Final report Summer Internship

Tilian bourachot

July 2024

Contents

Chapter 1

Introduction

During my internship at CREST, from June 10 to August 6, under the supervision of Ms. Khaleghi, I had the opportunity to explore several topics related to Large Language Models (LLMs). The primary focus of the internship was the detection of topic changes in texts using a library developed by my supervisor, which allows for the detection of change points in time series. As I was relatively unfamiliar with this subject at the outset, my initial tasks involved extensive research and reading of academic papers about state-of-the-art models, neural networks, and transformers.

Following this research phase, we conducted our first experiments to compare two recent LLM models, Phi3 and LLama3. We then delved into feature engineering and word embeddings, which led us to investigate the Word2Vec model. Numerous experiments were conducted on this model, particularly focusing on clustering and dimensionality reduction. Additionally, we adapted distributional distances and clustering algorithms from various research articles. In the final weeks of the internship, we explored the potential connections between LLMs and information theory, using compression techniques like Gzip.

I would like to express my sincere gratitude to Ms. Khaleghi for giving me the opportunity to undertake this internship and for enriching my knowledge of LLMs and current machine learning models. Her unwavering support and guidance throughout the three months were invaluable, as she was always available to answer my questions and help me progress.

All the work carried out during this internship, as well as the content of this report, can be found at the following GitHub links: GitHub Link. For any inquiries, feel free to contact me at tilian.bourachot@gmail.com.

Chapter 2

Quick introduction about language models and Llms

In recent years, the field of natural language processing (NLP) has witnessed significant advancements, primarily driven by the development and application of language models (LMs). Language models, such as GPT (Generative Pre-trained Transformer) and BERT (Bidirectional Encoder Representations from Transformers), are designed to predict the next word in a sequence based on the preceding words. These probabilistic models are capable of generating text that adheres to specific linguistic styles or patterns, making them invaluable for various applications, including text completion, translation, and summarization.

Building upon the foundation laid by traditional language models, Large Language Models (LLMs) have emerged, leveraging advanced deep learning techniques to enhance their capabilities in understanding and generating natural language. LLMs aim to produce coherent and contextually appropriate text by analyzing extensive datasets during their training. The core of LLMs' functionality lies in transformer architectures, which allow these models to handle complex language tasks with remarkable accuracy and fluency.

The versatility and power of LLMs have opened up numerous possibilities across different domains. They are utilized in conversational agents, content creation, sentiment analysis, and many other areas where the generation of human-like text is essential.

Chapter 3

Goals of the Internship

Our goal in this study is, first and foremost, to discover how language models function, what their architecture is, and what the state-of-the-art models are. Next, we want to conduct experiments on new models such as Phi3 and Llama3, as well as LLMs like GPT and BERT, to try to understand how they represent words in embeddings and how those models interpret this. Finally, our main goal is to use PyChest, introduced by A. Khaleghi as a Python package that provides tools for the simultaneous estimation of multiple changepoints in the distribution of piecewise stationary time series. We would like to adapt this algorithm to feature engineering and word embeddings to detect when the main subject of a text is changing. To achieve this goal, we first need to find a way for word embedding, choose between dynamic and static word embedding according to our issue. Subsequently, the algorithm of PyChest is implemented for low-dimension vectors, so we would have to find a way to reduce the number of dimensions of our word embeddings.

Chapter 4

Feature engineering and Words Embedding

Feauture engineering : [1] Feature extraction is vital in ANLP tasks, allowing for the effective representation and analysis of textual data. It captures linguistic properties, semantic information, and structural patterns, enhancing performance in sentiment analysis, machine translation, text summarization, and more. Extracting meaningful features enables deeper linguistic analysis, improving accuracy and efficiency in ANLP applications.

Feature extraction involves selecting and transforming raw data into a set of relevant features that effectively represent and describe the data. Word embedding, as a semantic approach to feature extraction, captures the underlying meaning and semantic relationships between words, representing textual data in a dense vector space. This representation enhances the understanding of semantic similarities and contextual associations, facilitating more effective text analysis and interpretation.

There is a wide range of state-of-the-art approaches for feature extraction, specifically word embedding. This subsection provides an overview of the most commonly used methods in the Arabic language, categorized as either static or contextualized.

The article [1] indicates that the embedding step significantly impacts model performance in ANLP. Additionally, the Arabic pre-trained BERT achieves the best performance when used to generate embeddings.

4.1 Dynamic Representation

4.1.1 Transformer model

The Transformer model represents a significant advancement in natural language processing and understanding. Introduced in [8] as a neural network architecture, it revolutionizes how machines comprehend the context of words

within sentences. Unlike traditional Recurrent Neural Networks (RNNs), which process words sequentially, Transformers can handle words in parallel, thanks to their attention mechanisms. This ability allows Transformers to understand word context more effectively and efficiently.

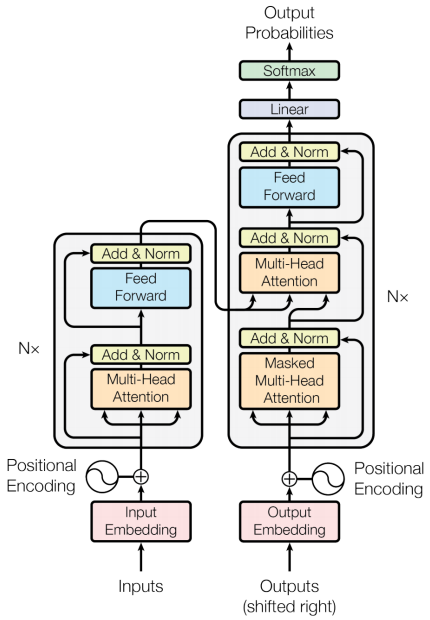


Figure 1: The Transformer - model architecture.

- Attention Mechanism

  A key component of the Transformer model is the self-attention mechanism. This mechanism weighs the relevance of different words in the input when making predictions. For each token in the sequence, the self-attention process calculates a weighted sum of all tokens, with the weights determined by the attention mechanism. This approach allows the model to capture the dependencies between words irrespective of their position in the sequence, enhancing its contextual understanding. It allows the vectors to talk to each other and to pass information to uptade their weights according to the sentence. For example : a machin Learning model and a fashion model, model has not the same meaning in both sentences. A current state-of-the-art mechanism is FlashAttention, which is used in small language models like Phi3 to achieve almost the same performance as LLMs with fewer resources. See the appendix to explore FlashAttention [2].

- Advantages Over Previous Architectures

Transformers offer several advantages over previous neural network architectures like RNNs and their variants (LSTM, GRU). Firstly, they are highly suitable for parallel computing because they do not require sequential processing of tokens. This parallelism significantly speeds up the computation, making Transformers more efficient in handling large datasets and complex tasks.

- Pre-trained Models

  Pre-trained Transformer models like BERT and GPT are trained on vast amounts of text data. These models can be fine-tuned for specific tasks, such as sentiment analysis, language translation, and text summarization. The pre-training involves exposing the models to large corpora of text, enabling them to learn grammar, facts about the world, reasoning abilities, and even biases present in the data.

- ChatGPT and GPT Models

  The core of ChatGPT is a Transformer. Specifically, ChatGPT is based on the GPT (Generative Pre-trained Transformer) architecture. GPT models are designed to predict the next word in a sequence, processing text from left to right. This unidirectional approach makes them particularly well-suited for text generation tasks. GPT-3, for example, consists of 175 billion parameters, making it one of the most powerful language models available.

- Self-Supervised Learning

  Transformer models like GPT and BERT are trained using self-supervised learning. For GPT, this means predicting the next word in a sentence, while BERT predicts a word based on the surrounding context. This training methodology allows the models to learn rich, contextual embeddings of the input text, known as dynamic embeddings, which can be fine-tuned for a variety of downstream applications.

4.1.2 Precision about how BERT and GPT are working

The transformer architecture is divided in 3 main part : Encoder-Decoder, Encoder only and Decoder only.

Encoder-Decoder can understand and generate sentences, they are used for Translation and Text-summurization.

The encoder is responsible for understanding and extracting relevant information from the input text. It then produces a continuous representation (embedding) of the input text, which is passed to the decoder. Encoders are useful for tasks such as classification, sequence tagging, and sentiment analysis. For example, BERT (Bidirectional Encoder Representations from Transformers) is a well-known encoder model.

BERT uses masked language modeling and next sentence prediction as pre-training objectives. In masked language modeling, random word tokens in the

input sequence are masked, and the model is trained to predict these masked tokens based on the surrounding context. This approach allows BERT to learn rich contextual representations of the input texts. Additionally, BERT's next sentence prediction objective helps the model understand the relationship between consecutive sentences.

Typically, models like BERT need to be fine-tuned for various downstream tasks, such as sentiment analysis or other specific tasks. However, it's important to note that while BERT can understand text, it cannot generate text. Its primary function is to comprehend and analyze textual information rather than create new text.

Finally, the Decoder model generates output word by word, and the predictions for position i can depend only on the known outputs at positions less than i. LLms like GPT2,3,4, ChatGPT are decoder and also state of art model such as Phi3 or Llama3 are too. Also, Decoder are controlled by prompting (instruction).

4.1.3 Conclusion

We have explored the state-of-the-art mechanism of transformer models, which can generate coherent text by utilizing the context and meaning of other words in a sentence. Each token can be represented by different vectors based on its meaning, context, and position within the sentence. Conversely, these models require extensive training data. Some advanced models, such as Phi3, attempt to reduce the required amount of data by using high-quality datasets, as mentioned in the appendix. Additionally, these models contain a large number of parameters (approximately 3 billion for Phi3 and 175 billion for GPT-3), which necessitate significant storage. The details on how models determine the number of parameters they retain are described in the appendix. In a next section, we will discuss ways to reduce this number. Lastly, it is important to consider static embeddings, as they use fewer parameters and will be more useful in our future experiments on change-point detection.

4.2 Static Reprensation

4.2.1 Introduction about Word2Vec

According to the article "word2vec Parameter Learning Explained" by Xin Rong [7], published in 2016, which was before the discovery of the Transformer architecture, Word2Vec is only an algorithm based on neural network learning. It enables the generation of a matrix that contains, for each word in the vocabulary, its vector representation based on its definition, extracted from context in the studied corpus / training dataset. It then captures similarities with other words etc.

Then, Word2Vec and GloVe produce **static embeddings** (the same vector for a word regardless of its position or context), in contrast to Transformer

models, which generate embeddings that vary depending on the context in which the words appear.

4.2.2 Applications of Word2Vec

- **Modeling Word Similarity**

  Word2Vec captures the semantic similarity between words by representing them as vectors in a continuous vector space where similar words are positioned close to each other. This enables various applications that require understanding and leveraging word relationships.

  Applications:

  - *Synonym Search:* Identifying synonyms by finding close vectors in the space. For example, the words "happy" and "joyful" will have similar vector representations.
  - *Search and Recommendation:* Enhancing the relevance of results by considering semantic similarity. Search engines and recommendation systems can suggest items that are contextually similar to what the user is interested in.

- **Text Analysis and NLP** Word2Vec provides dense word representations that can be used in various text analysis and NLP tasks.

  Applications:

  - *Text Classification:* Using word vectors as input features for text classification models, which categorize texts into predefined categories, such as spam detection or topic classification.
  - *Sentiment Analysis:* Representing words and phrases as vectors to determine the sentiment expressed in texts, such as identifying positive, negative, or neutral sentiments in reviews.

- **Machine Translation and Multilingual Processing** Word2Vec assists in aligning words from different languages based on their semantic meanings.

  Applications:

  - *Semantic Alignment:* Aligning words from different languages to similar concepts, aiding in the creation of multilingual word embeddings.
  - *Cross-lingual Tasks:* Enhancing the performance of tasks involving multiple languages, such as translating documents or extracting information from multilingual sources.

- **Anomaly and Error Detection** Word2Vec can detect anomalies and linguistic errors by analyzing words in their context.

  Applications:

- *Spelling and Grammar Correction:* Detecting and correcting linguistic errors by understanding the context in which words appear. For example, suggesting corrections for words that do not fit well in their context.
- *Outlier Detection:* Identifying words or phrases that are unusual or out of place in a given context, useful for spotting errors or anomalies in text data.

- **Clustering and Data Visualization** Word vectors can be clustered to uncover themes or topics within a text corpus, and visualized in lower-dimensional spaces.

  Applications:

  - *Word Clustering:* Grouping words to discover themes or topics within a corpus, revealing underlying patterns and relationships in the data.
  - *Dimensionality Reduction:* Visualizing word vectors using techniques like t-SNE or PCA, making it easier to interpret the relationships between words.

4.2.3 Word2Vec and Text Generation

Word2Vec is not directly used to generate text in the same way that models like GPT (Generative Pre-trained Transformer) do. However, it plays a foundational role in natural language processing (NLP) by providing meaningful word embeddings that can be utilized in various NLP tasks, including text generation.

- **Differences between Word2Vec and GPT**
  - **Purpose:**
    - *Word2Vec:* Designed to learn vector representations of words, capturing semantic relationships between them.
    - *GPT:* Designed to generate coherent and contextually relevant text by predicting the next word in a sequence.
  - **Architecture:**
    - *Word2Vec:* Utilizes shallow neural networks (such as CBOW or Skip-gram models) to produce word embeddings.
    - *GPT:* Utilizes transformer architecture, consisting of multiple layers of self-attention mechanisms to model long-range dependencies in text.
  - **Output:**
    - *Word2Vec:* Outputs dense vectors representing words.
    - *GPT:* Outputs sequences of words to generate text.

- **How Word2Vec is Utilized in Text Generation**

  While Word2Vec itself does not generate text, it can be a component in more complex systems that do.

  - **Preprocessing and Feature Extraction:**
    - *Embedding Initialization:* Word2Vec embeddings can initialize the embedding layers of more complex models like transformers.
    - *Feature Representation:* Word2Vec can convert text into vector representations that are fed into other NLP models.

4.2.4 Conclusion

We just see that static embeddings, like Word2vec, allow representing words as vectors. However, unlike dynamic embeddings and transformer architectures, static embeddings do not consider the context of the sentence and the position of the word. This means that a word has the same representation in two different sentences, even if the context is not the same. But this method also has advantages: for instance, words are represented with vectors of 300 dimensions with some Word2Vec models, instead of 3000-dimensional vectors for Phi3 or 12000 for GPT-3. Thus, it is easier to process them compared to high-dimensional vectors. Finally, our main goal, which is to detect change points (topics) in a text, will also be a way to consider words out of their context and correctly spot the change in context.

Chapter 5

Size compression for Llms

Large Language Models (LLMs) such as GPT-175B have revolutionized natural language processing by demonstrating exceptional capabilities. However, these advancements come with substantial resource requirements for storage and computation, presenting significant challenges. For instance, GPT-175B demands at least 320GB of storage and multiple A100 GPUs for inference. Additionally, the environmental impact of LLMs is considerable due to their significant energy consumption, which contributes to carbon emissions. Therefore, model compression is crucial for making these technologies more sustainable and accessible.

Model compression involves transforming large, resource-intensive models into more compact versions that retain much of the original performance while being more efficient in terms of storage and computation. This not only alleviates the resource requirements but also helps in reducing the environmental impact of deploying LLMs.

We can distinguish 2 compression modes : Model's parameters reduction and Embeddings' dimension reduction.

5.1 Model's parameters reduction

This paper [9] provides an overview of recent advancements in LLM compression, highlighting the importance of ongoing research to enhance the efficiency and applicability of these models in real-world contexts. Model compression is a promising solution to address the current limitations of LLMs in terms of resource requirements and environmental impact. Through continued innovation and evaluation, we can achieve more sustainable and accessible AI technologies.

5.1.1 Compression Techniques

- Quantization reduces the precision of model weights and activations, thereby decreasing the model size and speeding up computations.

This can be done during training (Quantization-Aware Training) or after training (Post-Training Quantization).

  - **Quantization-Aware Training (QAT)**: Techniques like LLM-QAT, PEQA, and QLORA train the model with quantized weights, allowing the model to adapt to lower precision during the training process.
  - **Post-Training Quantization**: This involves quantizing the model weights and activations after training. Methods like GPTQ, AWQ, and SqueezeLLM are prominent examples, often used due to their simplicity and effectiveness.

- Pruning reduces the number of parameters in a model by removing less important neurons or connections.
  - **Unstructured Pruning**: This method removes individual weights or neurons, leading to a sparse model structure. Techniques like SparseGPT and Wanda focus on optimizing this process to achieve high sparsity with minimal performance degradation.
  - **Structured Pruning**: This approach removes entire structural components such as layers or channels, maintaining the overall architecture but simplifying the model. Methods like GUM and LLM-Pruner have shown promise in reducing model complexity while preserving performance.

- Knowledge distillation involves transferring knowledge from a large "teacher" model to a smaller "student" model. This allows the student model to approximate the performance of the teacher model while being significantly smaller.
  - **White-box Distillation**: Access to the teacher model's parameters allows for deeper insights and better performance improvements, as seen in methods like MINILLM and GKD.
  - **Black-box Distillation**: Only the teacher model's predictions are available, which is often sufficient for many applications. Techniques here leverage emergent abilities like In-Context Learning, Chain-of-Thought, and Instruction Following to distill knowledge effectively.

- Other advanced methods include:
  - **Neural Architecture Search (NAS)**: This automates the design of compressed models, optimizing for both performance and efficiency.
  - **Low-Rank Factorization**: Techniques like TensorGPT decompose the weight matrices into lower-rank approximations, significantly reducing the model size while retaining performance.

5.1.2 Evaluation and Benchmarking

Evaluating the effectiveness of compressed models involves several key metrics and benchmarks:

- Metrics
  - **Number of Parameters**: Indicates the total count of learnable weights or variables in the model, impacting both memory usage and computational requirements.
  - **Model Size**: Refers to the disk space or memory footprint needed to store the model, including all weights and biases.
  - **Compression Ratio**: The ratio between the original size of the uncompressed model and the size of the compressed model, indicating the efficiency of the compression.
  - **Inference Time**: Measures the time taken by the model to process and generate responses for input data, crucial for real-time applications.
  - **Floating Point Operations (FLOPs)**: Counts the number of arithmetic operations the model performs, providing an estimate of its computational requirements.
- Benchmarks and datasets
  - **GLUE and SuperGLUE**: Evaluate performance across a range of natural language understanding tasks.
  - **LAMBADA**: Tests context-dependent understanding.
  - **LAMA and StrategyQA**: Assess reasoning abilities.
  - **SQuAD**: Focuses on machine reading comprehension tasks.
  - **BIG-Bench (BBH)**: Covers over 200 NLP tasks, offering a comprehensive evaluation of model performance across diverse challenges.

5.1.3 Challenges and Future Directions

- Performance-Size Trade-off : One of the primary challenges is balancing the reduction in model size with the preservation of performance. This requires more theoretical and empirical analyses to understand the underlying trade-offs.
- Dynamic Compression : Current methods often rely on manual design and trial-and-error. Integrating automated techniques like NAS could streamline and enhance the compression process.
- Explainability : As compressed models are deployed in critical applications, it is crucial to develop explainable compression methods to ensure transparency and reliability.

- Sustainability : Reducing energy consumption and carbon emissions remains a priority. Continued research in model compression can make AI technologies more sustainable and accessible.

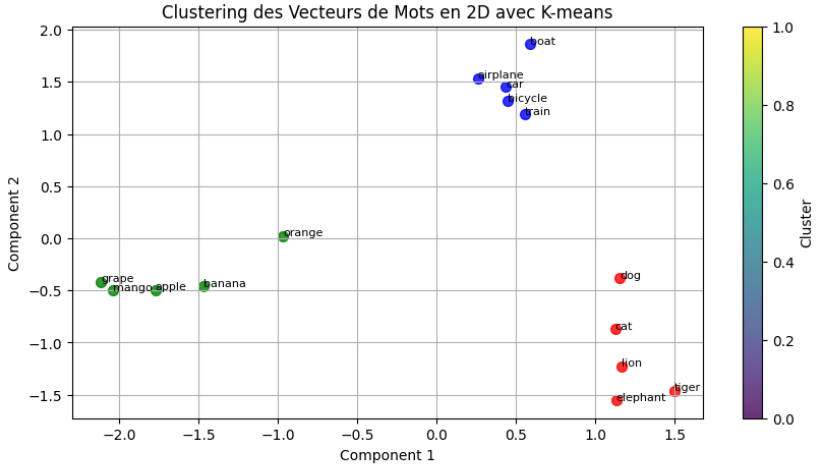
5.2 Embeddings' dimension reduction

In this section, we investigate the dimensions of word embeddings and explore how they can be reduced without significant loss of information. This investigation follows the methodology described in the notebook `word2vec_clustering` available on GitHub. Our analysis focuses on several key aspects.

Word2Vec Model

We began by utilizing the pre-trained Word2Vec model `GoogleNews-vectors-negative300.bin`, available here, which was trained on a Google News corpus containing three billion words. This model provides 300-dimensional word embeddings for three million English words.

Our primary goal was to understand the nuances of how this model works, particularly how its parameters influence the meaning encoded in word vectors. Initially, we selected three categories – fruits, animals, and transportation – to investigate whether these categories are distinguishable in the vector space. We used Principal Component Analysis (PCA) to project these vectors onto a 2D plane, revealing three distinct clusters corresponding to our categories.



However, the high dimensionality of the vectors (300 dimensions) led us to explore methods for identifying similarities between categories without relying on PCA. This led us to our first key question: what distance metric should we use to discern differences between these categories? We compared two classic

metrics – Euclidean distance and cosine similarity – and found that both were effective for our needs.

5.2.1 Experiments on dimension Reduction

Our main goal was to reduce the dimensionality of our word vectors for use in the PyChest algorithm for change point detection. To achieve this, we explored clustering these vectors using the KMeans algorithm. We examined inter-cluster and intra-cluster distances, both computed using Euclidean distance (with similar results for cosine similarity). The inter-cluster distance measures the minimum distance between points in two different clusters, while intra-cluster distance evaluates the distance between points within the same cluster. For each cluster, we plotted boxplots of these distances (minimum, maximum, and average) as a function of the number of reduced dimensions.

Our results revealed that for both distance metrics, the distance functions were concave with respect to the number of dimensions. This result is consistent with the intuition that higher-dimensional vectors contain more information, resulting in larger distances between points in the space.

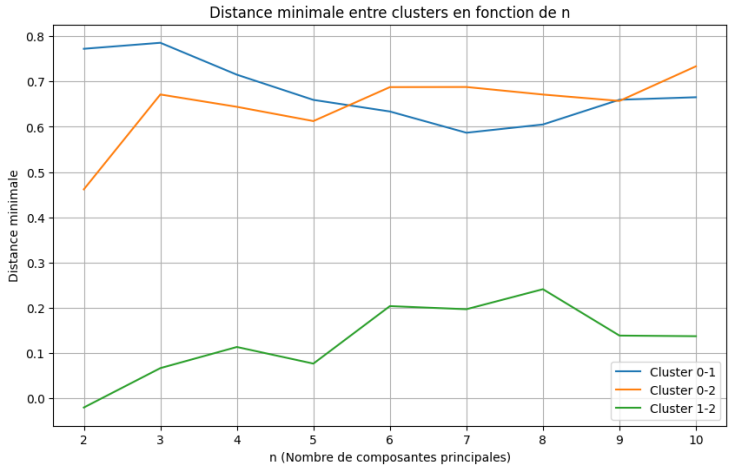


Figure 5.1: Inter-cluster distance as a function of reduced dimensions.

We also examined the ratio between these distances, which also exhibited a convex curve (Figure 4.3).

To verify our results and ensure comparability, we generated multivariate normal random variables for future testing, given that word vectors also appeared to follow a normal distribution.

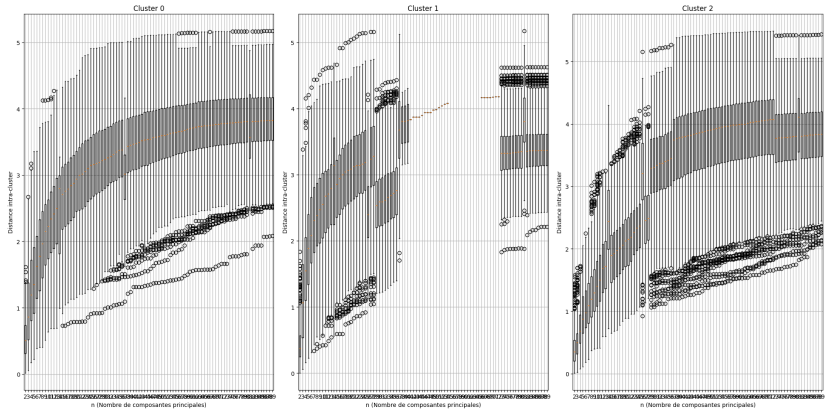


Figure 5.2: Intra-cluster distance as a function of reduced dimensions.

Experimentation with Multivariate Gaussian Distributions

To validate our hypotheses, we generated a new *vocabulary* by creating three different samples from three distinct multivariate Gaussian distributions, each corresponding to one of our word categories. For each distribution, we generated 30 vectors with 300 dimensions, analogous to our word embeddings. Inspired by our previous experiments, we plotted the inter-cluster and intra-cluster distance curves and observed similar patterns, including a convex curve for the inter/intra-cluster ratio, further validating our study.

Thus, these results validate our hypotheses and allow us to focus on the study of samples generated by multivariate Gaussian laws, which are more easily manageable and adaptable.

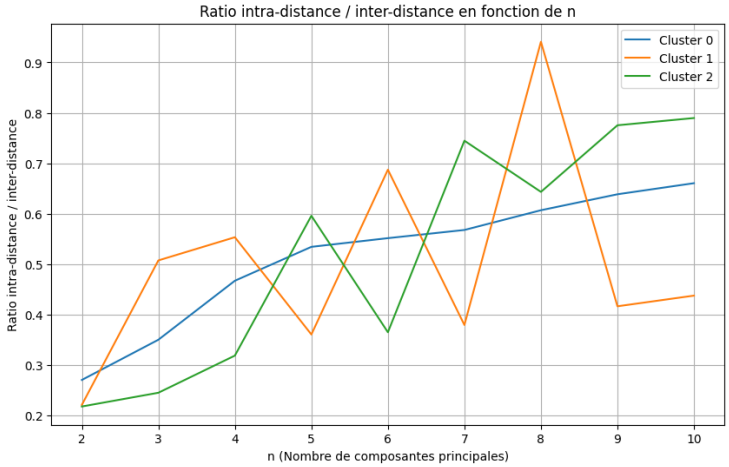


Figure 5.3: Ratio intra-distance / inter-distance as a function of reduced dimensions.

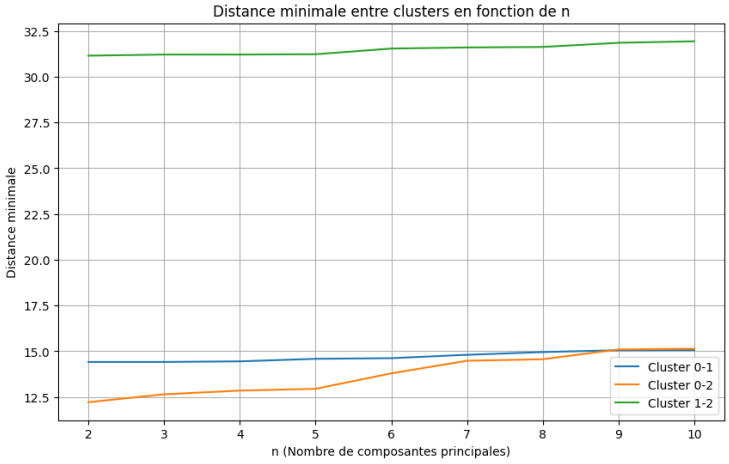


Figure 5.4: Inter-distance with Multivariate Gaussian Distributions.

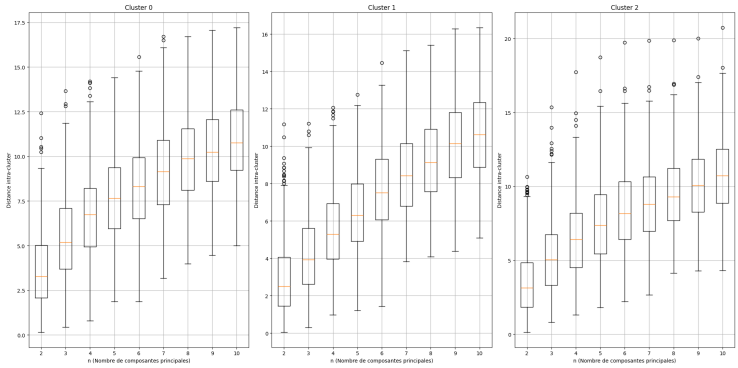


Figure 5.5: Intra-distance with Multivariate Gaussian Distributions

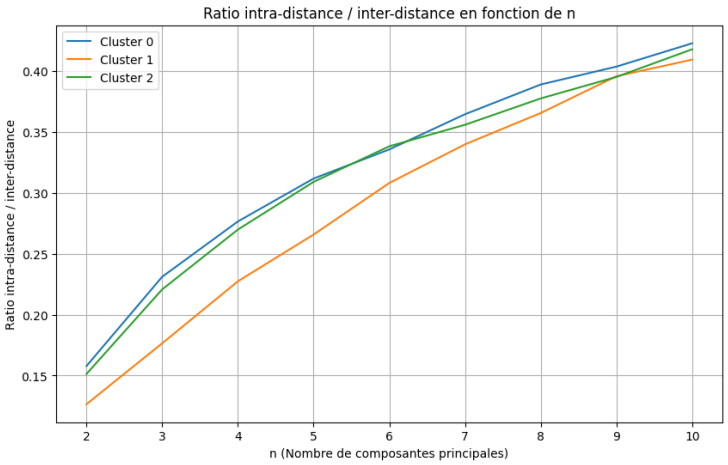


Figure 5.6: Ratio intra-distance / inter-distance as a function of reduced dimensions, with Multivariate Gaussian Distributions.

Chapter 6

Distributional Distances and Clustering

6.1 Introduction

Despite these results, we encountered challenges, particularly regarding the use of Euclidean distance in our inter/intra-cluster distance calculations. Ideally, we wanted to use a distance metric specifically tailored to distributions – a *distributional distance* – to determine if two samples follow the same distribution.

To address this, we consulted an article by Mme Khaleghi titled *Consistent Algorithms for Clustering Time Series* [5], which introduces a distributional distance :

Definition 1 (Distributional Distance) *The distributional distance between a pair of process distributions ρ_1, ρ_2 is defined as follows (Gray, 1988):*

$$d(\rho_1, \rho_2) = \sum_{m=1}^{\infty} w_m \sum_{l=1}^{\infty} w_l \sum_{B \in \mathcal{B}_{m,l}} |\rho_1(B) - \rho_2(B)| \,,$$

where we set $w_j := 1/j(j+1)$, but any summable sequence of positive weights may be used.

We first tried to adapt this distance for our study. As a reminder, all our tests are conducted using samples generated by a multivariate Gaussian distribution to obtain consistent results, and we will then attempt to adapt our results to the vectors of our initial words.

As a preamble, this section is done in collaboration with Théo Monet, who helped me a lot to understand the workings of this theoretical distance and contributed to its improvement.

After several simplifying assumptions (notably the fact that in our case, the variables are iid and continuous), we obtain a new distributional distance:

Definition 2 (New Distributional Distance) *The distributional distance between a pair of process distributions ρ_1, ρ_2 in our study case is defined as follows :*

$$\hat{d}(\rho_1, \rho_2) = \int_{\mathbb{R}^d} |\rho_1(B) - \rho_2(B)| \, dB,$$

First, to implement this distance, we need to determine the density of the observed sample, because we only have the observation and we don't have the exact value of ρ_1, ρ_2 . To do so, we use the Kernel Density Estimation (KDE) algorithm, which allows us to estimate the density of our observations using kernel functions.

6.2 Kernel Density Estimation (KDE)

Wikipedia Reference: `https://en.wikipedia.org/wiki/Kernel_density_estimation`

Multivariate KDE Formula:

The multivariate kernel density estimator is defined as follows:

$$\hat{f}_h(\mathbf{x}) = \frac{1}{n \prod_{i=1}^{d} h_i} \sum_{j=1}^{n} K\left(\frac{x_1 - X_{j1}}{h_1}, \ldots, \frac{x_d - X_{jd}}{h_d}\right)$$

where $\mathbf{x}$ is the evaluation point, $X_j = (X_{j1}, \ldots, X_{jd})$ are the sample points, $h_1, \ldots, h_d$ are the bandwidths, and K is the multivariate kernel function. We use the standard multivariate normal kernel. //

We then implement the Kernel Density Estimation method as well as this distance in Python, but we encounter a new obstacle: we want to integrate over $\mathbb{R}^d$. However, for $d = 3$, the program is extremely slow, and ideally, we would like to integrate over $\mathbb{R}^d$ with $d = 300$ in the future. Therefore, we need to find another approach to overcome this problem.

A first solution is to discretize the integral, which works quite well and gives us consistent results, with a distance close to 0 for similar distributions.

6.3 MMD and Offline clustering

Ultimately, we turned to another distributional distance, the *Maximum Mean Discrepancy* (MMD), described here. Maximum mean discrepancy (MMD) is a kernel based statistical test used to determine whether given two distribution are the same. MMD effectively measures the similarity between two samples, and as expected, the distances tended towards zero when the samples were similar.

The code for the following section is available in the notebook `MMD_Offline_clustering(1).ipynb`. Now that we have our new distributional distance and functional approach, we return to our clustering problem.

To achieve this, we implement the offline clustering algorithm presented in Mme Khaleghi's article [5].

Algorithm 1 Offline clustering

INPUT: sequences $S := \{\mathbf{x}_1, \cdots, \mathbf{x}_N\}$**, Number** κ **of clusters**
Initialize κ***-farthest points as cluster-centres:***
$c_1 \leftarrow 1$
$C_1 \leftarrow \{c_1\}$
for $k = 2..\kappa$ **do**
    $c_k \leftarrow \underset{i=1..N}{\operatorname{argmax}} \min_{j=1..k-1} \hat{d}(\mathbf{x}_i, \mathbf{x}_{c_j})$***, where ties are broken arbitrarily***
    $C_k \leftarrow \{c_k\}$
end for
Assign the remaining points to closest centres:
for $i = 1..N$ **do**
    $k \leftarrow \operatorname{argmin}_{j \in \bigcup_{k=1}^{\kappa} C_k} \hat{d}(\mathbf{x}_i, \mathbf{x}_j)$
    $C_k \leftarrow C_k \cup \{i\}$
end for
OUTPUT: clusters $C_1, C_2, \cdots, C_\kappa$

Figure 6.1: Offline clustering algorithm

Reminder on Object Identification

At this point in our study, it is crucial to revisit the objects to which we want to apply our algorithm. Improper identification of these objects caused us some issues, particularly when we attempted to apply the clustering algorithm.

We aim to apply our clustering algorithm to sequences of words, i.e., sequences of vectors rather than the vectors themselves. For example, we denote $x_1 = [v_1, v_2, \ldots, v_n]$ as a sequence of vectors of size n. Our goal is to perform clustering on sequences such as $x_1, x_2, \ldots, x_9$, where x_1, x_2, x_3 follow one distribution, x_4, x_5, x_6 follow another, and x_7, x_8, x_9 follow yet another. In the Word2Vec model, each word vector has a size of 300.

Initial Testing

After correcting the algorithm to handle sequences of vectors, we conduct an initial test focusing on Bernoulli distributions. We fix the dimension to 1 instead of 300. We generate 10 sequences of length $n = 100$, with the first 3 sequences being iid Bernoulli($p = 0.3$) and the remaining sequences iid Bernoulli($p = 0.6$). We compute the classification error and repeat this for $n = 500, 1000, 1500, 2000, 2500, 3000$, generating a graph of errors. We only have 2 clusters. The graph of the error rate as a function of n is shown below:

The results are quite satisfactory and consistent. More n is, less the error rate is. Indeed, as observed with the MMD distance, as the sample size (n) increases, the MMD distance becomes more precise, approaching zero for samples from the same distributions.

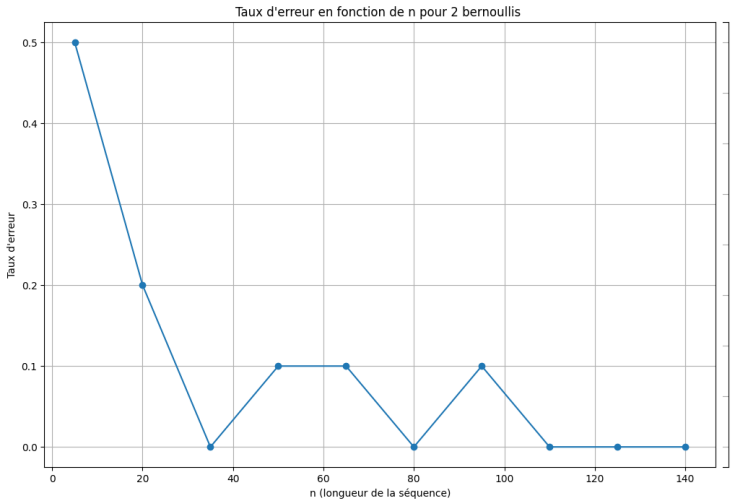


Figure 6.2: Taux d'erreur en fonction de n pour 2 Bernoullis.

Challenges with Multiple Clusters

However, the results are significantly less accurate when increasing the number of clusters to 3...

Gaussian Distributions Clustering

We then decided to move to clustering with 2 Gaussian distributions. For reference, in the case of our word vectors, which are independent, we only vary the mean vector to distinguish the distributions since the covariance matrix is always diagonal. We observed that clustering performs well as the sample size increases, and similarly when moving to 3 clusters.

6.4 Application to Word2Vec Embeddings

Now that our distributional distance and clustering algorithms work well for Gaussian distributions, we want to return to our original topic and apply these methods to Word2Vec embeddings.

This section follows the notebook `MMD_xord2vec.ipynb`.

We now aim to adapt our clustering algorithm to words. To achieve this, we use the Word2Vec model to access the vectors representing different words.

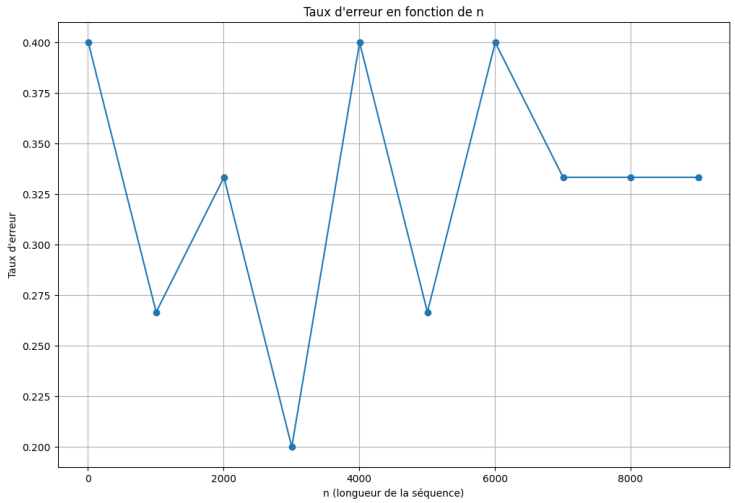


Figure 6.3: Taux d'erreur en fonction de n pour 3 Bernoullis.

Objective: Generating Good Test Sequences

To ensure effective clustering, we need long sequences of words. We start by choosing 3 distinct categories of words to serve as our 3 clusters: biology, food, and sports. For each category, we generate 3 sequences of 50 different words related to the category, with the help of ChatGPT. The first challenge is that some generated words are not recognized by the Word2Vec model, especially compound words. We need to replace these with other words that the model recognizes.

Applying the Clustering Algorithm

Now that our test set is created, we apply our clustering algorithm and keep in mind that one of our main goals is to observe how reducing the dimensionality of the vectors affects the clustering results. We plot the clustering error rate as a function of the number of principal components obtained via PCA.

The graph is quite satisfactory; indeed, we observe that as the vectors contain more information (higher number of components), the clustering becomes more effective.

Challenges with Increasing Sequences

We then attempted to add 2 more sequences per category to reach a total of 5 sequences. However, the clustering performance significantly degraded. I believe this issue arises because, as we generate more different words per category,

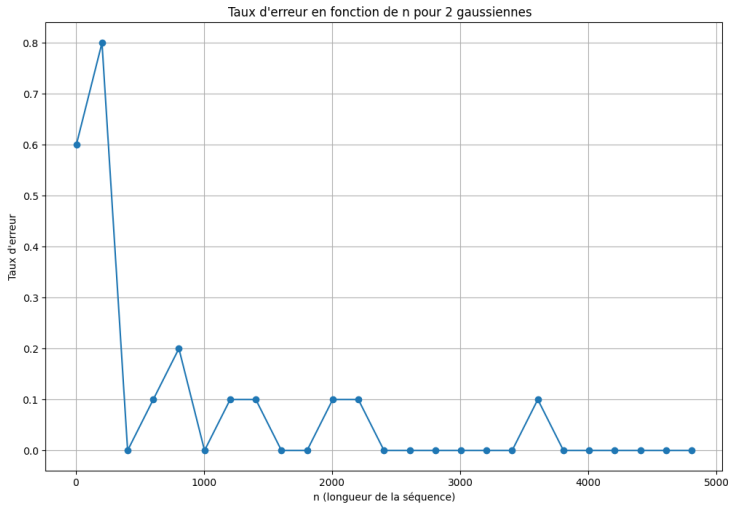


Figure 6.4: Taux d'erreur en fonction de n pour 2 Gaussiennes.

approaching around 200 words, some words begin to have connections or similarities with words from other categories, leading to potential clustering errors.

In conclusion, our results are quite interesting, but we are not entirely satisfied. Therefore, as the internship comes to an end, we have decided to shift our focus to a related topic in the realm of LLMs and NLP: information theory.

Information Theory and NLP.

Then, we decided to focus on something new for the end of the internship. We decided to completely change our approach. Instead of using Word2Vec and PCA to reduce the dimensionality of vectors for applying PyChest, we employed the Gzip library, which allows for folder compression.

How Gzip Works

Gzip uses a technique called LZ77 (Lempel-Ziv 77), a lossless compression method, along with Huffman coding for additional compression.

LZ77 Compression

LZ77's primary idea is to identify repeated substrings in the data and replace them with a reference to a previous occurrence of the substring.

For example, for the string: ABABABAB, the pattern "AB" repeats several times. The LZ77 algorithm identifies these repetitions. Instead of storing each

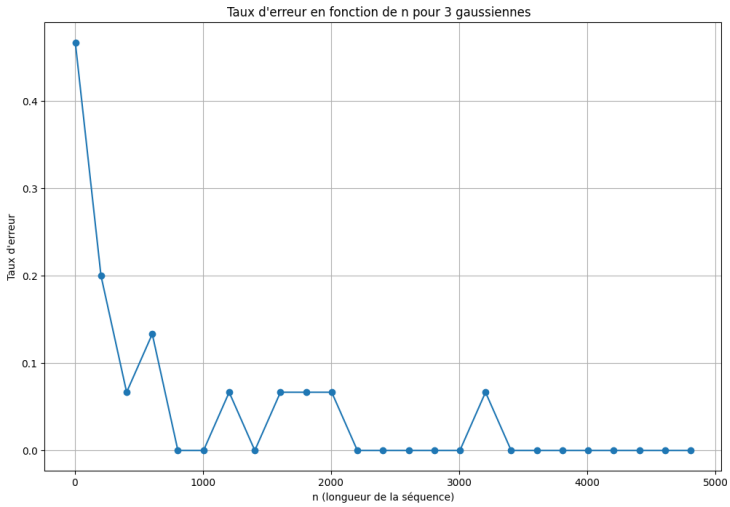


Figure 6.5: Taux d'erreur en fonction de n pour 3 Gaussiennes.

repetition of "AB", the algorithm stores "AB" once and adds a reference each time it encounters a repetition.

Thus, "ABABABAB" might be compressed to something like: [0,0,A], [0,0,B], [2,2], [2,2]. Here, [0,0,A] means "A" is the first character, [0,0,B] means "B" is the second character, and [2,2] means "take the last two characters and repeat them twice."

This approach reduces the amount of data by replacing repetitions with short references to previously found patterns.

Huffman Coding

Gzip applies Huffman coding for additional compression. It compresses data by using shorter codes for frequent characters or patterns and longer codes for less frequent ones.

For instance, if "A" appears frequently and "Z" appears rarely, Huffman coding might assign:

- "A" $\rightarrow$ 1
- "Z" $\rightarrow$ 1110

In the compressed file, "A" would be replaced by "1" (one bit), while "Z" would be replaced by "1110" (four bits). Since "A" is much more frequent, this reduces the overall file size, especially for spaces.

Decompression Process

During decompression, Gzip performs the inverse operation:

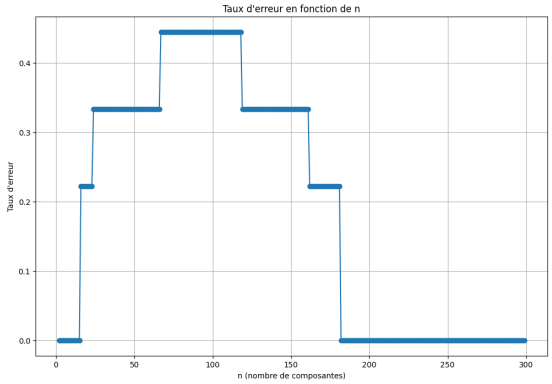


Figure 6.6: Taux d'erreur du clustering pour les mots en fonctions de la dimension des mots.

- Reads the references and reconstructs the repetitions.
- Applies Huffman coding to reconstruct the original data.
- Combines everything to restore the file to its original state.

Experimental Application

The Gzip library exists in Python and facilitates compression. However, since Gzip operates primarily on repeated sequences of characters, it is more effective on sentences than on individual words (see screenshot). Hence, applying it to single words is inefficient.

Additionally, Gzip can be used for text classification, as discussed in the paper "Low-Resource Text Classification: A Parameter-Free Classification Method with Compressors". In this study, Gzip compression technique is combined with the kNN algorithm for classification. They compare its success rate with other neural network models such as BERT across various datasets (AG News, DBpedia, etc.). Overall, Gzip performs better than other models.

Procedure for Text Classification

Here's how they proceed:

- Given two phrases (samples) x1 and x2, let x1x2 denote their concatenation.
- Let C(.) represent the compressed length. For example, C(x1x2) - C(x1) indicates the number of bytes required to encode x2 based on x1's information. They introduce a new distance: Normalized Compression Distance (NCD): NCD(x1, x2) = (C(x1x2) - C(x1)) / max(C(x1), C(x2)).

- To determine the class c of a sample:
    1. For each c, concatenate all samples in $d(c)$ from the training set belonging to c.
    2. Compress the concatenation $d(c)$ as a long document to obtain the compressed length $C(d(c))$.
    3. Concatenate the test sample $d(u)$ with $d(c)$ and compress to obtain $C(dcdu)$.
    4. The predicted class is $\text{argmin}_c(C(dcdu) - C(d(c)))$.

Applying PyChest for Changepoint Detection

We will apply this classification method to our changepoint detection problem using the PyChest library.

Our test object is a text where a subject change occurs in the middle; initially discussing biology and suddenly shifting to artificial intelligence. Here is the text:

> "The process of photosynthesis is crucial to life on Earth. Plants, algae, and certain bacteria use this biochemical process to convert light energy into chemical energy stored in glucose molecules. Through photosynthesis, carbon dioxide is absorbed and oxygen is released, sustaining the oxygen levels necessary for aerobic respiration in animals. This intricate process involves two main stages: the light-dependent reactions occurring in the thylakoid membranes of chloroplasts, and the light-independent reactions (Calvin cycle) in the stroma. Understanding photosynthesis is fundamental not only to biology but also to our understanding of climate regulation and sustainable energy solutions. Artificial intelligence (AI) continues to revolutionize various industries, from healthcare to finance and beyond. AI algorithms, powered by machine learning and neural networks, analyze vast datasets to uncover patterns and make predictions. In healthcare, AI aids in diagnostics and personalized medicine, while in finance, it optimizes trading strategies and risk management. Ethical considerations surrounding AI's impact on employment and privacy are ongoing topics of debate. As AI technology advances, its integration into everyday life raises questions about regulation, transparency, and the societal implications of machine intelligence."

For our experiment, we begin by compressing the text using Gzip, resulting in a list of bytes, which corresponds to a list of numbers. From this list, we remove the first 10 numbers (corresponding to the Gzip version, compression date, type, etc.) and the last 15 numbers (corresponding to the footer and the size of uncompressed data).

We then apply PyChest by adjusting parameters, notably ‘min-distance‘, which corresponds to the ”minimum normalized stationary segment length” mentioned in Mme Khaleghi’s article on PyChest [6]. I set it to 0.5, which seemed reasonable since the changepoint in the original text occurs in the middle. A too low value of min-distance returned changepoints that were too weak. We obtained a changepoint at position 413 for a list of size 703 (after modifications), which seemed consistent...

Verifying the Changepoint

We face a new problem: How to verify that the detected changepoint is correct. It is not possible to decompress only a part of a Gzip-compressed file because the compression uses a block-based method where each block depends on the previous data for proper decompression. Thus, trying to decompress only a part of the compressed file (e.g., half) misses the necessary information used in the previous compression process.

We found a method to verify the detected changepoint; while not 100% reliable I think, it works:

I compress the text incrementally, starting with the first 100 words, then the first 200, and so on until the entire text. Each time I compress, I check the size of the compressed text and display the original text incrementally. When the 100 words containing the initial changepoint are added, I compare the size of the compressed text before and after and deduce an interval in which the changepoint should be found. This method works perfectly when keeping 0.5 as the ”minimum normalized stationary segment length,” as PyChest finds a changepoint at 413, and the interval is [402,458]. See the example from our test.

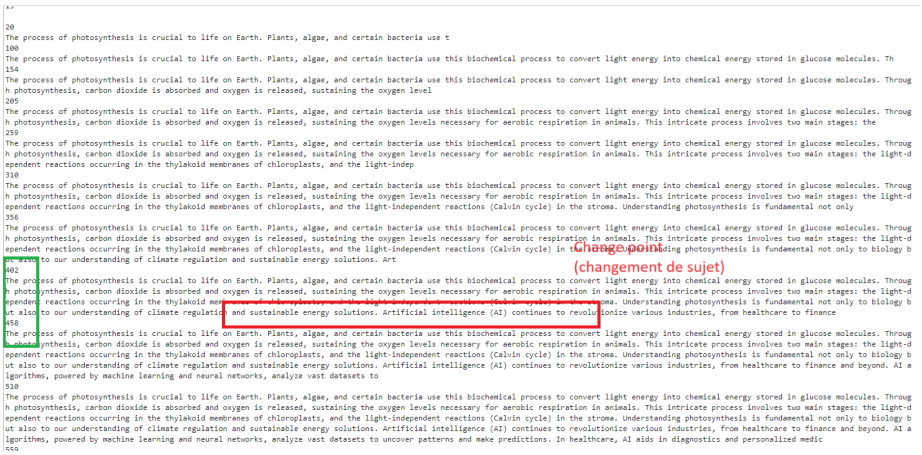


Figure 6.7: Méthode de vérification de détection de change-points

As we decrease the number of words added (instead of 100, try 80, 50, 20,

then 10), we pinpoint exactly 413 (the position detected with PyChest).

Chapter 7

Conclusion

n conclusion, my internship at CREST has been an incredibly rewarding experience, allowing me to delve deeply into the realm of Large Language Models and related technologies. Throughout the internship, I was able to apply and expand my knowledge of state-of-the-art models, neural networks, and transformers, and engage in hands-on experimentation with Phi3 and LLama3. The exploration of feature engineering, word embeddings, and advanced techniques like Gzip compression has significantly broadened my understanding of both theoretical and practical aspects of machine learning.

I am particularly grateful for the guidance and support provided by Ms. Khaleghi. Her expertise and mentorship were crucial in navigating the complexities of the projects and in fostering my professional growth. This internship has not only enhanced my skills but also deepened my appreciation for the intricacies of LLMs and their applications.

For further details on the work completed during this internship, please refer to the GitHub links provided: GitHub Link. Should you have any questions or need additional information, do not hesitate to contact me at tilian.bourachot@gmail.com.

Bibliography

[1] Ghizlane Bourahouat, Manar Abourezq, and Najima Daoudi. Word embedding as a semantic feature extraction technique in arabic natural language processing: an overview. *Int. Arab J. Inf. Technol.*, 21(2):313–325, 2024.

[2] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in Neural Information Processing Systems*, 35:16344–16359, 2022.

[3] Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

[4] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

[5] Azadeh Khaleghi, Daniil Ryabko, Jérémie Mary, and Philippe Preux. Consistent algorithms for clustering time series. *The Journal of Machine Learning Research*, 17(1):94–125, 2016.

[6] Azadeh Khaleghi and Lukas Zierahn. Pychest: a python package for the consistent retrospective estimation of distributional changes in piece-wise stationary time series. *arXiv preprint arXiv:2112.10565*, 2021.

[7] Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[9] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *arXiv preprint arXiv:2308.07633*, 2023.

Chapter 8

Annex

8.1 Annex : FlashAttention

FlashAttention [2]

- Motivation :
  - modelling longer distance sequences
  - NLP : large context required to understand books, plays, instruction manuals time series, audio, video, data naturally modeled as sequences of millions steps
- Challenge : How to scale Transformers to longer sequences ?
- Several solutions to memory-efficient attention :
  - One row at a time
  - One block at a time
- Here come FlashAttention : fast and memory-efficient algorithm for exact attention
- Key algorithmic ideas :
  - Tiling : Restructure algorthm to load block by block from HBM to SRAM to compute attention. Compute softmax reduction without access to full input
  - Recomputation : dont store attention matrix from forward, recompute it in the backward.
- The upshot : faster model training, better models with longer sequences

8.2 Annex : High quality data

When They deal with high quality data in the Report of phi3, they refers to another paper from Microsoft Research [4]. BUT this article deals with phi1 who was specialised in Python Coding.

Central Focus on Textbook-Quality Training Data:

The model's success relies on high-quality training data derived from textbooks. This dataset includes under 1 billion tokens of GPT-3.5 generated Python textbooks. These textbooks are synthesized to provide a high-quality source of natural language text interspersed with relevant code snippets. The content is specifically targeted to cover topics that promote reasoning and basic algorithmic skills. Diversity is achieved by setting constraints on the topics and the target audience of the generated textbooks.

Pre-training phase : they use CodeTextbook.

To fine tune the model, they use CodeExercices. This dataset is a small synthetic collection of Python exercises and solutions, containing less than 180 million tokens. Each exercise consists of a docstring for a function that needs to be completed, aiming to train the model for function completion tasks based on natural language instructions. Generated by GPT-3.5, diversity is achieved by constraining the function names. Explicit decontamination and alternative evaluations are conducted to ensure that problems similar to those from the HumanEval benchmark are not encountered during finetuning.

This contrasts with previous approaches using standard sources like The Stack and web-based datasets.

Critique of Standard Code Datasets:

- Existing datasets are not ideal for teaching algorithmic reasoning and planning.
- Issues identified in these datasets include:
- Lack of Self-Containment: Many code snippets depend on external modules or files, making them hard to understand in isolation.
- Trivial Content: Many examples consist of non-instructive, trivial code such as defining constants or setting parameters.
- Buried Algorithmic Logic: Useful algorithmic content is often hidden within complex or poorly documented code.
- Skewed Distribution: There's an imbalance in the representation of coding concepts, favoring certain topics over others.

Challenge of Creating High-Quality Datasets:

- Ensuring examples are diverse and non-repetitive is a major challenge.
- Diversity in examples is crucial for exposing the model to various coding concepts, skills, and scenarios.

Importance of Diversity:

- Helps the language model learn different ways of expressing and solving problems in code.
- Reduces the risk of overfitting or memorizing specific patterns or solutions.
- Enhances the model's generalization and robustness to unseen or novel tasks.

Inducing Creativity and Diversity:
-Finding the right "trick" to induce creativity and diversity while maintaining quality and coherence is essential. -Inspired by [3], where diversity was achieved by including a random subset of words from a fixed vocabulary in the prompt.

TinyStories: How Small Can Language Models Be and Still Speak Coherent English? Ronen Eldan and Yuanzhi Li

In this work, they presented TinyStories, a synthetic dataset of short stories generated by GPT-3.5 and GPT-4, using words typically understood by 3 to 4-year-olds. TinyStories is used to train and evaluate small language models (SLMs) that are significantly smaller than state-of-the-art models but still produce fluent, consistent, and grammatically correct stories. These models demonstrate reasoning capabilities and diversity in their outputs.

While large models trained on vast internet datasets show impressive capabilities, such datasets are too large for SLMs to effectively capture language complexity. TinyStories enables the study of capabilities like coherent text generation, reasoning, and instruction following on a smaller scale. Training SLMs on TinyStories has revealed behaviors similar to large language models (LLMs), such as scaling laws and trade-offs between model width and depth.

Moreover, SLMs trained on TinyStories offer higher interpretability, allowing visualization and analysis of their attention and activation patterns to understand story generation and comprehension. The study provides evidence that models trained on TinyStories can produce genuinely new stories, not just replicate text from the dataset. They have also introduced a new paradigm for the evaluation of language models, which uses GPT-4 to grade the content generated by these models as if those were stories written by students and graded by a (human) teacher.

Conclusion : We cannot determine if data is of good quality or not; it is up to us to assess whether we think the data is of good quality. There are no predefined criteria.

8.3 Annex 2 : how do they determine the number of dimensions ?

https://www.baeldung.com/cs/dimensionality-word-embeddings

Parameters do not hold any inherent meaning. Instead, they function collectively, working in unison to map intricate relationships between words and phrases in the training data. The **dimensionality** of word embedding refers to the number of dimensions in which the vector representation of a word is defined. This is typically a fixed value determined while creating the word embedding. The dimensionality of the word embedding represents the total number of features that are encoded in the vector representation.

Deciding on the appropriate dimensionality for a word embedding depends on several factors, such as the size and nature of the dataset we are using, the specific NLP task we are working on, and the computational resources available to us.//

Size of the Dataset

- Larger datasets can support higher-dimensional embeddings due to more training data.
- Datasets with less than 100,000 sentences: Benefit from lower-dimensional embeddings (50-100 dimensions).
- Larger datasets: Benefit from higher-dimensional embeddings (200-300 dimensions).

NLP Task

- Task requirements impact embedding dimensionality: High semantic accuracy tasks (e.g., sentiment analysis, machine translation): Benefit from higher-dimensional embeddings. Easier tasks (e.g., named entity recognition, part-of-speech tagging): May not require high-dimensional embeddings.

Computational Resources

- Higher-dimensional embeddings require more memory and processing power

In conclusion, to find the Right Number of Dimensions, we need to Experiment with different dimensionalities and evaluate model performance on a validation set and adjust dimensionality to find the optimal balance between semantic accuracy and computational efficiency for the specific use case. In phi3-mini, each vectors has 3072 dimension..