho-linguistically motivated measures (that is, the explored metrics) successfully characterise creative properties in language?
- Can a machine learning approach be adapted for evaluating creativity in natural language?

For the two questions listed, we develop a suite of benchmarks we shall distribute and report the results of.

1.4 Contributions

We contribute a set of benchmarks for evaluating the creativity of text, as well as a system for benchmarking text across the implemented benchmarks. Furthermore, we introduce two novel metrics based on Masked Language Models (MLMs) for evaluating structure in text samples. We also test the benchmarks on a set of texts, and provide a set of experiments that can be used to evaluate the system.

1.5 Structure of the Report

We outline the structure of the report as well as the contents of each chapter below. Furthermore, the beginning of each chapter contains a short summary for the contents of the chapter.

Chapter 1 Introduction The current chapter. Provides initial insights, motivations, the research question we seek to answer, as well as the contributions we plan to make.

Chapter 2 Related Work Provides background information on the field of computational creativity, as well as the field of natural language processing. We also provide a brief overview of the current state of the art in the field of computational creativity, as well as the current state of the art in the field of natural language processing.

Chapter 3 Methods Provides a detailed description of the methods we use to evaluate the creativity of text. We list some of the datasets we use, as well as accompanying descriptions for those datasets. We also provide a detailed description of the metrics we use to evaluate the creativity of text.

Chapter 4 Design Provides an outline for the design of the benchmark in terms of a software package. We provide a detailed description of the architecture of the software package, as well as a description of the implementation of the metrics.

Chapter 5 Implementation Provides a detailed description of the implementation of the software package. We provide a detailed description of the implementation of the metrics, as well as a description of the implementation of the system, the dependencies and algorithms we use.

Chapter 6 Evaluation Provides a detailed description of the evaluation of the system. We provide a detailed description of the experiments we run, as well as a description and a short discussion of the results of the experiments, along with baseline comparisons from related experiments.

Chapter 7 Discussion and Conclusion A short discussion of the accomplished goals of the project, the limitations that we encountered, as well as the contributions we made. Additionally, we outline potential grounds for future work in the field.

Chapter 2

Related Work

The field of creativity research has been studied for decades, and there are many different approaches to the problem. In this section, we will discuss the different approaches to creativity research, and how they relate to our work. We will also discuss the different approaches to creativity in the context of natural language processing, and how they relate to our work.

2.1 Challenge Landscape

Most of the work going on in our minds as we read a given text is unconscious. We automatically parse characters and tokenize the text into words, sentences, paragraphs, and so on. At the same time, we apply tagging to understand actors (subjects and objects), actions (verbs), setting and situation (adverb), and also try to understand the sense a given word is used in, and then we transform the connotations or the sense of words inside our own little mind representation of the words. We also apply sentiment analysis to understand the emotional state of the text (or speech), and we apply phonetic analysis to understand the pronunciation of the words, as we read them out in our minds, e.g. when reading a book. Note that we have constrained the example of autonomous processes to reading, although these subconscious processes happen regardless of whether we read, write, or speak. Fact of the matter is, language is inherently a complex social construct that takes years to learn, even more to master, and more than a lifetime to perfect. Because it is so expressive and hard to grasp, many rules have been invented and applied to constrain or clearly define the boundaries of the language (e.g. English), so that, within a given language speaker group (e.g. the space of English speakers), the biggest common denominator of people may understand what is being said by the others. The rules may additionally have their own rules and exceptions to the rules.

Therefore, there are multiple approaches being applied to natural language processing tasks. The more classical one, and the one that has been applied for the larger part of the developments in the field, has been the rule-based method. The approach seeks to use clearly defined rules to parse and understand the linguistic features of the given text. It is a strong approach, as language has been strongly studied for centuries and the aforementioned rules and constraints have already been applied to it. For languages with few changing features or slowly changing features, it performs acceptably well on most Natural Language Processing (NLP) tasks, and not far off from human speakers.

Another approach or a subclass of the rule-based approach, is the **statistical method** . The statistical method seeks to find patterns inside language as a whole. For example, the word “wind” may appear both as a noun and a verb – that is, the meaning of the word may be ambiguous – but the noun form is much more prevalent. Therefore, in tasks such as part of speech (PoS) tagging, some algorithms prefer to use the most common type of PoS class a given word occurs as, in a sufficiently large corpus of text. We do come back to the PoS tagging example later on. Alternative example of the statistical method being applied would be translation. Previous approaches to machine translation (MTL) included learning frequency of words and phrases occurring together (and how those map to counterparts in the language being translated to). For example:

> Finding Nemo is like finding fish in a school of fish.

A naive approach to the translation of this sentence would consider school as its most common (noun) definition - that of place of learning for *humans* and translated the word literally. A more sophisticated approach to translation, applying not just simple rules to translating, would consider the whole phrase “school of fish”, and would have understood that it refers to a countable form of the word fish (relating to a large number of fish), and therefore translated the phrase as a whole to the target language.

The most current approach to many of the challenging NLP tasks (text summarization, machine translation, speech recognition, etc.) is a machine-learning based one. The motivation is multifold: firstly, most of the work that goes on in our minds as we read a given text is unconscious and automatic, just as we do not have to consciously intend to breathe in order to breathe. In much the same way, we rarely consciously make an effort to understand every part of the text, and many of the details behind understanding language are vague and ambiguous (consider how someone would respond if they are asked to explain their thought process behind parsing a given text). Secondly, more in line with the topic of our research, we do not have a good (objective) way to measure the quality of the work. It is subjective and difficult to measure. Not to mention, behaviour – and consequently, consciousness – is something that arises from environment, upbringing, culture, and so on. Yet, value for quality is something that multiple individuals can share – many people can have a sense of appreciation for a novel they read or a speech they heard (of course, usually for slightly different reasons and perceptions) – but there tend to be common elements that people widely enjoy seeing and experiencing.

The intuition of machine learning and deep learning is that you can try to replicate the unconscious logical circuitry going on behind the scenes without having a very solid grasp of the exact logic behind it. Therefore, in fields with few or changing rules, such as linguistics, music, and image processing, the machine learning approach tends to find large success, and has even been shown recently to be able to perform very similarly to humans (Bubeck et al., 2023). All in all, we cannot ignore the potential for machine learning to be applied to the field of creativity, and we explore this potential in our work.

We, therefore, go into detail on some of the methods we utilize in the project, and the various approaches that have been taken in the past.

2.1.1 Part of Speech Tagging

As we have established, a word may play a different role depending on its position in the sentence, both absolute and relative to the other words. Words that denote objects or persons, we generally call **nouns**, while words that denote (usually active) actions, we call **verbs**, and so on. Part-of-speech tagging as a task poses the challenge of assigning to the sequence of tokens (we use "tokens" and "words" interchangeably) $x_1, x_2, \dots, x_n$, a sequence of tags $y_1, y_2, \dots, y_n$, where each token x_i has a corresponding tag y_i, making tagging a task of disambiguation, that is, removing ambiguities in text.

The phenomenon of ambiguity is hardly exclusive to the English language, although it is a key problem that we would need to take head-on, as many of the tasks we do want to tackle, rely on POS tagging, in order to avoid needless computations (and in consideration of the limited computing resources for both end-users and ourselves, as we may do larger-scale experiments). Human performance on POS-tagging has been shown to be around 97% accuracy (Manning, 2011) for English texts. We will use this as a base of comparison with automatic algorithms.

Types:	**WSJ**	**Brown**
Unambiguous (1 tag)	44,432 (86%)	45,799 (85%)
Ambiguous (2+ tags)	7,025 (14%)	8,050 (15%)
Tokens:		
Unambiguous (1 tag)	577,421 (45%)	384,349 (33%)
Ambiguous (2+ tags)	711,780 (55%)	786,646 (67%)

Table 2.1: Tag ambiguity in the Brown and WSJ corpora (Treebank-3 45-tag tagset). Adapted from Jurafsky and Martin (2009)

.

As seen in Table 2.1, many of the words in the tag-annotated corpora are, in fact, unambiguous. That is, the given word appears only as one single *type* of part of speech. What we may miss, however, is the fact that the actual *tokens*, the words, are mostly ambiguous (55% on WSJ and 67% on Brown). For example, *elephant* can only appear as a noun, but the word *back* can appear as either a verb, an adjective, a noun, or an adverb. And *back* is a much more common word than, say, *elephant*. Given this example, we can then proceed to potential solutions.

Generally, the issue is hardly new in the field of text mining, and solutions have been attempted since long ago in the past. The Brown corpus by Francis and Kucera (1979) is a manually annotated for parts of speech corpus of American English texts from variety of genres, and has been widely used as a benchmark for algorithms for POS-tagging – at least ones focusing on the English language.

One baseline metric is the **most frequent class** classifier. This method assigns to each word the most frequent tag it appears as in some training corpus. The metric has been shown to be 92% effective (Jurafsky and Martin, 2009) in correctly classifying the tags of a given text, which is just 5% below human accuracy, as described above. However, we can do better.

Hidden Markov Models are among the more successful ones, as demonstrated by Schütze and Singer (1994); Thede and Harper (1999); Nigam et al. (1999). The idea stems from the fact that the probability of a word x_i appearing in a given position in a sentence, is dependent on the word that came before it. This is a Markov assumption, as it assumes that the probability of a word appearing in a given position is independent of the words that come after it. The given assumption is a strong one, and it is not always true. However, it is a good starting point, and it has been shown to be effective in many cases. The

While the HMM is a useful and powerful model, it turns out that HMMs need a number of augmentations to achieve high accuracy, as exemplified by authors such as Goldberg et al. (2008). Brants (2000) demonstrate between 96 and 99% accuracy for HMMs, while Thede and Harper (1999) demonstrate an accuracy between 96-98%, and between 88 and 93% accuracy on various datasets by Goldberg et al. (2008). For example, in POS tagging as in other tasks, we often run into unknown words: proper names and acronyms are created very often, and even new common nouns and verbs enter the language at a surprising rate – the Oxford English Dictionary has added 700 new words to its definitions just in its most recent quarter-yearly update in March (Oxford English Dictionary Editorial, 2023).

Conditional Random Fields (CRFs) have been proposed to deal with the following motivations: the adding of arbitrary features, e.g. based on capitalization or morphology (words starting with capital letters are likely to be proper nouns, words ending with -ed tend to be past tense (VBD or VBN), etc.); knowing the previous or following words (if the previous word is "the", the current one is not likely to be a verb).

State-of-the-art POS taggers use neural networks behind the hood, being either bidirectional RNNs or Transformers like BERT.

2.1.2 Word Sense Disambiguation

2.1.3 Sentiment Analysis

2.1.4 Named Entity Recognition

2.1.5 Phonetic Analysis

2.1.6 Natural Language Generation

- top KP sampling
- challenges

2.2 Creative Measures

The field of creativity has been broadly studied, initially by psychologists, and more recently by computer scientists. Intuitively, the nature of creativity is a subjective matter, and therefore, it is difficult to define. However, there are some commonalities that can be found in creative works. For example, creativity is often associated with novelty, and is often associated with the ability to generate new ideas. Franceschelli and Musolesi (2022), for example, determine the three factors of creativity as value, novelty, and surprise, and then explore machine learning approaches to measuring creativity. We agree with them, but decide not to limit ourselves to just these three. However, their contributions provide insights we can use in our work.

- Burstiness of verbs and derived nouns: Patterns of language are sometimes 'bursty' Pierrehumbert (2012). This paper presents an analysis of text patterns for domain X. measures include XYZ...
-

2.3 Tools

Chapter 3

Methodology

In this chapter, we explore the different datasets to be used, the methods for evaluating creativity, and the algorithms for creativity evaluation. We will furthermore discuss the strengths and limitations of the proposed methods and algorithms, including time complexity, memory constraints, and the accuracy of the results. We take an informed approach to the selection of the datasets and the methods for evaluating creativity, and discuss the reasons for our choices in detail, and support them with relevant literature as explored by other researchers in the field.

3.1 Datasets

Datasets are vital for the success of any given project in the field of machine learning, and even more so when concerning linguistics. As evidenced by Torralba and Efros (2011), the quality of the data used for training a model has a direct impact on the quality of the results. A model trained on a specific dataset, e.g. a corpus of law documents, can be expected to perform poorly on a dataset of medical documents, as the two domains are inherently different. Thus, we take particular care in planning and selecting the datasets we use. We also consider ease of use and access, as some datasets may require additional processing, others are subject to availability issues (e.g. paid datasets and corpora), and some may be too large to be used in a reasonable amount of time. In this section, we will explore the datasets used in this project, and discuss their strengths and limitations.

3.1.1 Brown Corpus

The Brown Corpus (Francis and Kucera, 1979) is a widely used corpus in the field of computational linguistics, noted for the small variety of genres of literature it contains. The Corpus itself is founded on a compilation of American English literature from the year 1961. It is also small in terms of size, totalling around one million words, at least compared to modern corpora, which we also explore later on. The corpus also suffers from the issue of recency, as the works and language may be outdated for modern speakers of English.

Of interest is the fact that the corpus has been manually tagged for parts of speech, a process that tends to be error-prone. As we will see later on, this fact has implications in terms of the supervised learning algorithms we implement for creativity evaluation. Still, we opt to utilize it primarily for prototyping purposes and drawing preliminary conclusions about the effectiveness of the implemented algorithms, rather than in-depth analysis and publication of results.

3.1.2 Project Gutenberg

Project Gutenberg[1] is a large collection of more than 50,000 works available in the public domain. The collection contains literature from various years and various genres and thus is suitable for training and evaluation of the developed benchmarks in the context of creativity study.

As the Project does not offer an easy to process copy of its collection, we turn to the work of Gerlach and Font-Clos (2018). The team developed a catalogue for on-demand download of the entire set of books available on the Project Gutenberg website, intended for use in the study of computational linguistics. The tool avoids the overhead of writing a web-scraper or a manual parser for the downloadable collections of Project Gutenberg books made available by third parties, as well as enables easy synchronization of newly released literature. Instead, we are only required to develop a simple pipeline for the data to be fed into the utilized systems.

3.1.3 Hierarchical Neural Story Generation

In their work, Fan et al. (2018) trained a language model for text generation tasks on a dataset comprised of short stories submitted by multiple users given a particular premise (a prompt or a theme) by another user. The dataset in question is technically referred to a series of posts and comments (threads) to them on the popular social media platform REDDIT, and more tightly, the *subreddit* forum R/WRITINGPROMPTS. The authors of the work Fan et al. (2018) have made the dataset available for public use, and we have used it for the purpose of evaluating the performance of our creativity benchmarks. As described by the authors on their GitHub page[1], the paper models the first 1000 tokens (words) of each story.

3.1.4 Discarded Datasets and Corpora

Some datasets were considered, however, discarded due to: not being deemed applicable for the context of the application; general lack of availability of the dataset in a form that is easily accessible for our purposes; simply being infeasible to use due to the size of the dataset and the hardware constraints imposed on the project; or other reasons of similar nature.

The COCA

The Corpus of Contemporary American English (COCA)[2] is a large corpus of American English, containing nearly 1 billion words of text from contemporary sources. It is a collection of texts from a variety of genres, including fiction, non-fiction, and academic writing. The corpus offers a variety of tools for analysis of the data, including a concordance tool, a word frequency list, and a collocation finder. Naturally, many of those tools could be used in the field of statistical creativity analysis that we explore.

The corpus does offer limited access to the full API, as well as free samples of the data, however, the full corpus is not available for free, and the cost of acquiring it is prohibitive for the limitations set forward by the project. Nevertheless, the corpus is a valuable resource for the field of computational linguistics, and we would like to explore it further given less constraints.

[1] `https://www.gutenberg.org/`

[1] `https://github.com/facebookresearch/fairseq/blob/main/examples/stories/README.md`

[2] `https://www.english-corpora.org/coca/`

3.2 WordNet

WordNet(Fellbaum, 1998) is a lexical database of semantic relations between words that links words into semantic relations including synonyms, hyponyms, and meronyms. The synonyms are grouped into synsets (sets of synonyms) with short definitions and usage examples. It can thus be seen as a combination and extension of a dictionary and thesaurus (Wikipedia contributors, 2023).

For our specific use cases, we have identified it as a valuable resource in terms of relational representation of words in semantic space. In the given context, this enables us to traverse a semantic graph for synonyms and related words for the goal of enriching potential similarity between the set of creative parts of speech (i.e., nouns, adjectives, adverbs), which we narrow down our scope to in particular.

3.2.1 Numerical representations of semantic tokens

The idea of representing words or lexical tokens as numerical vectors (or even scalars) is hardly new. For example the SimLex-999 dataset (Hill et al., 2015) gives values on a scale from 0 to 10, like the examples below, which range from near-synonyms (vanish, disappear) to pairs that scarcely seem to have anything in common (hole, agreement):

word1	word2	score
vanish	disappear	9.8
hole	agreement	1.2

Table 3.1: Example Simlex-999 pairs

Early work on affective meaning by Osgood et al. (1957) found that words varied along three important dimensions of affective meaning:

- valence: the pleasantness of the stimulus
- arousal: the intensity of emotion provoked by the stimulus
- dominance: the degree of control exerted by the stimulus

Osgood et al. (1957) noticed that in using these 3 numbers to represent the meaning of a word, the model was representing each word as a point in a three-dimensional space, a vector whose three dimensions corresponded to the word's rating on the three scales. This revolutionary idea that word meaning could be represented as a point in space (e.g., that part of the meaning of heartbreak can be represented as the point $[2.45, 5.65, 3.58]$) was the first expression of the vector semantics models that we introduce next.

Word2Vec

Mikolov et al. (2013) show in their work that words may be represented as dense vectors in N-dimensional space, and we can perform mathematical operations on them that may yield effective results in terms of word representation.

Measuring distance in vector representations of semantic tokens

Intuition tells us that the dot product of vectors in N-dimensional space will grow when the set of vectors has similar values and decrease when the values are not similar. Thus, we can then construct the following metric for semantic similarity between vector representations of words:

$$D(v,w) = v \times w = \sum_{i=1}^{N} v_i w_i = v_1 w_1 + v_2 w_2 + \cdots + v_N w_N$$

The current metric, however, suffers from the problem that vectors of higher dimensions will inevitably be larger than vectors with lower dimensions. Furthermore, embedding vectors for words that occur frequently in text, tend to have high values in more dimensions, that is, they correlate with more words. The proposed solution is to normalize using the **vector length** as defined:

$$|v| = \sqrt{\sum_{i=1}^{N} v_i^2}$$

Therefore, we obtain the following:

$$\text{Similarity}(v,w) = \frac{v \times w}{|v||w|} = \frac{\sum_{i=1}^{N} v_i w_i}{\sqrt{\sum_{i=1}^{N} v_i^2}\sqrt{\sum_{i=1}^{N} w_i^2}}$$

This product turns out to be the same as the cosine of the angle between two vectors:

$$\frac{a \times b}{|a||b|} = \cos(\theta)$$

Therefore, we will call this metric the **cosine similarity** of two words. As mentioned, the similarity grows for vectors with similar features along the same dimensions. Note the boundaries of said cosine metric: we get -1 for vectors which are polar opposites, 0 for orthogonal vectors, and 1 for equivalent vectors. Of note is the fact that such learned vector embeddings only have values in the positive ranges, thus, it is impossible to have negative values for the cosine similarity (Similarity$(a,b) \in [0,1]$).

Contrary to it, we also identify the metric of **cosine distance** between two vectors, as one minus the similarity of the vectors, or:

$$\text{Distance}(v,w) = 1 - \text{Similarity}(v,w)$$

The cosine distance may prove useful when dealing with minimisation problems as is often the case with machine learning.

3.3 Metrics

3.3.1 Number of Words

The total number of words in a given piece of text. At first glance, this metric does not impress and is, in fact, exceedingly simple. But that is fine – we do not always need complex metrics. Sometimes, even a trivial metric as this one can inform a lot about the structure of the text. For example, the number of words in a text is directly correlated with the length of the text. This can be useful in determining the complexity of the text, as well as the time it takes to read it. In some uses, for example, comparing between books and *Twitter* posts, we do not need much more information to recognize that these texts belong to entirely different genres. Such a metric is a good complement to and often used in conjunction with other metrics.

3.3.2 Number of Sentences

The number of sentences, similarly to number of words, is a trivial measure for the length of the text. However, it can be used to determine the complexity of the text. For example, a text with a large number of sentences is likely to be more complex than a text with a small number of sentences. This is because a text with a large number of sentences is likely to involve longer intellectual activity. Of course, in light of recent developments in the field of natural language generation, this metric is not particularly useful. However, due to how trivial to implement it is, it can be used in conjunction with other metrics for text classification tasks.

3.3.3 Word Length

> *"Because even the smallest of words can be the ones to hurt you, or save you." –* Natsuki Takaya

Word length fills in the set of trivial metrics we implement for text benchmarking. The intuition is simple. Given a sufficiently large corpus, the average word length – that is, the number of characters in a word – will converge to a certain number – in English, this number tends to be between 4 and 5. Any deviations, either positive or negative, from this norm can then be used to determine the complexity of the text. For example, a text with a large number of long words is likely to be more complex than a text with a large number of short words. Naturally, words expressing more specific concepts tend to have a longer character length than words we use in general speech and are sometimes ambiguous. This phenomenon is established in English, although the essence may not generalize well for other languages, e.g. Chinese and Japanese, where a single character can generalize to a whole word or a concept as a whole, but given that we are working in the context of the English language, we are not concerned with this issue.

3.3.4 Sentence Length

Similar to word length above, the average sentence length is a trivial metric describing the number of characters per sentence. Intuition tells us it will be closely related to the average word length, but also indicative of text features such as complexity and readability. For example, legal documents tend to have longer sentences than, say, newspaper articles. This is because legal documents tend to be more complex and require more time to read and understand. In contrast, newspaper articles tend to be more accessible and are written in a way that is easy to understand.

Writers may also be interested in this metric, as very long sentences are often difficult to read and understand, as the reader may lose track of the subject of the sentence among the many objects, actions and modifiers; not to mention unnecessary punctuation where simply beginning a new sentence would be far more readable... a useful feature like this can pinpoint such writing issues, inform writers where they may cut or simplify their sentences, and in general help them improve their writing style – a feature that is often overlooked in the context of text understanding – this is also the longest sentence in the entire document.

3.3.5 Number of Tokens

Completing the set of trivial metrics is the general number of tokens in the text. The metric correlates highly with average sentence length and word length. Rather than counting characters in the sentence or word length, however, we take a look at the number of tokens encountered in the text, usually at the sentence level.

3.3.6 Concreteness

Concreteness is the degree to which a word refers to a tangible object or a concrete idea. For example, the word *apple* is concrete, while the word *time* is abstract. Brysbaert et al. (2014) provide a dataset of concreteness ratings for 40,000 English lemmas (English words and 2,896 two-word expressions (such as "zebra crossing" and "zoom in"), obtained from over four thousand participants by means of a norming study using internet crowdsourcing for data collection). The dataset is based on the concreteness ratings of the four thousand participants, who rated the concreteness of 40,000 words on a scale from 1 to 5. The concreteness of a word is measured on a scale from 1 to 5, where 1 is the most abstract and 5 is the most concrete:

> Some words refer to things or actions in reality, which you can experience directly through one of the five senses. We call these words concrete words. Other words refer to meanings that cannot be experienced directly but which we know because the meanings can be defined by other words. These are abstract words. Still other words fall in-between the two extremes, because we can experience them to some extent and in addition we rely on language to understand them. We want you to indicate how concrete the meaning of each word is for you by using a 5-point rating scale going from abstract to concrete.

The dataset provides norms for the 40,000 words and 2,896 two-word expressions – including mean and standard deviation for each entry.

The intuition of this metric is that a word that is more concrete is more likely to be used in a creative context, as it is easier to imagine and relate to. It not only describes one aspect of the word's meaning, but authors (and genres, in general), tend to exhibit specific characteristics, such as legal documents being more generally more concrete - one would expect concrete objects and entities to appear more in documents such as the UN Human Rights Charter, or protocols for health standards control, for example.

3.3.7 Imageability

Imageability is the degree to which a word evokes a mental image, as described by de Groot (1989). For example, the word *apple* is more imageable than the word *time*. Brysbaert et al. (2014) provide a dataset of imageability ratings for 9,000 English lemmas (English words and 2,896 two-word expressions (such as "zebra crossing" and "zoom in"), obtained from over four thousand participants by means of a norming study using internet crowdsourcing for data collection). The dataset is based on the imageability ratings of the four thousand participants, who rated the imageability of 9,000 words on a scale from 1 to 5. The dataset also contains the number of participants who rated each word, and the standard deviation of the ratings.

3.3.8 Frequent Word Usage

"Separate text from context and all that remains is a con."— Stewart Stafford

Word frequency refers to the number of times a given word appears in a given context. Word frequency naturally differs from text to text, and smart word choice in general is an excellent indicator for intellectual linguistic use. The intuition behind selecting this metric is that words that are occurring less frequently in common speech are more likely to be used in a creative context.

To give an example by rewording the last sentence, would yield: “The intuition behind identifying this linguistic measure owes to the words’ property of inverse proportionality between frequency and perceived creative or intellectual value.”

As noted, less common words are associated with higher perceived intellectual value. Even more so, the use of less common collocations (words occurring very close in a given context) hints at a higher level of linguistic skill. Of course, simply chaining completely unrelated words together (e.g. “palmarian tobaccophile ephemeron urbarial”) hints not to high intellectual value, but rather to spitting out a random sequence of words. Properly applied in context, though, commonly not associated words can be used to great effect. This is especially true in the case of poetry, where the use of uncommon words and collocations is a common practice, or, for example, in biological contexts, such as medicine and botany, where very precise yet niche namings and conventions are mandated. This type of dissonance between common speech and niche terminology is a common theme in creative writing, and is often used to great effect. For example:

> *“When they’d gone the old man turned around to watch the sun’s slow descent. The Boat of Millions of Years, he thought; the boat of the dying sungod Ra, tacking down the western sky to the source of the dark river that runs through the underworld from west to east, through the twelve hours of the night, at the far eastern end of which the boat will tomorrow reappear, bearing a once again youthful, newly reignited sun.”*
>
> – *The Anubis Gates*, Tim Powers

In this context, “boat” is a completely valid and understandable synonym of the word “sun”, yet the word “boat” co-occurring with the word “sun” outside this context is not common, and therefore, we are prompted to believe that this context is more ‘creative’.

We tackle the topic of contextual surprise further on with subsequent metrics, but for now, we focus on the general idea of individual word frequency.

Given a sufficiently large linguistic corpus, we obtain a list of words and their frequency of occurrence. We can then use this list to calculate the frequency of occurrence of a given word in a given text. We can then use this frequency as a metric for the text’s creativity. Choice of corpus is key here, as the corpus should be large enough to contain a wide variety of words, but not specialized enough to inflate the frequency of niche words. For example, a corpus of medical texts would contain a lot of medical terminology, which would inflate the frequency of medical terms, and therefore, would not be a good choice for a general creativity metric, for example in the case of a poetry contest.

For our use case, we opt to use the British National Corpus (BNC) (BNC Consortium, 2007), which is a 100 million word collection of samples of written and spoken language from a wide range of sources, designed to represent a wide cross-section of British English from the later part of the 20th century, both spoken and written. The BNC is a good choice for our use case, as it is a general corpus, and contains a wide variety of words, but is not specialized enough to inflate the frequency of niche words.

The frequency lists we use are provided by the work of Leech et al. (2014) and are readily available in sheet form for both lemmatized and non-lemmatized words. In our case, we attempt to adhere only to the lemmatized versions in order to have consistency with previous metrics, but also to have normalized results, e.g., although the words ‘am’, ‘is’, ‘are’ are all inflections of the verb ‘to be’, they may have different frequencies and different positions in the list. POS tagging and lemmatization again come into play here, as we need to be able to identify the lemma of a given word in order to find its proper frequency in the list. The frequency lists indicate the words’ frequencies per 100 million tokens. Intuitively, given a varied enough corpus such as the BNC, we expect these numbers to normalize and generalize well for general English. We then use the frequencies for the lemmas and the take the logarithm with base 10 of the given frequency like so:

$$\text{freq}(x) = \log_{10}(\text{Frequency}_{BNC\ 1M}(\text{Lemma}(x))) \tag{3.1}$$

Like before, if a word does not appear in the BNC, we discard it and continue. We then calculate the average frequency of the words in the text, and return the metric for interpretation by the end user.

3.3.9 Proportion of Parts of Speech

Chapter 4

Design

In the following chapter, we introduce concepts behind the design of the developed library and how those will be implemented in the application. We also discuss the technology choices we have made and the reasoning behind them. Furthermore, we take a look at how the application will be structured and how it will be distributed, and how the users will be able to interact with it via a command-line interface, or apply declared methods and classes inside their own applications. Finally, we discuss the delivery of a documentation and a user guide, as well as the testing and validation of the application.

4.1 Requirements

In this section, we detail the requirements regarding the application and the library in the following sections. We will also discuss the requirements regarding the documentation and the user guide, as well as the requirements regarding the testing and validation of the application.

4.1.1 Scenarios

We use scenarios to better illustrate the use-cases of the MAD HATTER package. They provide a frame for the functional and non-functional requirements and allow us to nail down the specific requirements and pain points, well in advance of the de facto launch of the package.

1. Evaluating linguistic features associated with creative aspects of language in a given text. Users should be able to clearly present a text and evaluate it against a set of metrics, which will then be presented in a clear and concise manner.

2. Providing methods for interacting, plotting and visualizing the results of the evaluation. Users should be able to easily interact with the results of the evaluation, and plot them in a way that is easy to understand and interpret.

3. Providing a pipeline for batch data analysis to be used in larger-scale linguistic research operations. The users should be able to easily process large amounts of data in a batch manner, and then interact with the results of the evaluation in a way that is easy to understand and interpret.

4.1.2 Functional Requirements

Functional requirements pinpoint the specific tasks that the application needs to be able to perform. They are the most important part of the requirements, as they are the ones that will be used to evaluate the success of the application. We list those below.

(FR1) The user must be able to evaluate a given text against a set of metrics.

When fed a text, the system must return evaluation of the text against a set of metrics. The user must be able to specify which metrics they want to use for the evaluation, and the system must be able to return the results of the evaluation in a clear and concise manner.

(FR2) The user must be able to interact with the results of the evaluation.

The user must be able to interact with the results of the evaluation in a way that is easy to understand and interpret. The user must be able to plot the results of the evaluation, as well as export them in a format that is easy to use in other applications.

(FR3) The user must be able to batch-process a large volume of texts.

The user must be able to batch-process a large volume of texts. The return format of the data must be easy to integrate with other applications, especially in the context of linguistic data analysis.

4.1.3 Non-functional Requirements

The package henceforth needs to satisfy a list of viable non-functional requirements, which we will list below.

(NFR1) The package must be able to be used by users with minimal technical knowledge.

The package should provide methods that make text analysis viable for users with minimal technical knowledge. Beyond installation, it should be trivial to evaluate and interpret the results of the evaluation. The package must also be able to be used with all sorts of texts — from single sentences to full-length books.

(NFR2) The package must work within limited computing capabilities.

As above, users should be able to run and evaluate their texts on their personal computers, without the need for specialized hardware. The package should be able to run on a single machine, and should not require any specialized hardware.

(NFR3) The package must be able to scale to large amounts of data.

The package should be able to scale to large amounts of data, and should be able to process large amounts of data in a reasonable timeframe.

(NFR4) The package must be easily configurable.

The package should be easily configurable, and should allow users to easily change the parameters of the evaluation. The package should also allow users to selectively include metrics to the evaluation.

(NFR5) The package must be easily extensible.

The package should be easily extensible, and should allow advanced users to easily add new metrics to the evaluation. The package should also allow users to easily add new methods for interacting with the results of the evaluation.

(NFR6) The package must be well-documented.

The package should be well-documented, and should provide a clear and concise user guide, as well as a clear and concise documentation for the package itself. The documentation should be easily accessible and should be easy to understand.

(NFR7) The package must be issue-proof.

The package should be well-tested against a set of test cases, and must contain as few bugs and unhandled behaviours as possible. Exceptions and errors, wherever they may occur, must have a clear and concise message, guiding users to the source of the issue.

4.1.4 Concerns

Working with text and developing a package for text analysis results in several concerns we need to address.

Speed. Working with text can be *slow*. This is especially true when working with large amounts of text, or when working with large language models. This means that we must take into account the algorithms and data structures we apply, as a single unoptimized algorithm can result in a very slow application. We must also take into account the size of the language models we use, as larger models tend to be slower to process.

Memory Consumption. The old adage that *memory is cheap* is not entirely true. While it is true that memory is cheap, it is also true that memory is not free (*and no, we cannot "just download more RAM"*). Furthermore, model accuracy tends to grow with the size of the neural network and the size of the used vocabulary. Naturally, we then need to seek a compromise on the size of the models we use, as we cannot:

1. Feasibly make use of the larger model variants during the research stage of the project, where we aim to process large corpora, evaluate the performance of the algorithms on them and make conclusions about the data. If we do aim to speed up this process, we can benefit from parallel computing — but processing large batches of text in parallel has a non-negligible likelihood of running out of allocated memory even on some High Performance Cluster (HPC) clusters.

2. Expect users to run too large models on their personal computers, as this would result in a very poor user experience and a far reduced space of potential users. We do not plan to hardcode any models (large or small) in the application, however, the provided guides will reference selected smaller-scale pretrained LLMs, which come with the advantage of being more sustainable long-term. Naturally, we will also provide a way for more experienced and more capable organizations or individuals to run larger models with minimal effort.

Ease of Use. The target users are not expected to be very technically proficient. This means that we need to provide a way for users to easily interact with the application, and easily interpret the results of the evaluation. We also need to provide a way for users to easily install and use the application, without the need for specialized hardware or software. The application should be potentially viable for instant feedback in the context of writing assistance, and potentially integrable with existing writing tools.

4.2 Technology Choices

4.2.1 Python

We will be using the Python[1] programming language for the development, prototyping, and release of the system. Python is a mature dynamically-typed interpreted programming language with a

[1]`https://www.python.org/`

rich ecosystem of libraries and frameworks, especially popular with academic staff and data scientists. Python fundamentally lags behind the competition in terms of raw speed and performance, but makes up for it with its ease of use and rich ecosystem. Wherever performance is desired, library developers instead implement core code in a more-performant language, and instead provide bindings for Python, thus making Python a viable choice even in terms of computing-heavy tasks.

Specifically, we use Python version 3.10.X as shipped by the Anaconda software package. We are aware that the most recent stable version of Python 3.11 brings non-negligible optimizations and faster execution speed for some Python scripts, however, in light of the fact that the Anaconda distribution is still shipping Python 3.10.X, and the fact that some packages have not been well-tested with Python 3.11, we opt to use that version for the time being. We will be using the Anaconda distribution as it is a very popular and mature distribution of Python, which is also very easy to install and use. It also comes with a large number of pre-installed packages, which will be very useful for the current developer experience.

4.2.2 NumPy

NumPy[2] is a Python library for scientific computing. NumPy provides a variety of highly optimized data structures, as well as a large number of mathematical functions that can be applied to said data structures. It is a very popular library, and is widely used in the Python ecosystem. NumPy boasts a very mature library, with a large community of developers and researchers, leading to a very well-tested and well-documented product. Furthermore, because the core of NumPy is implemented in low-level programming languages, such as the C programming language, the performance of functions using methods in NumPy barely lags behind for math-heavy operations against even the fastest languages available.

NumPy addresses our fundamental need for a performant library for scientific computing. Implementing specific array data structures in NumPy enables us to severely cut down on performance time for math-heavy operations, as well as cut down on memory consumption. NumPy also interfaces easily with PyTorch, another library for high-performance computing, which we will be using for the implementation of the models used in the application.

4.2.3 PyTorch

PyTorch is Python library for imperative-style high-performance computing, optimized for parallel operations on the GPU, and particularly applied for deep learning (Paszke et al., 2019). PyTorch is an open-source library with rich documentation, and many deep learning models have been implemented with it. The functions and models available within PyTorch are essential for the implementation of the heavyweight metrics we choose to implement.

Additionally, many of the freely available Large Language Model (LLM) architectures have already been implemented and are publicly available via high-level APIs interacting with the models. Thus, we can avoid implementing or reimplementing the LLM architectures on our own, and instead focus on the applications of the implemented package. Furthermore, any LLMs used in the application will be implemented in PyTorch.

[2]`https://numpy.org/`

Comparison with TensorFlow

TensorFlow is a Python library for the same purpose as PyTorch – to provide low- and high-level abstractions for efficient parallel computing, optimized for GPU operations, and implementations of deep learning scientific models. Anything one can achieve with PyTorch, they can achieve with TensorFlow, as well. Similarly, most of the common model architectures are also available as high level abstractions implemented in TensorFlow. In general, there are very few key differences between the two.

Ultimately, however, we have opted to use PyTorch over TensorFlow for the following reasons:

1. PyTorch is more popular in the academic community, and is more widely used in research. This means that there is a larger community of developers and researchers working with PyTorch, and a larger number of models implemented in PyTorch.

2. Integration with NumPy. Because key computations in the project will be handled by NumPy, it is important that the library we use for high-performance computing integrates well with NumPy. PyTorch integrates very well with NumPy, and is able to convert NumPy arrays to PyTorch tensors with minimal effort.

3. PyTorch is more "Pythonic" than TensorFlow. This means that PyTorch is more intuitive to use, and the general coding style is more natural – leading to an excellent developer experience.

4.2.4 NLP Frameworks

In this section, we evaluate a variety of available Python libraries for Natural Language Processing (NLP) tasks, and discuss the reasoning behind our choice of the library we will be using in the application.

NLTK

NLTK is a key Python library for natural language processing, primarily built for education purposes and managed as an open-source software, built to be relatively modular and lightweight. Commonly used by researchers and students for understanding and implementing algorithms for NLP tasks, it is a relatively popular and mature framework with a healthy extension ecosystem, where contributors are able to write their own modules and share them with the community.

The strengths of NLTK include:

- A large number of modules for a variety of NLP tasks, including tokenization, stemming, tagging, parsing, and so on.

- A large number of corpora and datasets for a variety of NLP tasks, including the Brown Corpus, the Penn Treebank, and so on.

TextBlob

TextBlob is an NLP library drawing heavy inspiration from algorithms available in NLTK, providing a higher-level abstraction for common NLP tasks. It is built to be easy to use, and provides a very intuitive API for common NLP tasks. It provides an easy interface for parsing and working with text on all levels, but it is somewhat lacking in terms of more advanced NLP tasks. Furthermore, development has stagnated and is rarely updated with new features or bug fixes.

SpaCy

SpaCy is an open-source Python library for advanced natural language processing, designed to be easily applied in production environments and implementing pipelines for enhanced NLP tasks. Whereas NLTK is primarily used for research and education, SpaCy is commonly being applied in industry environments. SpaCy is in active development and enjoys a large community of developers and researchers.

The strengths of SpaCy include:

- pretrained pipelines currently supporting tokenization and training for 70+ languages
- state-of-the-art speed and neural network models for tagging, parsing, named entity recognition, text classification and more
- multi-task learning with pretrained transformers like BERT, as well as a production-ready training system and easy model packaging, deployment and workflow management.

Comparison between NLTK and SpaCy

We draw a few differences between NLTK and SpaCy in the context of our application. We ignore TextBlob, as it has most of the capabilities of NLTK but seems to have stagnated in development. The two frameworks were compared across the following criteria:

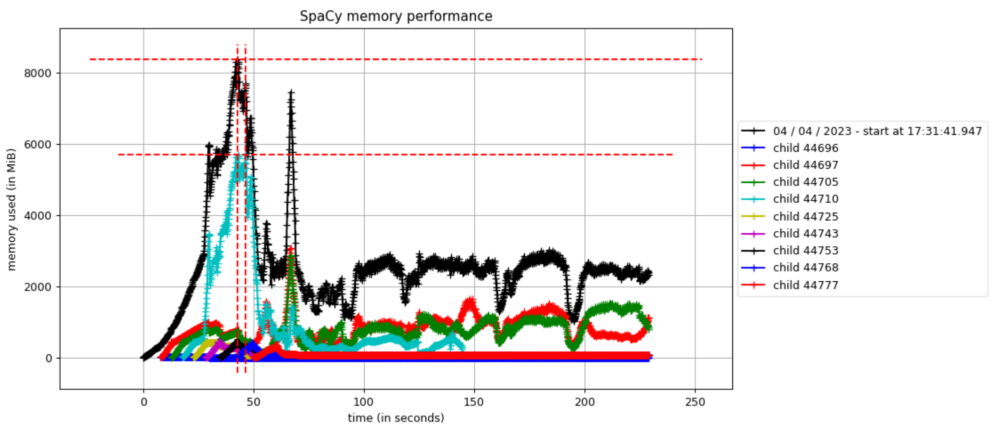
1. **Ease of Use.** Both frameworks are relatively easy to use, but SpaCy is built as an NLP Swiss army-knife not just for production, but all tasks. NLTK, on the other hand, requires some reading and understanding of the underlying algorithms to be used effectively.
2. **Performance.** Both frameworks perform relatively similarly in performance for key algorithms such as tokenization and tagging. A lot of the SpaCy key code has been implemented in Cython, a static compiler for Python providing magnitudes of speed improvements over regular Python, in theory making it faster for certain applications. We do not account for the performance of text processing pipeline modules using neural networks, as we will not be using those in the application. In the limited testing we did, we did not find a significant advantage for the simple use case we had for an NLP framework. Therefore, this criterion is a tie.
3. **Memory Consumption.** SpaCy inherently requires more memory as it does much more than tokenization internally and stores more information about the text. Modules utilizing NLTK algorithms, on the other hand, are more lightweight and can be selected to use less memory. Table 4.1 and Figure 4.1 list a small-scale experiment we carried out to compare the memory consumption of the two frameworks.

It is unlikely that we will use all capabilities that both of SpaCy and TextBlob provide straight out from the box. We can, therefore, avoid the overhead of the other two frameworks, and instead use NLTK and choose which modules we use selectively. Thus, our NLP framework of choice is NLTK.

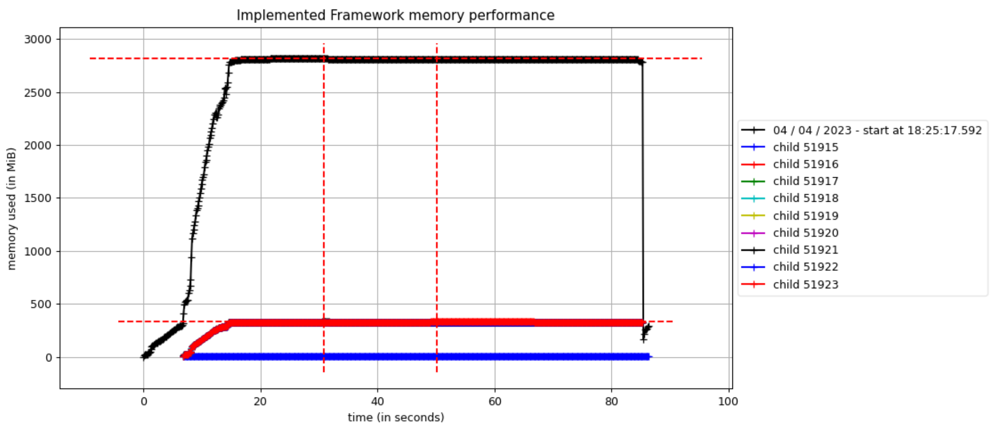
Framework	Peak Memory	Increment
SpaCy	5089.13 MiB	4465.29 MiB
NLTK	434.81 MiB	48.75 MiB

Table 4.1: We carried out a small experiment to compare the memory consumption of SpaCy and NLTK. We prepared a subset of 1000 book files and only opted to tokenize and tag the words of each, and store both as simple Python lists. We ran the experiment in parallel with multiple ($n = 16$) processes, to simulate a real use case we would have for the frameworks. A Python process running SpaCy, unfortunately, even when stripped down to a minimal pipeline, consumes massive amounts of memory. We can see that the memory consumption of SpaCy is almost 10 times that of NLTK.

Figure 4.1: Comparison between SpaCy and NLTK in a batch processing scenario. The simple task was to tokenize the text into words, sentences, and to POS tag the words. We used 1800 texts of length up to 100000 characters each. For SpaCy, we used the `pipeline` method, and for NLTK we set up a simple pipeline using Python's built-in `multiprocessing` module.



(a) Performance of SpaCy. Black line represents cumulative memory usage for the parent process. Seems to vary quite a bit and has a tendency to eat away memory.



(b) Performance of NLTK default subset of features with `multiprocessing`. Black line represents cumulative memory usage for the parent process. Relatively stable throughout and predictable. Each process only holds about as much memory as a standalone Python process.

4.3 Code Style

Python by nature is a very stylistically-opinionated language. Things like indentation, whitespace, and line length are all very important in Python, and are all enforced by the interpreter. This is a very good thing, as it means that Python code is very easy to read and understand, and it is very easy to maintain a consistent code style across the codebase.

4.3.1 PEP8

We will be using PEP8[3] (van Rossum et al., 2001) as our code style guide. PEP8 is a style guide for Python code, which is maintained by the Python Software Foundation. It is a very popular, and comprehensive style guide, widely used by many Python developers and organizations. It covers a wide range of topics, including naming conventions, indentation, line length, whitespace, comments, "docstrings" (short for documentation strings, or, more specifically, comments that explain the way a given procedure or a class works, inside the code itself), and so on.

Furthermore, we encourage the usage of type hints in function definitions as well as the usage of type annotations in docstrings. Line length should be no longer than 200 characters, which by itself is already stretching the recommended 88 character line length recommended by PEP8. Variable name and function name are to be written in `snake_case`, and class names are to be written in `PascalCase`. Similarly, the modules and the package itself are to be written in `snake_case`.

4.3.2 Docstrings

We use NumPy's code style guide for documenting classes and functions[4]. Using NumPy's style guide provides a clear and concise way of documenting the parameters and return values of functions and methods. It also has the added benefit of being familiar with other developers and researchers, as NumPy is a very popular library in the Python ecosystem, therefore potential users will be familiar with IDE-provided Intellisense help pop-ups on function definitions. Finally, it is handy as potential usage of a static documentation generator such as Sphinx[5] is made easier, as Sphinx is able to parse NumPy-style docstrings and generate documentation from them.

4.3.3 Linting

As part of the development process, we set up an automated linting service inside the IDE using the Pylint linter[6]. Pylint checks for errors in the code, as well as enforces the PEP8 code style guide. It also provides a variety of other checks, such as checking for unused imports, checking for unused variables, checking for undefined variables, and so on. We also use the Pylance language server[7] for the Visual Studio Code IDE, which provides a variety of other checks, such as type checking and errors arising from it.

4.3.4 Testing

We plan to implement minimal testing in order to avoid common errors and bugs. We will be using the pytest[8] framework for testing. Tests will be black-box style, input in, check output, and make

[3] `https://peps.python.org/pep-0008`
[4] `https://numpydoc.readthedocs.io/en/latest/format.html`
[5] `https://www.sphinx-doc.org/en/master/`
[6] `https://www.pylint.org/`
[7] `https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance`
[8] `https://docs.pytest.org/en/6.2.x/`

sure no errors are thrown.

4.3.5 Version Control

We will be using Git[9] for version control. Commits are to be routinely pushed to a central GitHub repository. Most code files are hosted in a monorepo containing the source code of the application, prototyping Jupyter Notebooks and tests, as well as the source of the very report being read here, and several other miscellaneous features.

4.4 Documentation

As mentioned, we use the NumPy code styling guide for documenting classes and functions, and part of the reason for that is the ease of use of the documentation generator Sphinx. Sphinx is a static documentation generator, which is able to parse NumPy-style docstrings and generate documentation from them. Sphinx-created files are then able to be hosted on a variety of platforms, such as GitHub Pages, or Read the Docs. In our use-case, we will be using Sphinx to generate documentation for the application, and will be hosting it on Read the Docs. Good documentation is essential, as we plan to release our package as an open-source library that can be installed and hosted on PyPi, and we want to make it as easy as possible for users to utilize the package to full effect.

[9] `https://git-scm.com/`

Chapter 5

Implementation

In the following chapter, we finally introduce the MAD HATTER package, a Python package for the analysis of linguistic features in text. The name borrows from the famous character of one of Lewis Carroll's most known works, "Alice in Wonderland". The Mad Hatter is a quirky character prone to speak in riddles and nonsensical sentences. The name is a nod to the fact that the package is designed to analyse creative text, and the Mad Hatter is a character that is often associated with (unconventional) creativity.

"We're all mad here. I'm mad. You're mad." – the Cheshire Cat, *Alice in Wonderland*

Nonetheless, we detail the steps we undertook to realize the project into a fully fledged package, along with its dependencies, usage examples, and a detailed description of the package structure. Furthermore, we provide justifications for the design choices we made and how the code has been implemented to be as efficient, modular and extensible as possible.

5.1 File Structure

The source code is structured as follows:

Item	Type	Description
docs	Directory	Contains the documentation code for the package.
madhatter	Directory	Contains the source code for the package.
notebooks	Directory	Contains the Jupyter notebooks used for the project.
tests	Directory	Contains the unit tests for the package.
.gitignore	File	Contains the files to be ignored by Git.
.readthedocs.yaml	File	Contains the configuration for ReadTheDocs' generator.
LICENSE	File	Contains the licence for the package.
pyproject.toml	File	Contains the configuration for the package. Additionally used by PyPi for managing dependencies and displaying basing info to potential users.
README.md	File	Contains a basic user guide for the package.

The file structure is organized in a way that should be familiar to more experienced Python developers and package maintainers, and enables user contributions in the future as we move on to share the package on the Python Package Index (PyPI) and share the source code on GitHub. As described above, the `madhatter` directory contains the complete source code for the package specifically. The other files complement it and ensure that the package is easily accessible to potential users via documentation and tests. Keeping this in mind, we can proceed to describe the structure of the package source code in more detail.

5.2 Package Structure

MAD HATTER is divided into several modules, each of which is responsible for a specific task. The package has been built to be as modular as possible to allow for easy extensibility and maintainability. The divisions are as follows:

Module	**Description**
`benchmark`	Contains the benchmarking suite, which is responsible for the evaluation of the text. The main class of `CreativityBenchmark` lives here.
`loaders`	Contains the data preprocessing pipeline and methods for downloading and loading static assets needed for either downloading testing suites or, more essentially, assets for benchmarking the text.
`models`	Contains methods for accessing language models that may be used if not otherwise supplied by the user themselves.
`utils`	Contains utility functions used throughout the package.
`metrics`	Contains key methods implementing metrics used throughout the package for the evaluation of the text.
`__init__.py`	The main entrypoint of the package. It bootstraps all essential modules and exposes the main classes and methods of the package to the user, for example, when they call `import madhatter` in their code.
`__main__.py`	The main entrypoint of the package when used as a CLI tool. Responsible for parsing the command line arguments and calling the appropriate methods to generate the report.

The files ending in the `.py` extension are the ones responsible for the actual implementation of the project, whilst the mirror files of the same name but ending in the `.pyi` extension are stub files meant to complement the actual implementation files with type annotations. The stub files are used by the `mypy` type checker to ensure that the code is type-safe, along with other code analysis tools used by various IDEs to provide rich type information for other developers using the package. Figure 5.1 provides a high-level overview of the main structure of the package.

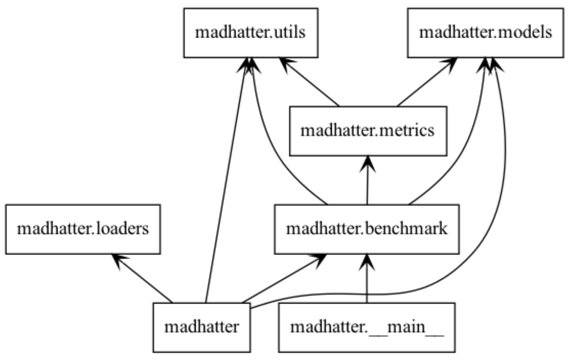


Figure 5.1: Detailed separation of the modules MAD HATTER is divided into along with their dependencies.

5.3 Benchmark Class

The main class for the text processing pipeline is the `CreativityBenchmark` class. The class contains all the methods needed to process the text and generate a report. The class is initialized with a text, and the text is processed through a pipeline of methods that are responsible for tokenizing, tagging, and lemmatizing the text.

5.3.1 Text Processing Pipeline

Upon initialization with a given text, the class preprocesses the text through a simple pipeline (visualized in Figure 5.2). Table 5.1 outlines the exact steps of the pipeline we apply in the implementation.

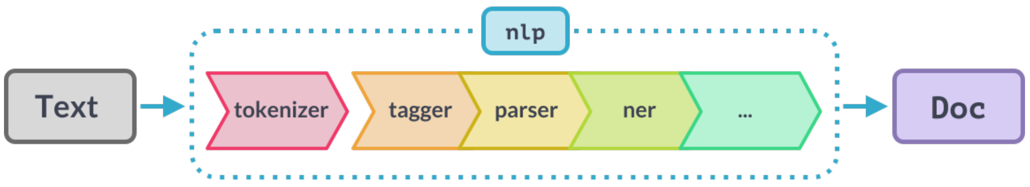


Figure 5.2: Example text processing pipeline. Resource made available by SpaCy documentation contributors.

Step	Description
Tokenization	Splitting the text into tokens, those being words (list of words), sentences (list of sentences), and words in sentences (list of words in list of sentences)
Tagging	Assigning a part-of-speech tag to each token, e.g. “running” → “verb”

Table 5.1: The steps of the text processing pipeline.

Following this initial process, all variables are accessible to users via the class’ instance variables, where some of the text processing ones have been listed in Table 5.2.

Variable	Description
`title`	The title of the text.
`words`	The text tokenized into words.
`sents`	The sentences of the text.
`tokenized_sents`	The sentences of the text both tokenized and POS-tagged.
`tagged_words`	Same as above, but the sentences are not separated.
`lemmas`	The words of the text lemmatized (reduced to their basic form, e.g. “running” → “run”).
`content_words`	The content words of the text (the words with specific very common words and expressions excluded).

Table 5.2: Basic instance variables for MAD HATTER.

After the initial processing, users are able to interact with the text via a variety of methods, the majority of which are listed in Figure 5.3.

Most users will likely be interested in the `report()` method, provided by the `CreativityBenchmark` class, which generates a `BookReport` object — a dataclass containing the results of the various benchmarks run on the text. The report function has several feature flags that may include or exclude certain features from the report. The flags are as follows:

- `include_pos`: whether to include the POS tag distribution in the report.
- `include_llm`: whether to include the LLM metrics in the report.
- `kwargs`: other keyword arguments to adjust the output of the LLM metrics.

Furthermore, the class provides basic plotting utility functions for the user to display some basic information about the text. Some of them may be more useful than others depending on the text. We showcase the following ones:

`plot_postag_distribution()` plots the POS tag distribution of the text over its length. Users may be interested in seeing where in the text certain POS tags are more prevalent, e.g. some users may want to narrow down on sections where there is a high concentration of adjectives – such as passive scenes where there may be some description of the setting of a novel, for example – or verbs – where there may be a lot of actions going on.

`plot_transition_matrix()` plots the transition matrix of the POS tags in the text. This a matrix-like structure displaying which POS tags follow which other POS tags, e.g. determiners are likely to be followed by adjectives and nouns, adjectives are likely to be followed by nouns, and so on. This may be useful for users to get a sense of the structure of the text.

`plot_report()` plots the report generated by the `report()` method. This may be useful for users to get a sense of the overall metrics of the text. It provides an intuitive interface in the form of a spider chart displaying how the text performs on each of the metrics compared to some global norm. The method can be customized with different norms depending on the type of text the user is analysing. For example, legal documents, news articles and short stories vary a lot in terms of structure, so if we apply one single norm, we will see that for example legal documents tend to have much longer sentences than either of the two other categories. Thus, using a common norm for all text documents may not be the best approach to evaluating the text as opposed to others in the said field.

Other than those, all methods returning text metrics provided by the `CreativityBenchmark` class are easily plottable through packages such as `matplotlib` or `seaborn`, can be reliably converted into `pandas` dataframes or `numpy` arrays, and are exportable to any of the commonly used data formats such as `JSON`, `CSV`, or `Apache Parquet` for storage. Furthermore, the Jupyter notebooks provide specific examples for how to use the package in a number of scenarios, ranging from a complete beginner to a trained data analyst.

5.4 Implemented Metrics

We detail the implementation of the metrics described in 3.3 in the following sections. Each metric is provided with a brief description, its purpose, and how it is implemented in the package.

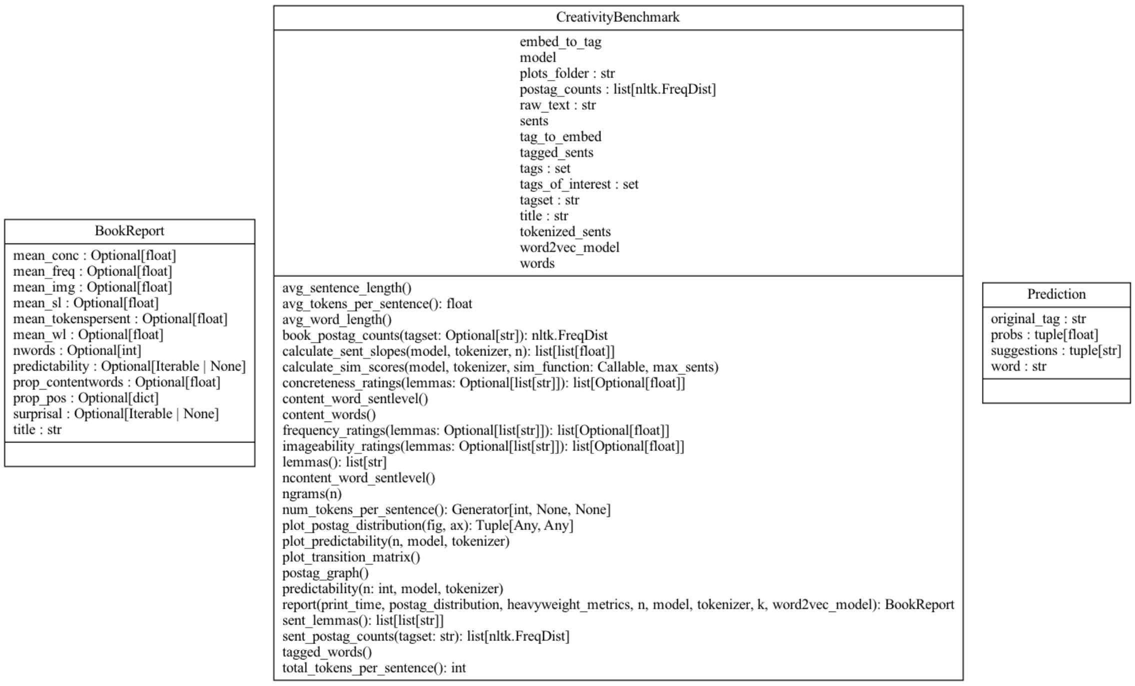

CreativityBenchmark
embed_to_tag
model
plots_folder : str
postag_counts : list[nltk.FreqDist]
raw_text : str
sents
tag_to_embed
tagged_sents
tags : set
tags_of_interest : set
tagset : str
title : str
tokenized_sents
word2vec_model
words
avg_sentence_length()
avg_tokens_per_sentence(): float
avg_word_length()
book_postag_counts(tagset: Optional[str]): nltk.FreqDist
calculate_sent_slopes(model, tokenizer, n): list[list[float]]
calculate_sim_scores(model, tokenizer, sim_function: Callable, max_sents)
concreteness_ratings(lemmas: Optional[list[str]]): list[Optional[float]]
content_word_sentlevel()
content_words()
frequency_ratings(lemmas: Optional[list[str]]): list[Optional[float]]
imageability_ratings(lemmas: Optional[list[str]]): list[Optional[float]]
lemmas(): list[str]
ncontent_word_sentlevel()
ngrams(n)
num_tokens_per_sentence(): Generator[int, None, None]
plot_postag_distribution(fig, ax): Tuple[Any, Any]
plot_predictability(n, model, tokenizer)
plot_transition_matrix()
postag_graph()
predictability(n: int, model, tokenizer)
report(print_time, postag_distribution, heavyweight_metrics, n, model, tokenizer, k, word2vec_model): BookReport
sent_lemmas(): list[list[str]]
sent_postag_counts(tagset: str): list[nltk.FreqDist]
tagged_words()
total_tokens_per_sentence(): int
BookReport
mean_conc : Optional[float]
mean_freq : Optional[float]
mean_img : Optional[float]
mean_sl : Optional[float]
mean_tokenspersent : Optional[float]
mean_wl : Optional[float]
nwords : Optional[int]
predictability : Optional[Iterable | None]
prop_contentwords : Optional[float]
prop_pos : Optional[dict]
surprisal : Optional[Iterable | None]
title : str
Prediction
original_tag : str
probs : tuple[float]
suggestions : tuple[str]
word : str

Figure 5.3: Available classes in MAD HATTER along with their class and instance methods and variables.

We divide the metrics into two main classes: **lightweight** and **heavyweight** depending on their complexity and the time it takes to compute them. The lightweight metrics are usually computed in linear time and relatively quick, whereas the heavyweight metrics tend to have a more complex computation mechanism and therefore tend to be slow.

5.4.1 Simple Features

Simple features are typically trivial to implement and do not pose a significant challenge to the implementation, but they are nonetheless useful for the evaluation of the text and in specific scenarios extremely descriptive, while in others not so much but still provide a distinguishing factor for the text. These usually take advantage of Python's built-in data structures and methods, such as `len()`, `sum()`, `set()`, etc. and are implemented through simple list comprehensions or generator expressions (note that these are, in fact, faster than typical for-loops in Python).

Number of Words returns the total number of words in the text. This is a simple metric that is useful for the evaluation of the text.

Average Word Length returns the average word length of the text in characters. A text with a high average word length may be more verbose and descriptive, whereas a text with a low average word length may be more concise and simpler to read.

Average Sentence Length returns the average sentence length of the text in characters. High average sentence length points to verbosity, very evident in legal documents. Texts with a low average sentence length may be more concise and simpler to read.

Average Tokens per Sentence returns the average number of tokens per sentence. Counts the number of tokens and averages over the total number of sentences. Relatively correlated with both the average sentence length and average word length.

Proportion of Content Words returns the proportion of content words in the text. Content words are words that are not stopwords. Stopwords are usually very common (such as "I, we, they, can, not, do, etc...”), and do not provide much meaning to the text.

Implemented with the use of an efficient HashSet lookup, the method counts the number of content words and divides them by the total number of words in the text. The proportion of content words is a good indicator of the complexity of the text and the writing skill of the author of the text.

5.4.2 Complex Features

Complex features are usually more difficult to implement and require more complex data structures and algorithms to compute. These are usually implemented with lists to leverage the strengths of Pythons, and are computed with the use of lookups in fast data structures. Those typically refer to features such as **concreteness**, **imageability**, and **rare word usage**.

Concreteness refers to the linguistic characteristic described by Brysbaert et al. (2014) and is discussed more in depth in Section 3.3.6. The metrics are stored in a CSV file containing the concreteness scores for the available words in the corpus by Brysbaert et al. (2014). The file is loaded into memory using a `pandas` dataframe. We implemented various methods to speed up the execution of the concreteness metric, such as caching the dataframe in memory, using fast dataframe lookups, re-indexing the dataframe with the words as index, and sorting. However, we found the most significant speed-up to be to convert the dataframe into a dictionary (HashMap). Dictionaries are highly-optimized in Python and are especially useful for fast string lookups. Therefore, we convert the dataframe into a dictionary with the words as keys and the concreteness scores as values. If a word cannot be found, the None (null) value is returned in order to avoid ambiguity with other potential replacements such as 0, some arbitrary value, or NaN (not-a-number). The concreteness score of a text is represented as the concreteness score of all of its words. The `report` function returns the mean concreteness score of the text.

Importantly, the words in the concreteness corpus are lemmatized, thus, we lemmatize the words in the text before looking them up in the dictionary. This is also a relatively fast operation, and we use the `WordnetLemmatizer` object provided by the NLTK library.

Imageability is the characteristic of words that describes how easily they can be visualized, and has been described in **??**. The metric is implemented similarly to the concreteness metric. The words are present inside a CSV file that is loaded with the use of a dataframe and converted into a dictionary mapping the word (more precisely, its lemma) to its mean imageability score. The imageability score of a text is the mean imageability score of all of its words.

Rare Word Usage refers to how frequently one uses not so-frequently seen words in the text. Usually this measure, however, varies wildly depending on the genre. For example, "spine" may be a somewhat unusually seen word in general English, maybe 1 in 300,000. However, in a medical text, it may be a very common word, maybe 1 in 10,000, a difference of potentially several magnitudes. Thus, the selection of corpus can affect this metric wildly and as such, we recommend most people to use a specific corpus or word count lists in their own context. In the application, we utilize a general word frequency list based on the BNC (British National Corpus) as described in 3.3.8.

Yet again, the results are stored in a CSV file that is cleaned and loaded into a dataframe. The column containing word frequency records occurrences per 1,000,000 words. We take the logarithm of base 10 for those with the following two motivations.

1. Because word frequency closely follows Zipf's Law, as explained by authors such as Powers (1998), and;
2. The logarithm of base 10 is a monotonic transformation of the original data, thus, it preserves the order of the data. This is important because we want to be able to compare the results of the metric across different texts. It furthermore nicely constrains the range of values between 0 ($10^0 = 1$) and 6 ($10^6 = 1,000,000$), which is a more manageable range than the original data.

After the transformation is applied, we construct a dictionary with the lemmas and their respective frequencies for fast lookups. Results are returned as Python lists, and, if a word cannot be found, again, *None* is returned. The rare word usage score of a text is the mean frequency of all of its words.

5.4.3 LLM-based Features

We introduce two metrics based on Masked Language Model (MLM) word masking. Firstly, we introduce the relevant functions for extracting the MLM predictions for a given text.

Process

The process is as follows: we mask a word in a given section of the text, and we ask the model to predict the masked word. The model returns a probability distribution over the vocabulary, and we take the top K likeliest words (sorted by likelihood to be the masked word) - this is all done by the `metrics.predict_tokens()` function which returns `Prediction` objects. In all cases we use the BERT (Bidirectional Encoder Representations from Transformers) model introduced by Devlin et al. (2019), and more specifically, `bert-base-uncased`, as provided by HuggingFace's `transformers` library.

`Prediction` objects are a dataclass instance containing the following fields:

- `word`: the token that was masked.
- `tag`: the original POS tag of the word.
- `suggestions`: a list of the top K predictions for the masked token.
- `probs`: a list of the likelihoods of the top K predictions for the masked token.

Essentially, the text we use as context can be one of two things:

1. **Individual Sentences** – we mask a word in a sentence and ask the model to predict it. This is useful for evaluating the text on a sentence-by-sentence basis. This is defined as the function `models.sent_predictions()`.

2. **Sliding Window** – we mask a word in a sliding window of a given length in tokens. This is useful for evaluating the text on a more global level. This is defined as the function `models.sliding_window_preds()`.

The default behaviour here is to use a sliding window of 20 tokens before and after, but this can be adjusted by the user. Below we show an example for a short sentence split into words and sliding window length of 3 tokens before and after.

The | quick | brown | fox | jumped | over | the | lazy | dog | . | ... |

The algorithm carries out a few steps before trying to predict for the word: firstly, it checks if the word is not present in a list of stopwords, and secondly, it checks if the word is of a tag we may be interested in. If any of the two conditions fail, we skip the word and continue on. For example, users can decide whether they do not want to predict for nouns or verbs, etc. This cuts down on computation time and avoids possibly uninteresting predictions. Likewise for stopwords, we may not be interested in predicting for words such as “the”, “a”, “an”, etc.

Now, we attempt to predict for the word *fox* and a sliding window of 3. This means that the MLM receives the following input: *“the quick brown [MASK] jumped over the”*. For this particular example, the model suggests the following results:

Likelihood	Token
9.453183	eyes
8.290386	man
8.150213	foxie
8.013705	cat
...	
6.271477	bird

We record those as a `Prediction` object and move on to the next word and its context. We do this for all possible words and obtain a list of `Prediction` objects, which we can then use to compute the predictability metric.

LLM-based Metrics

Additionally, we apply methods for context-dependent LLM-based feature metrics defining two novel metrics for the evaluation of the text. These we name the **surprisal** and **predictability** metrics. We describe the implementation of these metrics in the following sections.

Predictability has been defined as a measure of a Masked Language Model (MLM)'s confidence in predicting a masked word given some context. Now, **predictability is defined as the averaged gradient of the top K suggestions in the probability distribution over the vocabulary**. The gradient is computed using second order accurate central differences in the interior points and either first or second order accurate one-sides (forward or backwards) differences at the boundaries.

In layman's terms, at the boundaries, we calculate only the first difference. This means that at each end of the array, the gradient given is simply, the difference between the end two values, divided by 1. Away from the boundaries, the gradient for a particular index is given by taking the difference between the values on either side and dividing by 2. For example,

$$\begin{aligned} y[0] &= \frac{y[1]-y[0]}{1} \\ y[1] &= \frac{y[2]-y[0]}{2} \\ y[2] &= \frac{y[3]-y[1]}{2} \\ &\dots \\ y[n-1] &= \frac{y[n]-y[n-2]}{2} \\ y[n] &= \frac{y[n]-y[n-1]}{1} \end{aligned}$$

We do this over all possible suggestions and obtain a list of differences, e.g. $[-1.16279697, -0.65148497, -0.13834047, \dots, -0.14180756]$ for the example predictions provided above. **Finally, we take the average of this list to get a sense for how rapidly the probability distribution's confidence falls**. The lower the value (it is always negative, as the list is always sorted and therefore the values will always be negative), the more rapidly the confidence falls, and the more certain the model is that the word is one of these K suggestions.

For more intuitive understanding of this metric by users, we take the absolute value of the average gradient. We reason this by saying that the intuition of *high values correspond to high predictability and low values correspond to low predictability* is more natural than low (negative) values correspond to high predictability and high (negative) values to low predictability. It may be more accurate to call this a measure of the model's certainty on the word given the context, rather than predictability of the word given the context.

Surprisal is a derivative metric using the list of `Prediction` objects returned by either of the two methods described above. Surprisal is defined as the average similarity between the top K suggestions and the original word, given by some similarity metric. We implement several methods for computing the similarity between two words, namely:

- **Cosine Similarity** (the default method). The cosine similarity between two word vectors. The words are turned into dense vectors using Word2Vec vectors (described in **??**) and the similarity is computed using the cosine similarity metric.
- **WordNet Lin Similarity** – the WordNet Lin similarity between two words. The words are turned into synsets using WordNet synsets (described in **??**). Because words can have multiple recorded WordNet meanings, we avoid adding additional overhead of disambiguating the meaning of the words (and we cannot be sure since the nature of the MLM prediction does not return a tag, just some token), and instead fall back on the

heuristic of returning the most common word sense given a POS tag, e.g. "fox.n.01". Since we already know the POS tag with high degree of certainty. The similarity is computed using the WordNet Lin similarity metric which is based on the Information Content (IC) of the Least Common Subsumer (most specific ancestor node) and that of the two input synsets. The relationship is given by:

$$\text{Lin}(s_1, s_2) = \frac{2 * IC(lcs)}{IC(s_1) + IC(s_2)} \tag{5.1}$$

For this metric, we need some information content, which is a measure of how specific a synset is. By default, we use the information content provided by the NLTK WordNet corpus, which is based on the Brown corpus, but this can be adjusted by the user.

- **WordNet Wu-Palmer Path Similarity**. The words are turned into synsets using WordNet synsets (described in **??**). As above, we only choose the most common WordNet sense. The similarity is computed using the Wu-Palmer similarity metric which is based on the shortest path between the two input synsets. The relationship is given by the equation:

$$\text{Wu-Palmer}(s_1, s_2) = \frac{2 * \text{depth}(lcs)}{\text{depth}(s_1) + \text{depth}(s_2)} \tag{5.2}$$

  This similarity metric has the benefit of not requiring an additional Information Content (IC) dictionaries.

To reiterate, **the similarity is computed by taking the average of the similarity scores between the original word and the top K suggestions**. The higher the value, the more similar the suggestions are to the original word, and the less surprising the word is given the context. Furthermore, most of the time, the similarity is computed using Word2Vec vectors, which are dense vectors, and therefore the similarity is nicely constrained between 0 and 1.

5.5 Command-Line Interface

MAD HATTER is also available as a command-line interface (CLI) tool. The CLI tool is implemented in the `__main__.py` file and is responsible for parsing the command line arguments and calling the appropriate methods to generate a BookReport object which is printed to the console. The tool primarily takes in a text file as input and outputs a report of the text.

The tool enables "quick and dirty" usage of the package, and is useful for users who want to quickly evaluate a text without having to write any code. The tool provides several options to customize the report object, such as the usage of the heavyweight LLM metrics, the ability to include POS tags, specify context length for LLM predictions, and so on. We plan to extend this tool with plotting capabilities in the future, but for now, the tool is limited to printing the report to the console.

5.6 Dependencies

MAD HATTER is built on top of several open-source libraries and packages. We list the most important ones below:

NLTK (Recommended Version: 3.7.0)

The Natural Language Toolkit (NLTK) is a Python library for Natural Language Processing (NLP) tasks. It provides a wide variety of tools for text processing, such as tokenization, tagging, lemmatization, and so on. We use NLTK for the majority of the text processing pipeline, and it is a crucial dependency for the package. The initial preprocessing pipeline is entirely carried out through NLTK, and the package would not be possible without it.

NumPy (Recommended Version: 1.23.4)

NumPy is a Python library for scientific computing. It provides a powerful N-dimensional array object, along with a variety of tools for working with these arrays. We use NumPy for the majority of the data processing and computation in the package that deals with the MLM metrics.

HuggingFace Transformers (Recommended Version: 4.27.1)

HuggingFace Transformers is a Python library implementing the Transformer neural network architecture as well as methods for loading and using pre-trained Transformer models from the HuggingFace Hub website[1]. We use HuggingFace Transformers for the implementation of the LLM metrics. The library provides a simple and intuitive interface for loading and using pre-trained models, and it is a crucial dependency for the loading of the MLM models.

PyTorch (Recommended Version: 1.13.1)

We already explained the needs for PyTorch in Section 4.2.3. We use PyTorch for the implementation of the BERT models used for the LLM-based metrics. The `transformers` library provides a simple interface for loading the models, and PyTorch is used as the backend for the models.

Gensim (Recommended Version: 4.3.0)

Gensim is a Python library for topic modelling, document indexing, and similarity retrieval with large corpora. We use Gensim for the implementation of the Word2Vec vectors used for the similarity metrics. The library provides a simple interface for loading the models, and it is a crucial dependency for the loading of the Word2Vec models.

pandas (Recommended Version: 1.5.2)

pandas is a Python library providing high-performance, easy-to-use table data structures called DataFrames, as well as N-dimensional vectors called Series. The library provides a simple interface for loading and working with CSV files, and it is a crucial dependency for the loading of the concreteness, imageability, and rare word usage metrics.

Furthermore, it is used for the additional experiments we carry out in Section 6.1 for preprocessing the data and then further on usage in the machine learning pipeline we implement to test the efficacy of MAD HATTER.

matplotlib (Recommended Version: 3.6.2) and

seaborn (Recommended Version: 0.12.2)

Matplotlib is the most widely used Python library for plotting and data visualization. Seaborn is a Python library built on top of matplotlib providing a high-level interface for statistical data visualization. We use both libraries for the plotting utilities in the package, as well as visualizations for the experiment section.

[1]`https://huggingface.co/`

Others

We also list some additional libraries which bear mention but are not as crucial to the package as the ones listed above. These are:

Name	**Version**	**Description**
tqdm	4.64.1	Displaying progress bars, e.g. during data loading, benchmarking.
requests	2.30.0	Making HTTP requests, e.g. for data loading.
scikit-learn	1.2.0	Basic machine learning, implementation of simple pipelines.
scipy	1.9.3	Dependency of scikit-learn.

Chapter 6

Evaluation

In the context of application, we opt to implement several experiments that give a better understanding of the potential applications of MAD HATTER.

Component	**Description**
CPU	2.3 GHz Intel Core i9-9800H
RAM	16 GB DDR4 2400 MHz
GPU	Intel UHD Graphics 630 / Radeon Pro 560X
OS	macOS 13.1

Table 6.1: Specifications of the computer used for the experiments.

Wherever not explicitly mentioned, assume the specifications listed in Table 6.1.

6.1 Experimental Design

In this section, we describe the experiments we conducted to evaluate the performance of MAD HATTER. We start by describing the datasets we used for the experiments. Then, we describe the experiments we conducted and the metrics we used to evaluate the performance of MAD HATTER. Finally, we describe the baselines we used for comparison.

6.1.1 Datasets

Table 6.2 describes the utilized datasets along with their specific application in the experiment. Further descriptions of the datasets can be found at section 3.1.

Experiment	Dataset(s)
Document Class Identification	1. Project Gutenberg (PG) 2. EU DGT-Acquis & Europarl Corpus [NLTK] (Legal) 3. r/WRITINGPROMPTS (WP)
Authorship Identification	Project Gutenberg (PG) Up to 30 works from the 1000 most prolific authors
Machine-Generated Text Detection	1. WebText (representing real text) 2. Generated texts from GPT-2 XL-1542M

Table 6.2: Listing with the datasets used for the experiments.

6.2 Experiments

We implement three different experiments as a way of evaluating the performance of the application. The first experiment is a document class identification experiment, where we evaluate the performance of MAD HATTER in identifying the class of a document, thus demonstrating that the features we implement are well-defined and enable differentiating between different types of writing. We then move on to evaluating how well the algorithm can differentiate between different writing styles, a task also known as authorship identification. Finally, we follow the logical progression of authorship identification to address a topic that has been gaining traction in recent years, that of machine-generated text detection. This may have further applications in the future, as the field of natural language generation has been steadily growing in the past few years, with the advent of LLM such as GPT-2 (Radford et al., 2019) and GPT-3 (Brown et al., 2020).

6.2.1 Document Class Identification

In this experiment, we evaluate the performance of MAD HATTER in identifying the class of a document. The datasets, described in Table 6.2, form the basis of the classes we designate, those being: (conventional) fictional literature (Project Gutenberg / PG), legal texts from the EU DGT-Acquis as well as the Europarliament Corpus distributed with NLTK (Legal / LG), and short-form stories from the subforum WRITINGPROMPTS of the social media platform REDDIT(WP).

Setup

Initially, all distinct texts are split into chunks of 100,000 characters (with the trailing chunk on its own). This is done primarily to maximize the potential data points of the dataset, but also to speed up the processing of the algorithm for large texts (for example, the texts in PG dataset are usually long-form full books which have upwards of 600,000 characters, assuming a ratio of 100,000 characters per 60-70 pages of text in traditional font and size). Normally, this may carry a potential for overfitting, as the chunks may not be representative of the whole dataset. However, as the texts are 1) very distinct from each other, and 2) have been shown to not split to more than 6-7 chunks, this is not a concern. The datasets are run through a simple pipeline that generates the features described in Section 3.3. For more flexibility in combining and comparing the datasets for classification, each dataset is separately run through the pipeline. After the features are extracted, each dataset is assigned its respective category. The datasets are then combined and shuffled.

The combined dataset is split into a training, validation, and test set, with a ratio of 80:10:10. The training set is used to train a logistic regression with L2 penalty, which is then used to predict the class of the documents in the test set. As an intermediary step, we run a grid search with the training dataset and the validation dataset in order to find the best parameter for the inverse of regularization strength of the algorithm. The parameter is chosen from the set $\{\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8, 16, 32, 64\}$. The parameter with the highest accuracy on the validation set is chosen for the final model. The accuracy of the model is then evaluated on the test set.

Results

Table 6.3 shows the performance results for the document classification experiment. The results show that MAD HATTER is able to identify the class of a document with a very high accuracy. This is not surprising, as the classes are very distinct from each other, yet it affirms that the implemented features capture well specific characteristics of the text. The results also show that the model is

Table 6.3: Performance results for Document Classification

Experiment	Document Classification
Size of Train Set	4686
Train Accuracy	99.827%
Validation Accuracy	99.808%
Test Accuracy	99.827%

not overfitting, as the accuracy on the test set is very similar to the accuracy on the training set.

It should be noted that, despite the size of the training set is relatively small as opposed to other experiments in the field of document classification, the accuracy achieved is remarkably high. This is due to the fact that the features used are very simple and straightforward, and thus do not require a large amount of data to be learned. Furthermore, the algorithm is a step-up in terms of speed from existing baselines such as SVMs and TF-IDF algorithms, which makes it more suitable for large datasets and big scale text analysis. Figure 6.1 shows the confusion matrix for the document classification experiment. As seen, the document is able to distinguish between the classes with an excellent accuracy, precision and recall.

Discussion

Via the algorithm, the classes have been shown to not only be evidently distinct on their own, but also in terms of the features used. The features used in the experiment are very simple and straightforward, and thus do not require a large amount of data to be learned. Potential applications for document classification may include categorizing documents in a large database or potential dataset. Categorization can possibly be applied for sentiment evaluation for product reviews, social media posts, and so on. We go on to explore other potential uses of the algorithm in the following experiments.

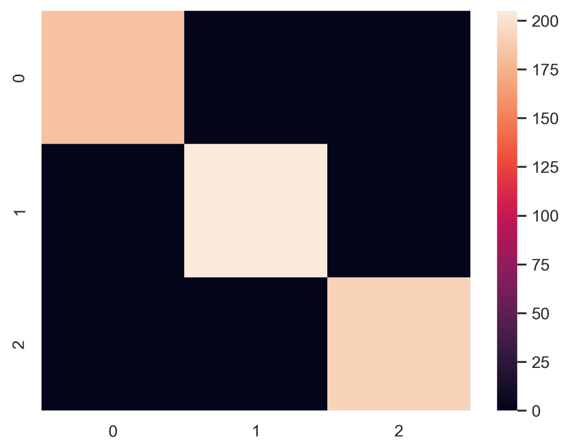


Figure 6.1: Confusion Matrix for Document Classification. The rows represent the true labels, while the columns represent the predicted labels.

6.2.2 Authorship Identification

After we have identified that the features are able to distinguish between different classes of documents, we now ask, “Can a machine distinguish between texts from the same class?”

This is a homogenous classification task, where the classes are very similar to each other, and the specific task is to identify the author of a text, given a set of candidate authors. For example, given the text of "Alice in Wonderland", MAD HATTER's task is to identify that the author is Lewis Carroll. The task is a natural progression from the document classification task and a more fine-grained existing problem in the field of NLP.

Setup

We make use of the work done by Gerlach and Font-Clos (2018) to standardize the Project Gutenberg for data exploration and analysis. We filter out for the most prolific 1000 authors in the available non-copyright literature. Furthermore, we randomly sample a maximum of 30 works per author. This is done in order to avoid overfitting for the more prolific authors (number one has more than 300), and even then only the top 200 authors have more than 30. A single chunk of 100,000 characters is then taken from each text and added to a list for processing.

The pipeline is similar to the one listed for document classification. We process all samples and obtain the features described in Section 3.3. The dataset is then split into a training, validation, and test set, with a ratio of 80:10:10. The features are then standardized (subtracting the mean and dividing by the standard deviation for each column). The training set is used to train a logistic regression with L2 penalty, which is then used to predict the author of the documents in the test set. As an intermediary step, we run a grid search with the training dataset and the validation dataset in order to find the best parameter for the inverse of regularization strength of the algorithm. The parameter is chosen from the set $\{\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8, 16, 32, 64\}$. The parameter with the highest accuracy on the validation set is chosen for the final model. The accuracy of the model is then evaluated on the test set.

Experiment	A. Id. ($n = 1000$)	A. Id. ($n = 50$)
Size of Data (train/val/test)	17306:962:961	1290:72:72
Accuracy (train/val/test)	26.83%/23.91%/20.19%	55.50%/51.39%/56.94%
Precision	0.229/0.159/0.132	0.528/0.453/0.408
Recall	0.249/0.19/0.164	0.539/0.414/0.529
F1-Score	0.216/0.158/0.134	0.518/0.414/0.431

Table 6.4: Performance results for Authorship Identification ($n = 1000$ and $n = 50$).

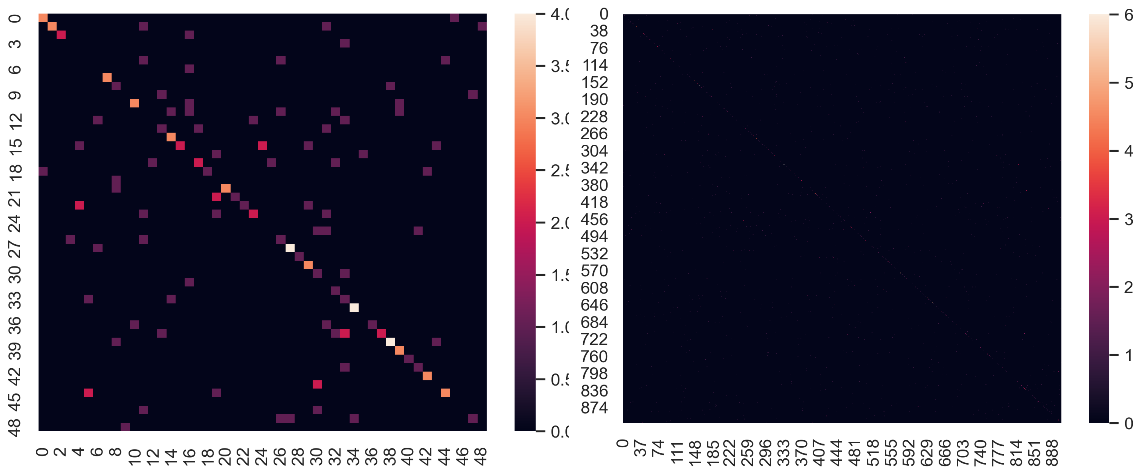
Results

Table 6.4 details the results of the case study for both the 50 authors case and the 1000 authors case. Accuracy, precision, recall and F1 scores are reported. The model is able to distinguish between authors with an accuracy of 55.50% for the 50 authors case, and 26.83% for the 1000 authors case. The results also show that the model is not overfitting, as the accuracy on the test set is similar to the accuracy on the training set. Although not clearly visible, Figure 6.2 shows the confusion matrix for the authorship identification experiment.

Unfortunately, the model struggles to achieve high accuracy for a huge number of authors, in our case the total number being a thousand. However, if we consider a baseline of a simple coin flipping, that is, a model that will randomly assign a label from the available labels (authors) to each work with a probability of $\frac{1}{1000}$ (for the bigger classification case), we can see that the model does perform relatively well in distinguishing some features that are similar between authors. The confusion matrix potentially indicates similar features between authors, a characteristic that may be a topic for further research. Note the highest number of correct predictions (6 and 6 out of a maximum of 30) respectively in the middle of the matrix in 6.2a and towards the tail end of the diagonal of 6.2b (around (855,855)). The other values are relatively low, with the highest number of incorrect predictions being 3 and 3, respectively.

The results are, in fact, in line with the results of Qian et al. (2019), who report an accuracy of 69.1% on the Reuters 50_50 (C50) dataset and 89.2 % on the Gutenberg dataset (for a maximum of 50 authors and 45000 paragraphs of text from their works). The authors used sentence- and article-level GRUs and an article-level LSTM neural network to achieve these results. The authors also provided a baseline accuracy of 12.24% via Gradient Boosting Classifier with 3 features, those being average word length, average sentence length, and Hapax Legomenon ratio (fraction of unique words), 2 of which we used. Given this baseline and the trivial computational complexity of our experiments, we have a reason to believe that we surpass the baseline and the researched features do enable stronger distinction between authors in this classification task.

Potential areas for improvement include the use of more sophisticated features, such as the ones described in **??**, as well as the use of more sophisticated models for multiclass classification, such as SVMs, neural networks, or Naive Bayes classifiers. We then move on to the next case study, which is the identification of machine-generated text.



(a) $n = 50$ **(b)** $n = 1000$

Figure 6.2: Confusion Matrices for Authorship Identification. The rows represent the true labels, while the columns represent the predicted labels.

6.2.3 Machine-Generated Text Identification

Having tested our hypothesis that the features are able to distinguish between different classes of documents in section 6.2.1, and then identified that the features are able to distinguish between different authors in section 6.2.2, we now move on to the ambitious goal of identifying machine-generated text. LLMs have seen explosive growth in the past few years, and generating human-like text, both grammatically and (somewhat) logically-sound, is now far from a distant dream. However, the ability to generate text indistinguishable from human-written text has raised concerns about the potential for misuse of such models. Particular issues may arise for example with academic grading, as AI writing tools become more and more prevalent. Fields other than academia may also be affected, such as journalism, where AI writing tools may be used to lazily generate non-proofread articles with the potential to spread misinformation. Of course, the potential for misuse is not limited to the above examples, and there are plenty of uses that malicious agents can come up with, either for personal gain or for the sake of spreading chaos.

Now then, we arrive back at the essence of the problem we are trying to solve: "What defines creativity? What defines human creativity?" The answer to this question is not simple, and it is not the goal of this thesis to answer it in full. However, we can attempt to answer a more specific question: "Can a machine distinguish between human-written text and machine-generated text?"

Setup

We make use of the WebText dataset (Radford et al., 2019), which is a large dataset of text scraped from the internet and used to train the influential GPT-2 (Generative Pre-trained Transformer), a LLM developed by OpenAI. Texts generated by GPT-2 themselves serve as the basis for the "machine-generated" classification labels and are labelled as Machine-Generated Text (MGT). Samples from the WebText dataset used to train GPT-2 form the basis for the "human-written" classification labels. 20,000 samples are randomly drawn from the set of machine-generated texts, and 20,000 — from the set of human texts.

The dataset is split into a training, validation, and test set, with a ratio of 80:10:10. The training set is used to train a logistic regression with L2 penalty, which is then used to predict the class of the documents in the test set. As an intermediary step, we run a grid search with the training dataset and the validation dataset in order to find the best parameter for the inverse of regularization strength of the algorithm. The parameter is chosen from the set $\{\frac{1}{64}, \frac{1}{32}, \frac{1}{16}, \frac{1}{8}, \frac{1}{4}, \frac{1}{2}, 1, 2, 4, 8, 16, 32, 64\}$. The parameter with the highest accuracy on the validation set is chosen for the final model. The accuracy of the model is then evaluated on the test set.

Results

The model reports an accuracy of 69.7% on the training set, 70.0% on the validation set, and 70.0% on the test set. The results show that the model is not overfitting, as the accuracy on the test set is similar to the accuracy on the training set. Figure 6.3 shows the confusion matrix for the machine-generated text detection experiment. As seen, the document is able to distinguish between the classes with a passable accuracy, precision and recall.

Table 6.5: Performance results for MGT Detection

	Split		
	Train	Val	Test
Size of Data	32000	4000	4000
Accuracy	0.697	0.700	0.700
Precision	0.698	0.702	0.700
Recall	0.697	0.700	0.700
F1-Score	0.697	0.700	0.700

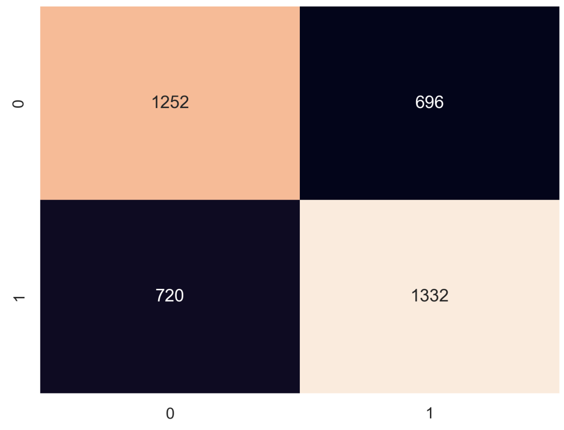


Figure 6.3: Confusion Matrix for MGT Detection. The rows represent the true labels, while the columns represent the predicted labels.

Discussion

The results displayed by the model show some promise, especially given the trivial nature of the features we studied. Accuracy if higher, could be used to detect machine-generated text with a high degree of certainty. However, the results are not as high as we would like them to be, and there is a lot of room for improvement. As discussed in prior points, potential areas for improvement include the use of more sophisticated features, such as the ones described in **??**, as well as the use of more sophisticated models for multiclass classification, such as SVMs, neural networks, or Naive Bayes classifiers.

The authors of GPT-2 provide a baseline of their own [1], citing 74.31% accuracy for the temperature 1-sampled output of the XL version of the GPT-2 model (1582 billion parameters), and 92.62% for the K40-sampled output of the same model. Again, our results only serve as a demonstration of the accuracy of the MAD HATTER package, rather than a definitive leap in the field of machine-generated text detection. We do look forward, however, to more expansive testing of the package, as well as the implementation of more sophisticated features and models in the future.

6.2.4 Summary

In this section, we have implemented three experiments to evaluate the performance of MAD HATTER. We started by evaluating the performance of MAD HATTER in identifying the class of a document, thus demonstrating that the features we implement are well-defined and enable differentiating between different types of writing. We then moved on to evaluating how well the algorithm can differentiate between different writing styles, a task also known as authorship identification. Finally, we followed the logical progression of authorship identification to address a topic that has been gaining traction in recent years, that of machine-generated text detection. This may have further applications in the future, as the field of natural language generation has been steadily growing in the past few years, with the advent of LLMs such as GPT-2 (Radford et al., 2019) and GPT-3 (Brown et al., 2020).

[1] `https://github.com/openai/gpt-2-output-dataset/blob/master/detection.md`

Chapter 7

Discussion and Conclusion

In the following chapter, we discuss the results of the experiments conducted in the previous chapter. We also discuss the limitations of the system and the results, as well as the future work that can be done to improve the system. Finally, we conclude with a summary of the contributions of this work.

7.1 Formal Evaluation

We introduced MAD HATTER, a text processing and a linguistic benchmarking tool, as well as provided a set of benchmarks for evaluating the creativity of text. We have also tested our benchmarks on a set of texts, and provided a set of experiments that can be used to evaluate the system.

7.1.1 Evaluation of the system

The system for benchmarking text we implemented is a useful tool for text analysis, that fills in a niche in the Python ecosystem for textual analysis that has not been filled before – that of a tool for benchmarking text. As we mention, the system is also a strong tool for text analysis that can be used for a variety of baseline NLP tasks, such as text tokenization, part-of-speech tagging, and word frequency analysis.

MAD HATTER occupies a similar category of text analysis tools such as the Natural Language Toolkit (NLTK) (Bird et al., 2009), the TextBlob library (Loria, 2018), and the spaCy library (Honnibal and Montani, 2017). However, MAD HATTER is the only tool that provides a set of benchmarks for evaluating the creativity of text and handy operations to aid with visualization.

Similarly to the tools above, Mad Hatter is extensible and can be used to build more complex tools for text analysis. Because MAD HATTER has been built on top of the NLTK library, it can be extended with any of the tools provided by NLTK in the future, as well. Yet, the implemented methods are also framework-agnostic, as the main functions have been built on top of pure Python, and would require minimal configuration to be borrowed by other frameworks such as `SpaCy`, `gensim`, and others. Furthermore, as a significant portion of the project was spent on researching methods for evaluating creativity in text, and the documentation in the source code is exhaustive, the project is easily portable in other, more performant programming languages such as C++ or Rust. We hope that this project can be used as a starting point for future research in the field.

7.1.2 Evaluation of the benchmark

We carried out an evaluation on the majority of the metrics implemented in the benchmark, for the purposes of determining the validity of the metrics. We examined the metrics' effectiveness in determining the source of the text – their genre, e.g. article, short story, or a legal document. Then, within the same genre, we followed by examining the metrics' effectiveness in determining the author of the text. Finally, we examined the metrics' effectiveness in determining whether a given text has been written by a human, or a machine.

All metrics performed well **in providing distinctive features for the given texts**, and complemented each other to a high degree in the selected tasks. In terms of accuracy, the accomplished results fare well against the performance of alternative algorithms implemented for the same tasks and the same datasets. Furthermore, if we compare the methods providing said alternative algorithms, we can see that the metrics which underpin our data for the machine learning pipelines are much more lightweight and computationally efficient, and therefore have potential to be used at a large scale.

The fact that our metrics can uniquely identify authors with a relatively high accuracy given the trivial computational load of the lightweight metrics, suggests that improvements and additions to metrics evaluating the text in terms of its structure, such as the ones we proposed in Section 5.4.3, can be used to further improve the accuracy of the metrics and yield more accurate results in the difficult task of authorship identification. **We conclude that creativity boils down to the expression of an author's unique character through their work. If our algorithms were able to capture that, then we can make the claim that our metrics are a good representation of creativity.**

7.1.3 Limitations

We were limited in terms of the amount of time we could spend on the project, and the amount of resources we could use. Subsequently, that limited the amount and scale of experiments we could run, and the amount of data we could use. In terms of design, we have gained a valuable experience in working with NLP systems, and, if given the chance, would like to design the system in a more modular way, or such that it completely abstracts away the underlying NLP framework. We are fascinated with what we could do if we had access to a high-performance cluster, and would like to explore the possibilities of running the benchmarks on a larger scale.

These limitations, however, leave room for future work, which we discuss in a following section.

7.2 Contributions

As discussed in the previous section, we have made a number of contributions to the field of computational creativity. We have implemented a system for benchmarking text, and a set of benchmarks for evaluating the creativity of text. We have also provided a set of experiments that can be used to evaluate the system. These are all non-negligible, as most of the work in the niche of computational evaluation of text has been done in other topics, such as sentiment analysis, or text classification – both of which have major importance in business applications, of course. However, the field of computational creativity is still in its infancy, and we hope that our work can be used as a starting point for future research in the field, and potential applications in the business sphere.

7.3 Future Work

We have identified a number of areas for future work. Firstly, improvement of the existing system is a priority. We would like to improve the system in terms of its design, and make it more modular, and more extensible. We would also like to improve the system in terms of its performance and accuracy. We would like to explore the possibility of running the benchmarks on a larger scale, and on a high-performance cluster. We would also like to explore the possibility of running the benchmarks on a larger dataset, and on a more diverse dataset.

Finally, we would like to explore the possibility of using the benchmarks for evaluating the creativity of text generated by LLMs, in light of recent advances in other "creative" AI, such as text-to-image generation and synthesis AI like Midjourney, Stable Diffusion (Rombach et al., 2022), and DALL-E (Ramesh et al., 2021).

Appendix A

User Manual

The following user manual lists and explains the features of the Mad Hatter package. It also provides a guide on how to install and use the package, as well as how to use the command-line interface. The guide assumes that the user **has installed Python 3.7+**, and has access to a terminal.

A.1 Installation

Run the following command to install the package and its dependencies:

```
pip install madhatter
```

We highly recommend also running NLTK's downloader module in order to have access to all of the features that Mad Hatter provides. To do so, simply run the following command:

```
python -m nltk.downloader all
```

A.1.1 Usage

The package provides high-level abstractions for text analysis that can be used with any text. The following example shows how to use the package to analyse a simple text file within Python:

```
from madhatter.benchmark import CreativityBenchmark

text = "The␣quick␣brown␣fox␣jumped␣over␣the␣lazy␣dog."
bench = CreativityBenchmark(text)

bench.report()
>>> BookReport(title='unknown', nwords=10, mean_wl=3.7, mean_sl
    =45.0, mean_tokenspersent=10.0, prop_contentwords=0.1,
    mean_conc=4.0633333333333335, mean_img=5.359999999999999,
    mean_freq=-1.6792249660842167, prop_pos={'ADJ': 0.2, 'NOUN':
    0.3, 'VERB': 0.1}, surprisal=None, predictability=None)
```

A.1.2 Command Line Interface

Mad Hatter is also available as a CLI tool. Simply provide a path to a text file to the CLI, and it will generate a report for that text. Table A.1 lists the available options to the CLI. The CLI must be supplied a filename or a path to the file to be analysed. The following example shows how to use the CLI to generate a report for an arbitrary text file:

```
> python -m madhatter text.txt -p -m 100 -c 20 -t "My␣Text"

BookReport(title='My␣Text', nwords=100, mean_wl=3.7, mean_sl
    =45.0, mean_tokenspersent=10.0, prop_contentwords=0.1,
    mean_conc=4.0633333333333335, mean_img=5.359999999999999,
    mean_freq=-1.6792249660842167, prop_pos={'ADJ': 0.2, 'NOUN':
    0.3, 'VERB': 0.1}, surprisal=None, predictability=None)
```

Short	Long Form	Default	Description
`-h`	`--help`		Show a help message and exit the program.
`-p`	`--postag`		Whether to return a POS tag distribution over the whole text. The option is a flag, so it only needs to be added.
`-u`	`--usellm`		Whether to run GPU-intensive LLMs for additional characteristics. The option is a flag, so it only needs to be added.
`-m`	`--maxtokens`	-1	Maximum number of predicted words for the heavyweight metrics. Count starts from the beginning of text, -1 to read until the end.
`-c`	`--context`	10	Context length for sliding window predictions as part of heavyweight metrics. Longer context for better results, but may potentially result longer compute times.
`-t`	`--title`		Optional title to use for the report project. If not supplied, the name of the file supplied is used in the book report.
`-d`	`--tagset`	universal	Tagset to use. Default is the universal tagset.

Table A.1: Available options to the CLI.

A.1.3 Advanced Usage

Users are also able to use the package's lower-level functions to create their own custom analysis pipeline or to integrate with other NLP packages such as SpaCy.

```
from madhatter import metrics
from madhatter import benchmark

text = "The␣quick␣brown␣fox␣jumped␣over␣the␣lazy␣dog."
bench = benchmark.CreativityBenchmark(text)

bench.words
>>> ['The', 'quick', 'brown', 'fox', 'jumped', 'over', 'the'
    , 'lazy', 'dog', '.']

metrics.imageability(bench.words)
>>> [1.41, 2.45, 3.14, 4.2, 3.4, 3.65, 1.41, 2.42, 4.1, 0.0]
```

Appendix B

Maintenance Manual

This should be used to describe the details of your implementation. It should be usable by people wanting to install the program, modify the program, extend the program, or trace bugs in its execution. This is an important part of the documentation, and you should ensure that you include details such as: • instructions on how to install the system • instructions on how to compile/build the system • hardware/software dependencies, including libraries and other packages 8 • Organisation of system files, including directory structures, location of files within directories, details of any temporary files • space and memory requirements • list of source code files, with a summary of their role • crucial constants, and their location in the code • the main classes, procedures, methods or data structures • file pathnames, particularly for accessing files of data values • directions for future improvements • bug reports Again, the Maintenance Manual must be included as an appendix to the main report and will therefore be marked as part of the disserta- tion. (this is in addition to a copy of the maintenance manual with the code tar file) Here is an example template project report document, and here is a tarfile of the corresponding LaTeX source (on Unix, just type "make").

Bibliography

Bird, S., Klein, E., and Loper, E. (2009). *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, Beijing.

BNC Consortium (2007). British national corpus, XML edition. Literary and Linguistic Data Service.

Brants, T. (2000). Tnt: A statistical part-of-speech tagger. In *Proceedings of the Sixth Conference on Applied Natural Language Processing*, ANLC '00, page 224–231, USA. Association for Computational Linguistics.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., 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. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). Language Models are Few-Shot Learners. arXiv:2005.14165 [cs].

Brysbaert, M., Warriner, A. B., and Kuperman, V. (2014). Concreteness ratings for 40 thousand generally known english word lemmas. *Behavior research methods*, 46:904–911.

Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y. T., Li, Y., Lundberg, S., Nori, H., Palangi, H., Ribeiro, M. T., and Zhang, Y. (2023). Sparks of artificial general intelligence: Early experiments with GPT-4.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. (2021). Evaluating large language models trained on code.

de Groot, A. M. (1989). Representational aspects of word imageability and word frequency as assessed through word association. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 15(5):824.

Derczynski, L., Bontcheva, K., and Roberts, I. (2016). Broad Twitter corpus: A diverse named entity recognition resource. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, pages 1169–1179, Osaka, Japan. The COLING 2016 Organizing Committee.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Fan, A., Lewis, M., and Dauphin, Y. (2018). Hierarchical Neural Story Generation. arXiv:1805.04833 [cs].

Fellbaum, C. (1998). *WordNet: An Electronic Lexical Database*. Bradford Books.

Franceschelli, G. and Musolesi, M. (2022). DeepCreativity: Measuring Creativity with Deep Learning Techniques. arXiv:2201.06118 [cs].

Francis, W. N. and Kucera, H. (1979). Brown corpus manual. *Letters to the Editor*, 5(2):7.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. (2021). The pile: An 800gb dataset of diverse text for language modeling. *CoRR*, abs/2101.00027.

Gerlach, M. and Font-Clos, F. (2018). A standardized Project Gutenberg corpus for statistical analysis of natural language and quantitative linguistics. *CoRR*, abs/1812.08092.

Goldberg, Y., Adler, M., and Elhadad, M. (2008). Em can find pretty good hmm pos-taggers (when given a good start). In *Proceedings of ACL-08: HLT*, pages 746–754.

Hill, F., Reichart, R., and Korhonen, A. (2015). SimLex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695.

Honnibal, M. and Montani, I. (2017). spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear.

Jurafsky, D. and Martin, J. H. (2009). *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Leech, G., Rayson, P., and Wilson, A. (2014). Word frequencies in written and spoken English.

Loria, S. (2018). textblob documentation. *Release 0.15*, 2.

Manning, C. D. (2011). Part-of-speech tagging from 97% to 100%: is it time for some linguistics? In *Computational Linguistics and Intelligent Text Processing: 12th International Conference, CICLing 2011, Tokyo, Japan, February 20-26, 2011. Proceedings, Part I 12*, pages 171–189. Springer.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781 [cs].

Mirowski, P., Mathewson, K. W., Pittman, J., and Evans, R. (2022). Co-Writing Screenplays and Theatre Scripts with Language Models: An Evaluation by Industry Professionals. arXiv:2209.14958 [cs].

Nigam, K., Lafferty, J., and McCallum, A. (1999). Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, volume 1, pages 61–67. Stockholom, Sweden.

Osgood, C. E., Suci, G. J., and Tannenbaum, P. H. (1957). *The measurement of meaning*. University of Illinois press.

Oxford English Dictionary Editorial (2023). Updates to the OED, March 2023. Blog.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information*

Processing Systems 32, pages 8024–8035. Curran Associates, Inc.

Pierrehumbert, J. B. (2012). Burstiness of Verbs and Derived Nouns. In Santos, D., Lindén, K., and Ng'ang'a, W., editors, *Shall We Play the Festschrift Game? Essays on the Occasion of Lauri Carlson's 60th Birthday*, pages 99–115. Springer Berlin Heidelberg, Berlin, Heidelberg.

Powers, D. M. (1998). Applications and explanations of zipf's law. In *New methods in language processing and computational natural language learning*.

Qian, C., He, T., and Zhang, R. (2019). Deep Learning based Authorship Identification.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.

Ramesh, A., Pavlov, M., Goh, G., Gray, S., Voss, C., Radford, A., Chen, M., and Sutskever, I. (2021). Zero-shot text-to-image generation.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. (2022). High-resolution image synthesis with latent diffusion models.

Schütze, H. and Singer, Y. (1994). Part-of-speech tagging using a variable memory markov model. In *32nd Annual Meeting of the Association for Computational Linguistics*, pages 181–187.

Thede, S. M. and Harper, M. (1999). A second-order hidden markov model for part-of-speech tagging. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics*, pages 175–182.

Torralba, A. and Efros, A. A. (2011). Unbiased look at dataset bias. In *CVPR 2011*, pages 1521–1528, Colorado Springs, CO, USA. IEEE.

van Rossum, G., Warsaw, B., and Coghlan, N. (2001). Style guide for Python code. PEP 8, Python Software Foundation.

Wikipedia contributors (2023). Wordnet — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=WordNet&oldid=1143619785`. [Online; accessed 14-March-2023].