



Accelerating Dictionary-based Sentiment Analysis with GPGPUs

Emmanouela Theodoraki

Thesis submitted in fulfillment of the requirements for the

Diploma of Electrical and Computer Engineering

Technical University of Crete

School of Electrical and Computer Engineering

University Campus, Akrotiri, Chania, GR-73100, Greece

Thesis Committee:

Assoc. Prof. *Sotiris Ioannidis* (Supervisor)

Prof. *Apostolos Dollas*

Prof. *Michael Zervakis*

June 2023

Abstract

Sentiment analysis is a natural language processing (NLP) technique that extracts subjective information such as opinions and emotions from textual data. The rapid growth of online social networks and the vast amount of content generated by their users has led the research community to dedicate a significant amount of study to the development of effective analysis techniques in this field. In addition, sentiment analysis has wide application in various areas, such as brand intelligence and market research, political campaigns, and spam detection, among others.

The goal of this thesis is to develop the algorithms and tools that enable the acceleration of dictionary-based sentiment analysis using General Purpose Graphics Processing Units (GPGPUs) and other multi-core processors. To achieve this, we design and implement a data-parallel sentiment analysis system that extends previous literature on data-parallel pattern matchers, based on the Aho-Corasick algorithm, using thousands of data blobs as input, simultaneously. This system is able to analyze large feeds of data (e.g., Twitter feeds) and assign the respective scores to the content.

Also, we re-design and implement sentiment analysis techniques found in popular tools, such as Vader, aiming to provide fast and accurate sentiment analysis results. We implement the core engine of our system using C/OpenCL, enabling it to execute on a large variety of devices and evaluate our system using a large corpus of Twitter feeds related to the COVID-19 pandemic.

We compare our sentiment analysis tool against state-of-the-art solutions found in the literature, utilizing both lexicon-based sentiment analysis and machine learning and identify that our proposal can outperform them in computational speed by orders of magnitude while providing the same accuracy. This work provides a fast and accurate sentiment analysis tool that can execute on commodity systems without modifications, operating either as a stand-alone tool or as a library that can be embedded in other applications, allowing users to obtain sentiment analysis results in an almost real-time fashion.

Περίληψη

Η ανάλυση συναισθήματος (Sentiment Analysis - SA), είναι μια τεχνική επεξεργασίας φυσικής γλώσσας (ΝΛΠ) που αναγνωρίζει υποκειμενικές πληροφορίες, όπως απόψεις και συναισθήματα σε περιεχόμενο κειμένου. Η ραγδαία ανάπτυξη των μέσων κοινωνικής δικτύωσης και ο μεγάλος όγκος περιεχομένου που παράγεται από τους χρήστες τους, έχει οδηγήσει την επιστημονική κοινότητα να αφιερώσει σημαντικό ποσοστό της έρευνας στην ανάπτυξη αποτελεσματικών τεχνικών ανάλυσης για το πεδίο αυτό. Επιπλέον, η ανάλυση συναισθήματος έχει ευρεία εφαρμογή σε πολλούς τομείς, όπως στο **brand intelligence** και στην έρευνα αγοράς, στις πολιτικές καμπάνιες, στο **spam detection**, κ.ά.

Ο στόχος της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη αλγορίθμων και εργαλείων που επιταχύνουν την ανάλυση συναισθήματος βασισμένη σε λεξικά χρησιμοποιώντας επεξεργαστές γενικού σκοπού (GPGPUs) και άλλους πολυπύρηνους επεξεργαστές. Για να το επιτύχουμε αυτό, σχεδιάζουμε και υλοποιούμε ένα σύστημα ανάλυσης συναισθήματος με παράλληλη επεξεργασία δεδομένων, το οποίο επεκτείνει υπάρχουσα μελέτη για παράλληλη αναζήτηση αλφαριθμητικών προτύπων, βασισμένη στον αλγόριθμο Aho-Corasick, χρησιμοποιώντας χιλιάδες **blobs** δεδομένων ως είσοδο, ταυτόχρονα. Το σύστημα αυτό, είναι ικανό να αναλύει μεγάλες ροές δεδομένων (π.χ. Twitter feeds) και να αναθέτει τα αντίστοιχα σκορ στο περιεχόμενο.

Ακόμα, υλοποιούμε και επανασχεδιάζουμε τεχνικές SA που χρησιμοποιούνται σε δημοφιλή εργαλεία, όπως το Vader, με στόχο να παρέχουμε γρήγορα και ακριβή αποτελέσματα συναισθηματικής ανάλυσης. Υλοποιούμε την κύρια μηχανή του συστήματός μας με τη χρήση C/OpenCL, δίνοντας τη δυνατότητα να εκτελείται σε μία μεγάλη ποικιλία συσκευών. Η απόδοση του συστήματός μας αξιολογείται χρησιμοποιώντας ένα μεγάλο σύνολο κειμένων από Twitter feeds τα οποία αναφέρονται στην πανδημία COVID-19.

Τέλος, συγκρίνουμε την προσέγγισή μας για ανάλυση συναισθήματος με τις προηγμένες λύσεις που υπάρχουν στη βιβλιογραφία, χρησιμοποιώντας τόσο αναλύσεις συναισθήματος που βασίζονται σε λεξικά (Lexicon-based), όσο και σε μηχανική μάθηση (Machine Learning - ML) και διαπιστώνουμε ότι η πρότασή μας μπορεί να τις ξεπεράσει σε υπολογιστική ταχύτητα κατά πολλές τάξεις μεγέθους, διατηρώντας την ίδια ακρίβεια. Αυτή η εργασία, παρέχει ένα γρήγορο και ακριβές εργαλείο ανάλυσης συναισθήματος το οποίο μπορεί να εκτελείται σε κοινά συστήματα γενικής χρήσης χωρίς τροποποιήσεις. Καταλήγοντας, το σύστημά μας λειτουργεί είτε ως ένα αυτόνομο εργαλείο είτε ως μία βιβλιοθήκη που μπορεί να ενσωματωθεί σε άλλες εφαρμογές, επιτρέποντας στους χρήστες να αποκτήσουν αποτελέσματα συναισθηματικής ανάλυσης σε σχεδόν πραγματικό χρόνο.

Acknowledgments

Completing this thesis, as part of my Diploma in Electrical and Computer Engineering has been a challenging yet rewarding journey, and I am deeply grateful to those who have supported me along the way.

First and foremost, I would like to thank my supervisor, Prof. Sotiris Ioannidis for his trust in assigning me this thesis and his valuable guidance through our constructive conversations. I am grateful for the opportunity he gave me to work with his team at FORTH and for all the resources and support they provided.

Special thanks also go to the members of my thesis committee, Prof. Apostolos Dollas and Prof. Michael Zervakis for their constructive comments and questions during the evaluation of my work.

Furthermore, my sincere appreciation goes to Dimitris Deyannis for his unwavering guidance throughout the development of this work. His support, advice, and insightful ideas have been invaluable, and I am grateful for his continuous assistance.

I would also like to acknowledge the insightful collaboration and assistance provided by Despoina Antonakaki and Alex Shevtsov, which significantly enhanced the quality of this research.

Last but not least, I would like to express my deepest gratitude to my family for their immense and selfless support and patience during these years of my studies. I would also like to thank George for his unwavering support and constant presence by my side all this time. His encouragement and belief in me have been a consistent source of motivation. This work would never have been possible without them (and coffee).

Contents

1	Introduction	1
1.1	Contributions	2
1.2	Outline	3
2	Background	5
2.1	Natural Language Processing	5
2.1.1	Text Mining and Polarity Detection	7
2.1.2	Sentiment Analysis	8
2.1.2.1	Techniques	8
2.1.2.2	Use Cases	10
2.2	String Matching	12
2.2.1	Single-pattern	13
2.2.2	Multi-pattern	13
2.2.3	Other Classifications	14
2.2.4	The Aho-Corasick Algorithm	14
2.3	Parallel Computing	16
2.3.1	Types of Parallelism	16
2.3.2	Flynn's Taxonomy	17
2.3.3	Hardware for Parallelism	17
2.3.4	Frameworks for Parallel Computing	19
3	Design & Implementation	21
3.1	Architecture	21
3.2	Sentiment Analysis Life-cycle	22
3.3	DFA Construction	23
3.4	Data Pre-processing	25
3.5	Parsing and Tokenizing the Input	27
3.6	OpenCL-based Pattern Matching	28
3.6.1	Data Buffer	28
3.6.2	Score Assignment	30
3.7	Sentiment Analysis	31
3.7.1	Sentiment Report	35

3.8	Optimizations	35
3.8.1	Single Data Preparation Pass	35
3.8.2	Special Word and Character Encoding	35
3.8.3	Ring Buffer	36
3.8.4	Lexicon Automaton Memory Savings	36
4	Evaluation	39
4.1	Experimental Setup	39
4.2	Datasets	39
4.3	Lexicon	40
4.4	Performance Baseline	41
4.4.1	Vader Performance Analysis	41
4.4.2	Machine-Learning Approaches	42
4.5	GPU-Vader Performance Analysis	43
4.5.1	Overall Execution Time	43
4.5.2	GPU-based polarity Performance	44
4.5.3	GPU Engine Performance	45
4.6	End-to-End Performance Comparison	46
4.7	Execution on the CPU Die	46
4.8	Throughput Comparison	48
5	Related Work	51
5.1	Sentiment Analysis	51
5.2	Machine Learning approaches	52
5.3	Deep Learning Approaches	53
5.4	Lexicon-based Approaches	55
5.5	Hybrid Approaches	56
5.6	Sentiment Analysis on Twitter	57
5.7	GPGPU Acceleration	58
6	Conclusions and Future Work	61
6.1	Summary of Contributions	61
6.2	Future Work	61
6.3	Conclusion	62
7	List of Acronyms	63

List of Figures

2.1	Sentiment analysis approaches	11
2.2	The Aho-Corasick functions using the pattern set $\{she, he, hers, his\}$. . .	15
2.3	Parallel architectures according to Flynn’s taxonomy.	18
3.1	Architecture overview.	22
3.2	Sentiment analysis life-cycle overview.	23
3.3	Serialized Aho-Corasick DFA represented as a two-dimensional integer array with negative values indicating final states.	24
3.4	Tweet table overview.	29
3.5	<code>databuff</code> structure overview.	30
3.6	Overview of the four stage ring data buffer.	37
4.1	Time required by Vader to process the various datasets presented as a high-level function breakdown and as a percentage of the overall time. . .	41
4.2	Time breakdown required by Vader’s <code>polarity</code> function to perform the sentiment analysis on the loaded data.	42
4.3	End-to-end performance comparison of <code>DeBERTa</code> and <code>XML-RoBERTa</code> when processing the Twitter workload.	43
4.4	Time required by the GPU-based sentiment analysis system to process the datasets, presented as a high-level function breakdown and as a percentage of the total time.	44
4.5	Time required by the GPU-based <code>polarity</code> function to process the workloads presented at a function-level breakdown.	44
4.6	Time required by our GPU engine to process the workloads, presented at a function-level breakdown.	45
4.7	End-to-end performance comparison between <code>XML-RoBERTa</code> , <code>DeBERTa</code> , Vader and our GPU-based sentiment analysis system when processing workloads containing 1K - 200K tweets.	46
4.8	Time required by the CPU and iGPU to process the datasets using our OpenCL-based sentiment analysis tool, presented as a high-level function breakdown and as a percentage of the total time.	47
4.9	Time required by our OpenCL engine to process the workloads using the CPU and iGPU, presented at a function-level breakdown.	47

4.10	Throughput comparison between our system executed on GPU, CPU and iGPU, the vanilla Vader implementation, XML-RoBERTa, and DeBERTa.	48
4.11	Throughput comparison between our system executed on GPU, CPU and iGPU for datasets ranging from 100K to 500K tweets.	49

List of Tables

3.1	Metadata provided with each tweet in datasets retrieved via the Twitter API.	26
3.2	Sample of special words encoded into our modified Vader's lexicon. . . .	36
4.1	Test files generated by pre-processing Twitter datasets containing Tweets related to the COVID-19 topic.	40

Chapter 1

Introduction

The rapid growth of online social networks has revolutionized communication on the web. With the widespread availability of platforms such as Twitter, Facebook, and Instagram, millions of users now have the ability to express their thoughts, opinions, and feedback on a wide range of topics. This surge in user-generated content has generated a need for advanced techniques that can effectively analyze and understand the sentiment and opinion expressed within these texts.

Sentiment analysis (SA), also known as opinion mining, has emerged as a crucial research area in natural language processing (NLP). Its primary objective is to extract subjective information, such as attitudes, opinions, and emotions, from textual data. Given the vast amount of user-generated content available on social media platforms, SA offers valuable insights into public opinion towards various topics, including products, services, politics, events and more. This research interest stems from the increasing recognition that understanding public sentiment has a profound impact on decision-making processes, marketing strategies, and policy development.

The polarity of text can be determined by a range of sentiment analysis techniques and methodologies (i.e., whether it conveys a positive, negative, or neutral sentiment). By utilizing computational linguistics, machine learning, and statistical approaches, SA enables automated identification and extraction of sentiment-related information from text data. The ultimate goal of SA is to gain a deeper understanding of the overall feeling surrounding a particular subject matter.

In the wake of the COVID-19 pandemic, analyzing sentiments expressed in tweets related to the virus has become particularly important for understanding public perceptions, identifying misinformation, and tracking the emotional impact of the pandemic. The COVID-19 pandemic stood out as an ideal test case due to its prolonged presence as a top trending topic and the enduring interest it garnered from users over a period of at least two years. Furthermore, the availability of a vast dataset related to the long-lasting and widely-discussed topic of COVID-19 allowed for comprehensive analysis, surpassing the

limited temporal scope of other events and subjects that may only trend for a few days or months.

Despite the increasing demand for sentiment analysis, traditional methods are faced with limitations that restrict their scalability and efficiency. Many existing approaches, such as machine learning-based techniques, are computationally expensive, time consuming and require substantial resources, making them unsuitable for large-scale analysis. Moreover, while lexicon-based sentiment analysis tools exist, they often lack efficiency and are primarily implemented in Python.

To address this challenge, this thesis proposes a lexicon-based, sentence-level sentiment analysis tool with GPU acceleration, implemented in C/OpenCL. The proposed tool employs the Aho-Corasick algorithm for string matching and utilizes multi-core processors such as GPUs, iGPUs and CPUs, for parallel execution, enabling faster sentiment analysis of large-scale datasets. The proposed sentiment analysis tool offers significant advantages over traditional sentiment analysis methods, including high performance, scalability, and accuracy. The tool's effectiveness is demonstrated through extensive comparisons, which show that our proposed model outperforms state-of-the-art sentiment analysis tools in terms of efficiency. The tool's scalability is also demonstrated by its ability to analyze large volumes of data in an almost real-time fashion, making it suitable for real-world applications.

To demonstrate the effectiveness and efficiency of our approach, we compare our tool's performance against the widely-used lexicon-based sentiment analysis model, Vader, and the popular Machine Learning systems XML-RoBERTa and DeBERTa. We show that our data-parallel approach outperforms any other system in terms of processing speed, while maintaining the accuracy of Vader. Through this thesis, we aim to bridge the research gap in lexicon-based sentiment analysis approaches, by providing a faster and more efficient solution.

1.1 Contributions

The contributions of this work are the following:

- We provide the first, to our knowledge, data-parallel lexicon-based sentiment analysis tool, based on Vader and implemented in OpenCL, that can execute on a wide variety of multi-core processors, such as GPUs, CPUs, and iGPUs.
- The system is able to outperform the state-of-the-art Vader sentiment analysis system by up to 20 times, while achieving the same accuracy.
- We evaluate our system using real-world Twitter datasets and lexicon, and provide a thorough comparison with other lexicon- and ML-based sentiment analysis approaches.
- We identify and implement a set of performance optimizations over Vader's original implementation, many of which could be directly adopted by the vanilla Vader.

1.2 Outline

The rest of this dissertation is organized as follows. Chapter 2 provides an overview of the background concepts related to natural language processing, sentiment analysis, string matching, and parallel computing. Chapter 3 describes the architecture of the proposed sentiment analysis tool and presents its implementation details in terms of data preprocessing, DFA construction, tweet table creation, pattern matching, sentiment analysis using the VADER library, and optimizations. In Chapter 4, we evaluate the performance of our tool through extensive experiments, comparing it against existing approaches. Chapter 5 discusses related work in sentiment analysis, machine learning approaches, and GPGPU acceleration. Finally, in Chapter 6, we conclude the thesis, summarizing our contributions and outlining future research directions.

Chapter 2

Background

2.1 Natural Language Processing

The field of computer science has long been focused on enabling machines to recognize written and spoken language but not truly communicate with people. Natural Language Processing (NLP) is a subfield of AI that seeks to address this challenge by equipping computers with the ability to understand and interpret the contextual meaning of the information they receive, similar to how humans do. The objective of NLP is to enable computers to comprehend a given text or speech with high accuracy, which can be very useful in a wide range of AI applications [1]. At its core, NLP involves converting unstructured text into a structured representation that can be processed by a computer. This process is known as Natural Language Understanding (NLU). NLP can also involve converting structured data into unstructured text, a process referred to as Natural Language Generation (NLG).

NLP typically starts with the process of pre-processing the text. This process may include tasks such as tokenization, stemming, and lemmatization, which aid in the preparation and cleaning of raw text data for further analysis. Tokenization breaks down a text into smaller units, such as words, phrases, or sentences, called tokens. The rest of the process is performed for one token at a time, allowing the text to be analyzed at a more granular level. Stemming is the process of deriving the root or "stem" form of a word by removing prefixes and suffixes, and normalizing the tense. For example, the words "running", "runs", and "ran" can all be stemmed from the root form "run". Lemmatization, on the other hand, involves identifying the root or "lemma" form of a word based on its meaning and the grammar of the sentence through a dictionary definition to produce the base form. For example, the word "better" could be lemmatized to "good", depending on its usage in a sentence.

Part-of-speech (POS) tagging identifies and labels the parts of speech (nouns, verbs, adjectives, etc.) for each token and finds where it is used within the context of a sentence. For example, the word "make" has a different meaning and POS in the phrases "make of

laptop" and "make dinner". Another way to highlight features in the text is Named Entity Recognition, a process that identifies if there is an entity associated with a given token. For example, the token "Arizona" has the entity "U.S. State", while the token "Ralph" has the entity "a person's name".

Dependency parsing involves analyzing the grammatical structure of a sentence and identifying the relationships between its words, such as the subject, verb, and object. Semantic role labeling identifies the role that each word plays in a sentence, such as the agent, patient, or recipient. Finally, sentiment analysis involves analyzing the sentiment or emotional tone of a given text, such as whether it is positive, negative, or neutral.

In the second phase of NLP, an algorithm is developed to work with all the features that are revealed after the data has been preprocessed. As NLP is a rapidly evolving field, there are various approaches that can be used. In this section, we will provide an overview of the most commonly used algorithms for processing natural language data.

Rule-based approaches Rule-based approaches rely on carefully designed linguistic rules that are based on the syntax, semantics, and structure of the language and can be used to identify and extract specific pieces of information from the data. One of the main advantages of rule-based approaches is that they are relatively easy to implement and can be highly accurate when applied to well-defined tasks. However, they can be inflexible and may not perform well when faced with data that do not conform to the predefined rules.

Statistical approaches Statistical approaches use statistical models and machine learning algorithms to analyze and understand natural language data. These approaches rely on large amounts of annotated data, which is used to train the models and make predictions about the meaning and structure of the data. Statistical approaches can be highly effective, but they require a large amount of data and computational resources. Also, they may not be able to capture subtle nuances that are important for certain tasks.

Neural network-based approaches Neural network-based approaches employ artificial neural networks, which are inspired by the structure and function of the human brain, to process and analyze natural language data. These approaches can be highly effective at tasks such as language translation and text classification, but they may require a large amount of data and computational resources to train. Also, they are not always able to explain their decisions or predictions.

Hybrid approaches Hybrid approaches to NLP involve combining multiple approaches, such as rule-based and statistical approaches, to improve the accuracy and effectiveness of NLP systems. These approaches can be useful when different approaches are better suited for different parts of a task or when the data is too complex for a single approach to handle effectively.

Knowledge-based approaches Knowledge-based approaches use a collection of facts and information about a particular domain, which is a knowledge base, to understand and interpret natural language data. These approaches can be highly effective at tasks such

as question answering and natural language generation, but they may be limited by the completeness and accuracy of the knowledge base.

Ontology-based approaches Ontology-based models use a representation of a domain's concepts and relationships (ontology), to understand and interpret natural language data. These approaches can be useful for tasks such as information extraction and text classification, but they may be limited by the complexity and coverage of the ontology.

2.1.1 Text Mining and Polarity Detection

Data mining is a process of extracting valuable information from large datasets which can be used to make more informed decisions. With the evolution of data warehouses and big data, it has become necessary to process huge amounts of data and transform it into useful knowledge since patterns and trends can be identified in them. One important advantage is that we can make predictions about future trends and how things might develop in the future by analyzing past data. Additionally, it can help to uncover relationships between different pieces of data that would have otherwise been difficult to detect. To effectively use data mining, the process begins with setting the objectives (define a business problem that data mining can be applied to), preparing the data (e.g., which set of data will answer the pertinent questions), applying the appropriate data mining algorithms, and finally, evaluating the results to ensure their validity, novelty, usefulness, and understandability. Some of the most popular data mining techniques are the following:

- *Association* is a rule-based method that identifies relationships between variables in a given dataset by correlating multiple items with the same type to find patterns.
- *Classification* is used to build an idea of the type of customer/item by describing multiple attributes.
- *Clustering* groups individual pieces of data to form a structure.
- *Deep Learning* techniques utilize Artificial Neural Networks (ANN) to make predictions. If input data is labeled, a regression can be applied to predict the likelihood of an assignment. If input data is unlabeled, the training datasets can be compared with one another and clustered.

Textual data is one of the most widely used data types within databases, and can be classified into three categories: (i) Unstructured, (ii) Semi-structured, or (iii) Structured. Structured text is arranged into a tabular format with rows and columns, which facilitate querying, filtering, processing, and analytics. Unstructured text, on the other hand, has no predefined format and can include any type of text. Semi-structured data has some structure, but not enough to meet the requirements of a relational database, such as JSON code. Approximately 80% of all data in the world is estimated to be in an unstructured format [2].

Text mining is the process of analyzing vast amounts of textual materials to capture key concepts, trends, and underlying relationships. It involves transforming unstructured text

into a structured format to obtain meaningful insights and patterns. The text mining process is divided into stages, (i) identifying the text to be mined, (ii) text processing to standardize the format and remove noise (e.g., removing stopwords, tokenization, lemmatization, part of speech tagging, bag of words, etc.), (iii) concept and category creation, and (iv) analysis to make predictions and discover relationships.

The use of NLP to comprehend the language utilized helps to decrease the ambiguity of the text, making text mining in the linguistics area a more reliable approach. Therefore, in the text mining process, the fourth stage of relationship discovery and prediction analysis is implemented through Data Mining.

2.1.2 Sentiment Analysis

Sentiment Analysis (SA), also referred to as opinion mining or polarity detection, is a field of NLP that aims to identify, classify, and derive the sentiment polarity of written text. Through this process, the emotional tone of the text can be understood. SA seeks to extract subjective data or opinions expressed in written text or speech in order to determine the writer's attitude, intent, and feelings. Sentiment scoring is a core aspect of NLP and is typically achieved by recognizing the fine-grained emotion behind an opinion such as happiness, anger and sadness, or by quantifying the level of positivity and negativity. However, in most cases, the spectrum of emotions is divided into three categories, (i) negative, (ii) positive, and (iii) neutral — called a 3-class classification. This process involves various techniques, such as language identification, sentence parsing, word segmentation, stemming and lemmatization, tokenization, POS, chunking, and syntax parsing, used to format data appropriately for analysis. The sentiment score is then calculated based on factors such as the number and type of emotions expressed, the strength of those emotions, and the context in which they are used. However, without considering the context of the text, sentiment can be incorrectly interpreted, as in real life, resulting in context errors in a few general classes, such as polysemy, multi-polarity, sarcasm, and irony.

2.1.2.1 Techniques

Researchers have investigated sentiment at various levels of text granularity, including document level, sentence level, and aspect (or feature) level. These levels allow for a comprehensive understanding of sentiment within textual data, capturing sentiment from overall documents to individual sentences and specific aspects or features for fine-grained analysis. The existing literature usually classifies sentiment analysis approaches into three main categories, (i) Machine Learning approaches, (ii) Lexicon-Based approaches, and (iii) Hybrid methods [3], [4]. In this section, we provide an overview of the most used approaches to perform sentiment analysis. Figure 2.1 shows a comprehensive visual representation of the sentiment analysis approaches.

Machine learning approaches Among the different sentiment analysis approaches, Machine Learning approaches have gained significant attention and emerged as the most widely-used methods. They leverage the power of machine learning algorithms and lin-

guistic features to perform sentiment classification and can be further classified into three subcategories: (i) supervised, (ii) semi-supervised, and (iii) unsupervised techniques [5].

Supervised learning methods have gained popularity in sentiment analysis due to their ability to achieve high accuracy. These approaches require labeled training data to learn sentiment patterns and make predictions. Linear classifiers, also known as deterministic classifiers, utilize statistical approaches to classify sentiment based on linear or hyper-plane decision boundaries. Support Vector Machines (SVMs) are a popular example that separates data linearly or non-linearly and can handle both discrete and continuous variables. Artificial Neural Networks (ANNs) extract features from linear combinations of input data and model the output as a non-linear function of these features.

Decision tree-based algorithms, such as C4.5 and Random Forest, decompose the training data space hierarchically and classify input data into predefined classes, allowing for the creation of interpretable classification models. Rule-based approaches rely on predefined linguistic rules or heuristics to determine sentiment. These methods utilize explicit linguistic patterns or sentiment lexicons to make sentiment predictions. Probabilistic classifiers, also known as generative classifiers, predict a probability distribution over a set of classes based on Bayes' theorem. These classifiers estimate the conditional distribution of the class label given the document to be classified. Naïve Bayes is a simple yet widely used algorithm based on the assumption of word independence and BOW feature extraction, while Maximum Entropy (ME), also known as a conditional exponential classifier or Maxent classifier, shows high accuracy when combined with unigrams and bigrams as features. Ensemble learning combines the predictions of multiple classifiers to enhance overall sentiment classification accuracy.

Deep learning models such as Convolutional Neural Networks (CNNs), Deep Neural Networks (DNNs), and Recurrent Neural Networks (RNNs) have been extensively employed in sentiment analysis at the document, sentence, or aspect level. These models capture complex relationships in the data and have shown impressive performance in various natural language processing tasks, including sentiment analysis. DNN models, including deep feedforward networks and deep convolutional networks, have demonstrated their efficacy in sentiment analysis by automatically learning hierarchical representations. CNNs excel in capturing local patterns and spatial relationships in text, making them suitable for sentiment classification tasks. RNNs, such as Long Short-Term Memory (LSTM) networks, are effective in modeling sequential dependencies and have been applied to sentiment analysis to capture context information.

Semi-supervised learning methods utilize training datasets that contain both labeled and unlabeled data. Several approaches fall under this category, including generative, co-training, self-training, graph-based, and multi-view learning approaches. These methods leverage the unlabeled data to enhance the sentiment classification performance.

Unsupervised learning techniques for sentiment analysis focus on leveraging external resources, knowledge bases, and lexicons to infer sentiment. These approaches do not require labeled data for training and two common unsupervised approaches are Hierarchical

and Partition methods.

Lexicon based approaches These sentiment analysis methods, also called knowledge-based, offer valuable insights into the sentiment conveyed in text documents. These approaches rely on predefined lexicons or sentiment dictionaries which consist of lists of words and phrases commonly associated with positive or negative sentiments. Lexicons assign scores or labels to tokens (words) indicating their positive, negative, or neutral nature. There are several subcategories within lexicon-based approaches, including corpus-based, dictionary-based, and manual approaches.

Corpus-based approaches leverage semantic and syntactic patterns present in text corpora to determine the emotional polarity expressed in a sentence or document level. These approaches can be further classified into two subcategories, namely semantic and statistical. The semantic approach (also called ontology-based approach) involves calculating the similarity between tokens and assigning the same sentiment scores to the semantically close words or synonyms, relying on rules and sentiment dictionaries. The statistical approach, on the other hand, identifies seed opinion words or co-occurrence patterns statistically to determine sentiment orientation. The underlying principle is that if certain words occur more frequently in positive contexts compared to negative contexts (or vice versa), they are likely to have a corresponding sentiment orientation.

Dictionary-based approaches assume that synonymous words have the same sentiment polarities, while antonyms have opposite polarities, and that positive (negative) adjectives appear more frequently near a positive (negative) seed word. Initial seed words with predefined orientations are manually collected, and the list expands iteratively by searching for synonyms and antonyms from dictionaries, such as the well-known WordNet [6, 7], forming a lexicon.

Manual approaches involve human intervention in annotating sentiment lexicons. This process includes generating a list of sentiment-bearing words and assigning sentiment labels to these words. While laborious, costly, and time-consuming, manual approaches can yield consistent and reliable sentiment lexicons.

Hybrid approaches These sentiment analysis techniques combine the strengths of machine learning and lexicon-based techniques to enhance classification performance. By integrating these approaches, researchers aim to leverage the high accuracy and flexibility of machine learning methods along with the throughput and stability of lexicon-based approaches.

2.1.2.2 Use Cases

In this section, we will discuss the different use cases of sentiment analysis and how it can be used to improve business processes, customer satisfaction, and brand reputation.

One of the most common use cases of sentiment analysis is spam detection. By analyzing the content of emails and messages, sentiment analysis algorithms can detect spam and prevent it from reaching users. This is an essential function for email providers, social media platforms, and other online services.

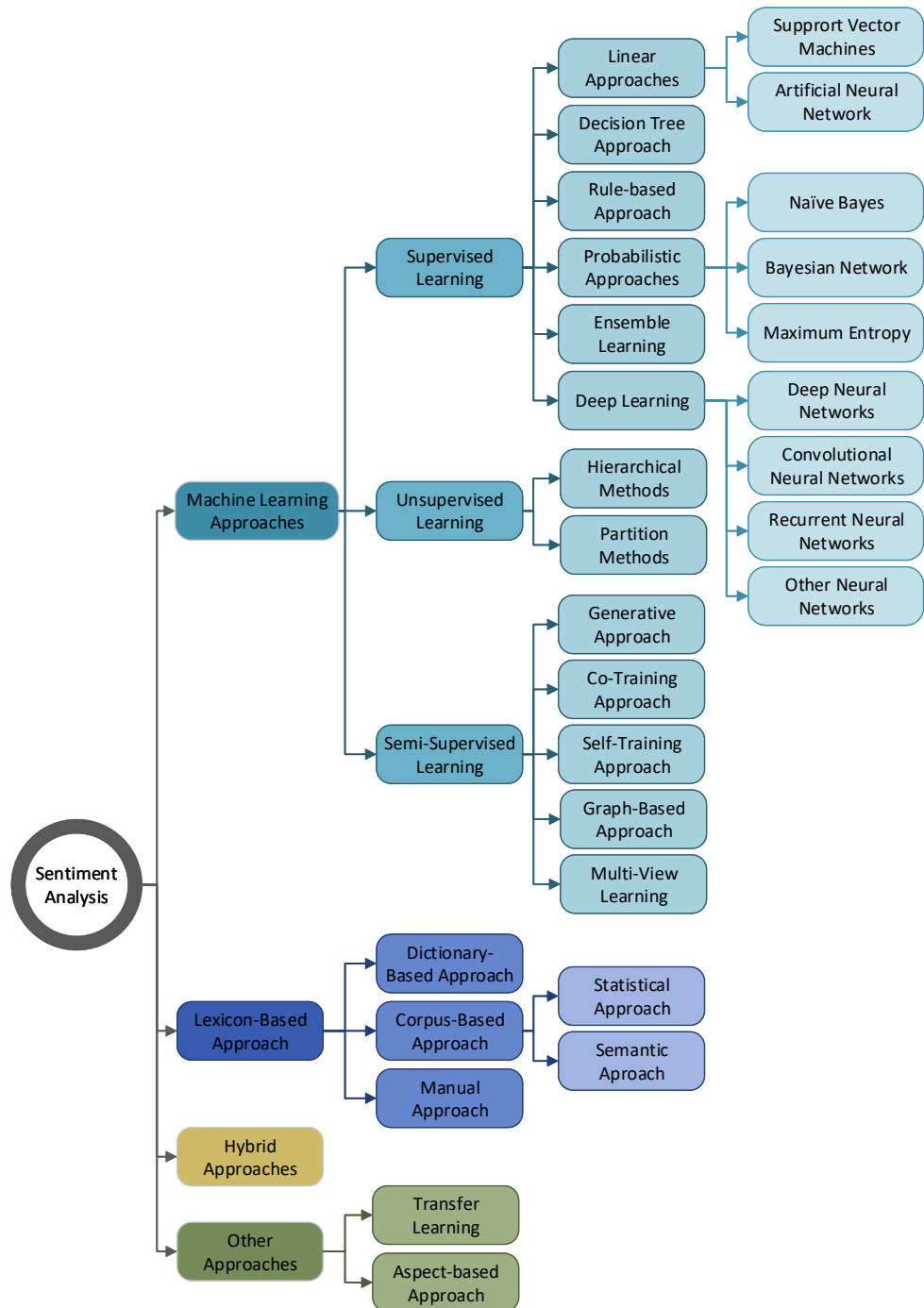


Figure 2.1: Sentiment analysis approaches

Another use case for sentiment analysis is machine translation. By understanding the sentiment of a text, machine translation algorithms can provide more accurate translations, as they can take into account the intended emotion of the original text.

Virtual agents and chatbots are becoming increasingly popular in customer service. By integrating sentiment analysis algorithms into these systems, businesses can improve the quality of their customer interactions. Sentiment analysis can be used to detect frustration, anger, or other negative emotions, allowing the system to provide appropriate responses and solutions.

Text summarization is another area where sentiment analysis can be useful. By analyzing the sentiment of a large volume of text, SA algorithms can provide a summary that captures the overall sentiment of the text.

Customer feedback is an important source of information for businesses. Sentiment analysis can be used to analyze customer feedback and gain insights into customer preferences and opinions. This information can be used to improve products and services and make the business more responsive to customer needs. Also, by analyzing customer interactions with customer service representatives, sentiment analysis algorithms can detect problems and improve customer service and product offerings.

Brand intelligence is another important use case for sentiment analysis. By monitoring social media and other online platforms, sentiment analysis algorithms can provide a real-time snapshot of brand reputation. By quickly detecting negative sentiment, businesses can take steps to address the underlying issues and improve their brand reputation. By providing insights into customer needs, preferences, and opinions, sentiment analysis can help businesses improve customer satisfaction, product offerings, and brand reputation.

2.2 String Matching

String pattern matching is the process of finding one or more occurrences of a string, known as patterns or signatures, within a larger text or string. It has a wide range of applications, including DNA sequencing, data compression, text processing, and natural language processing. The goal of string pattern matching algorithms is to efficiently and accurately locate patterns within large text datasets. A widely used approach of pattern matching is using finite alphabets. In this technique, consider searching a string pattern P , consisting of $p_1p_2\dots p_n$ characters, within a text T , of $t_1t_2\dots t_m$ characters, then, both the pattern and text sequences can be represented as a finite set of characters, denoted by A .

The classification of string pattern matching algorithms can be based on various criteria, such as the number of patterns to be matched, the type of pattern, and the computational model used. Focusing on the pattern number, these algorithms can be divided into two main categories, (i) single-pattern, and (ii) multi-pattern.

2.2.1 Single-pattern

Single-pattern algorithms are used to search one pattern against a given text. The simplest algorithm in this category is the naive string search algorithm, which does not require any data pre-processing. However, this algorithm has a time complexity of $O(mn)$, where "m" is the length of the pattern and "n" is the length of the text, and can be very slow and inefficient for large texts or complex patterns.

On the other hand, the Knuth-Pratt-Morris algorithm uses a partial-match table that is built by preprocessing each pattern individually and reduces the number of comparisons required by exploiting the properties of the target pattern. This table determines the number of positions to shift the pattern to the right, based on where a mismatch occurs. The Knuth-Pratt-Morris algorithm has a worst-case time complexity of $O(m+n)$.

The Boyer-Moore (BM) algorithm also locates string patterns in a text with mismatches or errors. The algorithm initiates the matching process from the last character of the pattern and in case of a mismatch, skips a portion of the input. The BM algorithm has a worst-case time complexity of $O(mn)$ but is highly efficient for large datasets. Also, Boyer-Moore-Horspool is an improved version of the Boyer-Moore algorithm that requires less memory without affecting the latency performance.

A probabilistic approach to string pattern matching is the Karp-Rabin algorithm which uses hashing to detect a single pattern in a group of strings within a text and is highly efficient for datasets with a small alphabet size.

In addition, the Baeza-Yates-Gonnet (bitap) algorithm, uses bit-wise operations to facilitate approximate string matching and determines whether a given input text contains a substring that is approximately equal to a pattern, where "equality" is defined by the Levenshtein distance. This algorithm pre-processes a set of bit-masks and uses bit-wise operations, which significantly improve the algorithm's performance.

2.2.2 Multi-pattern

When there is a need to search a finite number of patterns in a given text, executing the single-pattern matching algorithms separately for each pattern is extremely inefficient. Therefore, multi-pattern matching algorithms have been developed to address this issue and provide better scalability. One such algorithm is the Karp-Rabin which has a worst-case time complexity of $O(nm)$ in space of $O(p)$, where "m" is the length of text T and "n" is the length of each pattern. A notable algorithm is the Aho-Corasick string matching algorithm, which matches all patterns simultaneously and has a worst-time complexity of $O(n+m)$ in space $O(m)$. Additionally, the Commentz-Walter algorithm combines the basic principles of the Aho-Corasick algorithm with the fast matching of the Boyer-Moore string search algorithm. Finally, Wu and Manber proposed another multi-pattern searching algorithm, which is implemented as part of the `agrep` tool.

2.2.3 Other Classifications

String matching algorithms can be further categorized based on other features such as the preprocessing phase and the matching strategy. Preprocessing is conducted to achieve faster searching by preparing either the pattern or the input text, or both. There are four categories based on preprocessing, (i) naive or elementary, (ii) finite state automata-based, (iii) indexing-based and (iv) finite state automata and indexing-based. In regards to the matching strategy, the algorithms are grouped into (i) prefix-matching (e.g., Aho-Corasick), (ii) suffix-matching (e.g., Boyer-Moore), (iii) best-factor-matching, and (iv) others (e.g., Karp-Rabin, naive pattern matching).

The naive string search algorithm is the simplest approach to locating a pattern within a string, but it is also the most inefficient and resource-consuming. In this algorithm, the search process starts by looking for the pattern at the first character of the input string, and if it is not found, the algorithm proceeds to the next character until the pattern is located. The complexity of the naive string search algorithm is $O(nm)$, where "n" is the length of the pattern and "m" is the length of the input string. In the average case, it requires $O(n + m)$ steps.

To avoid backtracking, a Deterministic Finite Automaton (DFA) can be created. However, this process is costly as it requires the use of the Powerset Construction technique to convert a Non-Deterministic Finite Automaton (NFA) into a DFA. Therefore, this approach is typically used only when the DFA needs to be generated infrequently.

Index-based search requires preprocessing of the input text. In this approach, a sub-string index is built using structures like a suffix tree, a suffix array, or a compressed suffix array, to allow the fast discovery of occurrences of a string pattern. For instance, building a suffix tree takes time, but locating all occurrences of a string pattern takes $O(k)$ time assuming a constant size for the alphabet and that all inner nodes in the tree know their child leaves.

Exact string matching is the approach of locating the occurrence of a simple string pattern within another string. The exact string matching problem is to find all substrings in the input text that are exactly the same as the pattern. For example, the pattern matches exactly the string inside an input text.

2.2.4 The Aho-Corasick Algorithm

One of the most popular algorithms in the field for multiple simple string pattern matching is the Aho-Corasick algorithm, which was introduced by Alfred V. Aho and Margaret J. Corasick in 1975. The Aho-Corasick algorithm is considered one of the most efficient algorithms for multiple pattern searching as it matches all signatures simultaneously. To achieve this simultaneous matching, the Aho-Corasick algorithm pre-processes the set of patterns to build an automaton that is used during the matching phase, where each character of the text is processed only once. So, the algorithm's processing time does not explicitly depend on the number of patterns.

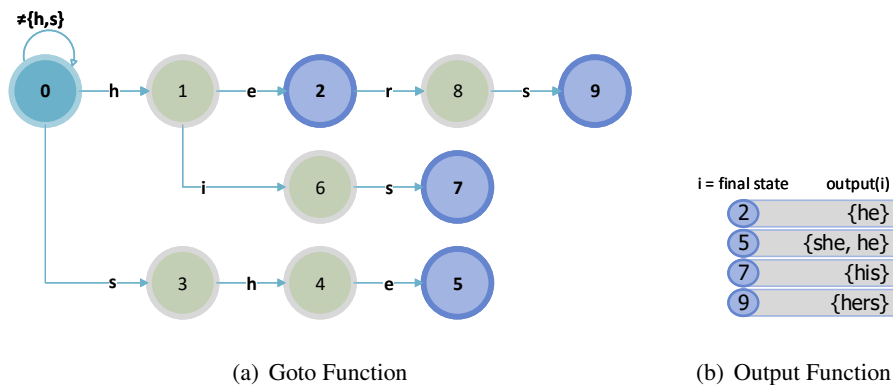


Figure 2.2: The Aho-Corasick functions using the pattern set $\{she, he, hers, his\}$.

Assuming we want to search for a set of patterns $P = p_1p_2\dots p_n$ inside a text $T = t_1t_2\dots t_m$, both sequences of characters form a finite character set Σ , then the complexity of the algorithm is linear to the pattern length ν , plus the length of the given text μ , plus the number of output matches.

The algorithm constructs a pattern matching machine that matches all patterns in the text at once, one byte at a time. The machine has three distinct functions, (i) a goto function, (ii) a failure function, and (iii) an output function. The goto function determines if a state transition can be performed based on the current state and the ASCII value of the input character. If the input character matches one of the transitions starting from the current state, then the state pointed to by this transition becomes the next state. Otherwise, the next state is resolved by the failure function.

The failure function either drives a transition to one or more intermediate states or to the initial state, represented with 0 in the goto graph. After each state transition, the algorithm checks the output function to determine if the pattern matches a sub-string of the text T . This procedure continues and terminates with the end of the input text T .

The produced automaton is non-deterministic (NFA) because failed transitions may not consume any input. Additionally, the failure function can result in numerous state transitions for a single input character. Therefore, the matching operation might require the exploration of multiple paths before the actual match of a pattern.

To avoid performance loss when the patterns' sizes are large, a revised version of the traditional Aho-Corasick algorithm exists, which replaces all failure transitions. The new automaton that is produced is a DFA and provides one transition per state and input character. Although this approach requires more memory than the previous one, it is more efficient in terms of processing throughput, and the achieved complexity of this approach is $O(n)$. A graphical representation of the algorithm's goto and output functions is presented in Figure 2.2.

2.3 Parallel Computing

Parallel computing is a technique of solving large computational problems by dividing them into smaller tasks that can be executed simultaneously on multiple processors or cores. This approach is widely used in high-performance computing for many years. Recently, interest in parallel computing has increased due to physical limitations that prevent frequency scaling. The hardware architectures supporting parallel computing can be classified according to Flynn's Taxonomy, based on the number of concurrent instructions and data streams available. This design concept has enabled significant advancements in fields such as scientific research, data analysis, and artificial intelligence, making parallel computing a critical tool for modern computing.

2.3.1 Types of Parallelism

Bit-level parallelism This type of parallelism is based on the increasing word size of the processor. It reduces the number of instructions needed to be executed by the processor when performing a task on variables with sizes greater than the length of the word. For example, in a scenario where a 16-bit processor needs to add two 32-bit integers, it must first add the 16 low bits of each integer and then the remaining 16 high bits. This process requires two instructions to complete, while a 32-bit processor can perform it with one.

Instruction-level parallelism (ILP) ILP is a standard of how many instructions can be executed simultaneously in a process. These instructions can be re-ordered and grouped when executed concurrently without affecting the result of the program. The first way this execution can be achieved is with dynamic parallelism, supported by the hardware, where the processor decides at runtime which instructions to execute in parallel. The other approach is through software with static parallelism, where the compiler decides a priori the set of instructions that are going to be simultaneously executed. Some common techniques used to exploit ILP are instruction pipelining, superscalar execution, out-of-order execution, and branch prediction.

Task-level parallelism (TLP) Multiprocessor systems can execute different processes (threads) by assigning them to the various processors. Each process can execute on different or the same datasets as the others. This level of parallelism is called task-level parallelism. TLP generally falls into either the form of running various processing steps of the algorithm as different (communicating) tasks/threads or the Single Program Multiple Data style (SPMD).

Data-level parallelism (DLP) The concept of data parallelism is fundamentally different from the levels of parallelism described above. Data parallelism is achieved when the same function is simultaneously executed on multiple cores across different data. On the other hand, task-level parallelism is the simultaneous execution of different functions on multiple cores across the same or different data. In most cases, when data parallelism is applied, the different threads control the operations on all data elements, while in other cases a single execution thread controls the operations. However, in both designs, all threads execute the same code

2.3.2 Flynn's Taxonomy

In 1972, Michael J. Flynn developed a methodology to clarify the types of parallel computer architectures based on their instruction set interaction with data streams, the number of concurrent instructions, and data streams present in the architecture. The taxonomy uses the stream concept to categorize the instruction set architecture. A stream is simply a sequence of objects or actions. There are both instruction streams and data streams, and there are four simple combinations that describe the most familiar parallel architectures. SISD (Single Instruction stream, Single Data stream) is the simplest category and refers to a uni-core processor that executes a single instruction stream on a single data stream. A SIMD (Single Instruction stream, Multiple Data streams) architecture consists of multiple processing elements executing the same operation on multiple data streams, categorized as array processors, pipelined processors, or associative processors. The MISD (Multiple Instruction streams, Single Data stream) architecture operates multiple processing elements on a single data stream, typically used for fault tolerance. MIMD (Multiple Instruction streams, Multiple Data streams) is the class where multiple autonomous processing elements execute different instructions on multiple data streams. A graphical representation of these architecture categories is presented in Figure 2.3.

Other classifications such as SIMT (Single Instruction stream, Multiple Threads), SPMD (Single Program, Multiple Data streams), and MPMD (Multiple Programs, Multiple Data streams) have been introduced to cover modern architectures, with SIMT combining SIMD with multithreading, SPMD executing the same program on multiple data streams, and MPMD executing at least two independent processes on multiple data streams with a manager and processing units.

2.3.3 Hardware for Parallelism

In recent years, the development of high-performance computing technologies, such as reconfigurable computing with FPGAs, general-purpose computing on GPUs (GPGPUs), and application-specific integrated circuits (ASICs), has significantly expanded the capabilities of parallel computing. These technologies provide specialized hardware that can perform certain types of computations much faster than general-purpose CPUs, allowing for even faster and more efficient parallel computing systems.

The Central Processing Unit (CPU) is the primary and vital processing unit of modern computers and is responsible for executing instructions and managing the flow of data in a system. With the rise of multi-core CPUs, their capabilities have been extended to support parallel computing models. Processors, provided by the majority of vendors, offer extended instruction sets with support for hyper-threading and vector operations. Multi-core CPUs can be utilized to concurrently execute various independent, or co-operating, processes with high execution bandwidth. Since modern CPUs allow parallelism in both instruction and task levels, as well as some level of data parallelism, they are ideal for a vast variety of workloads. In comparison to GPUs, CPUs are regarded as more efficient for branch-intensive workloads since they do not follow the lock-step model.

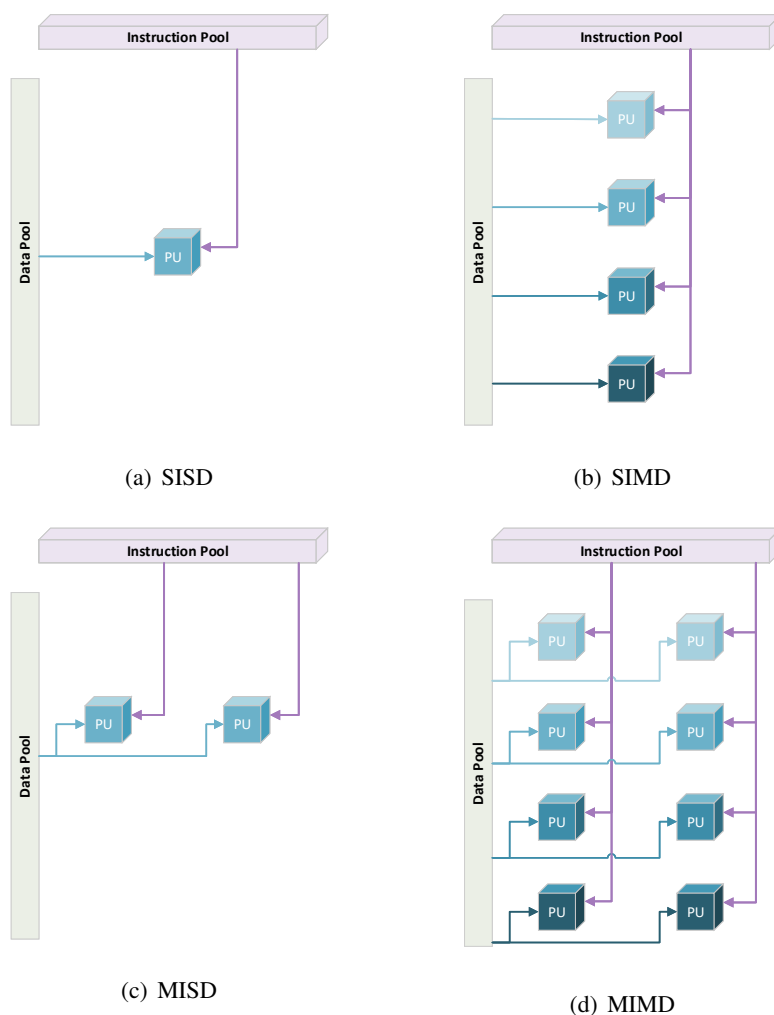


Figure 2.3: Parallel architectures according to Flynn's taxonomy.

Graphics Processing Units (GPUs) were originally designed to handle the massive amounts of data that are required for rendering graphics. However, modern GPUs are capable of handling massively parallel computations, making them suitable for compute-intensive applications that require high memory access bandwidth. GPGPU architectures are based on a set of multiprocessors, each containing multiple streaming processors that operate according to the Single Instruction Multiple Thread (SIMT) model, allowing them to execute thousands of threads simultaneously. This architecture enables GPUs to be highly efficient for parallel applications.

The tasks sent from the host computer to the connected GPU are referred to as kernels and each one of them is launched on the GPU as a set of hundreds or thousands of threads organized in thread blocks. These thread blocks are executed in a SIMT architecture by

the multiprocessors, which are organized into same-sized groups called wraps. A typical kernel execution involves four steps, (i) transferring data from host memory to GPU memory using a DMA controller, (ii) launching the kernel on the GPU, (iii) executing the threads in parallel, and, (iv) transferring the resulting data back to the host memory using the DMA controller.

The GPU memory organization typically includes multiple memory spaces, including a shared memory space for each thread block that is accessible to all threads within the block and has the same lifetime as the block itself. Each thread in a block has its own dedicated local memory. Moreover, there are three extra memory spaces named *global*, *constant*, and *texture*, that can be accessed by all threads and persist across kernel launches initiated by the same process.

In addition to high-end discrete GPUs, there are also integrated GPUs (iGPUs) that are integrated on the same die as the CPU. Integrated GPUs do not have dedicated memory and share the same physical address space as the CPU. Although integrated GPUs perform better with workloads bound to the I/O interface, their computational capacity is limited by the internal power control unit to avoid exceeding the thermal constraints of the processor die.

2.3.4 Frameworks for Parallel Computing

The availability of powerful multi-core processors and off-the-shelf GPUs that can be used to build highly efficient and parallel applications and systems made parallel designs and architectures to be increasingly popular in the field of general-purpose computing.

Off-the-shelf GPUs, such as NVIDIA GeForce, AMD Radeon, and Intel Iris Pro offer high performance and efficient power consumption for parallel computing tasks in modern commodity systems. They use programming models such as CUDA and OpenCL to enable efficient parallel execution and are widely used in scientific simulations, industrial applications, and academic research.

CUDA is a parallel computing platform and application programming interface that was developed by NVIDIA. It enables developers to program GPUs for general-purpose processing, beyond just graphics rendering. The CUDA platform provides direct access to the GPU's virtual instruction set and various parallel computational elements, allowing for the execution of compute kernels - units of work issued by the host computer to the GPU.

OpenCL, on the other hand, is a framework that enables the uniform programming of heterogeneous platforms, which may include CPUs, GPGPUs, DSPs, FPGAs, and other types of processors and hardware accelerators such as the Intel Xeon Phi-coprocessor. OpenCL provides a standard interface for parallel computing, enabling task-based and data-based parallelism through the execution of compute kernels.

Chapter 3

Design & Implementation

In this chapter, we provide a detailed description of our sentiment analysis tool’s design and implementation. The tool is based on Vader’s [8] sentiment analysis logic but is completely re-designed and re-implemented in C/OpenCL, able to utilize multi-core processors, such as GPUs, iGPUs and CPUs, in contrast to Vader’s Python-based single threaded design. Also, the system includes a data pre-processor able to prepare structured Twitter datasets for sentiment analysis. Finally, our work provides several lexicon and logic optimizations over Vader’s initial design that boost its performance by re-ordering tasks, avoiding to perform the same task twice, and most significantly, allowing us to utilize multi-core processors.

3.1 Architecture

Our sentiment analysis tool is able to operate on all external GPUs, iGPUs and CPUs supporting the execution of OpenCL applications [9], such as CPUs offered by Intel and AMD, as well as GPUs provided by NVIDIA, AMD, and Intel. Our system is composed of four modules and its core engine is developed using OpenCL. The first module is responsible for analyzing the input text and pre-processing it accordingly in case it is a well-structured Twitter dataset, preparing it for the sentiment analysis process. The second module is tasked with handling data transfers to and from the discrete GPU when operating on external hardware, or the memory mappings to the CPU and iGPU memory spaces that OpenCL utilizes, when a CPU or iGPU is used. To cover the most complex case, we assume, for the rest of this chapter, that an external GPU is utilized. The third module is completely OpenCL-based and performs string matching on the input data using our modified Vader’s lexicon as the pattern set. Thus, this module is responsible for assigning the initial scores to the input’s words using the selected multi-core processor. The fourth module is responsible for performing the fine-tuned sentiment analysis and compiling the results. Our system can be compiled as a standalone application as well as a single library, exposing a unified API, and enabling it to be embedded in other applications. An overview of our sentiment analysis tool architecture is presented in Figure

3.1.

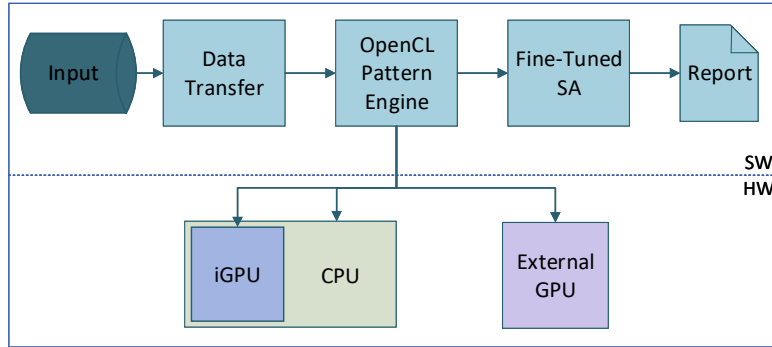


Figure 3.1: Architecture overview.

3.2 Sentiment Analysis Life-cycle

To perform sentiment analysis, our tool requires two input files, (i) a lexicon file containing words or text patterns along with their corresponding polarity weights and, (ii) a file containing the text whose content we want to determine in terms of the polarity of emotions. The first file is used as the pattern set of the OpenCL-based string matching engine and is compiled as an Aho-Corasick DFA during the system's initialization phase. If the data file, subject to sentiment analysis, contains unstructured text that has to be analyzed as a whole or is partially structured, (i.e., text chunks separated with empty lines) it can be processed as is. However, if the input file contains data in a well-structured format, containing various labels and metadata, (i.e., a Twitter dataset), it has to first be pre-processed by the input parsing module so that only the actual text data are selected for sentiment analysis.

Once the lexicon DFA is compiled and the input data are pre-processed, the system generates the data structures (`tweet table`) that hold various information per text entry (such as scores, tokenized words, punctuation information, etc.). Then each tweet enters a data buffer and it's simultaneously parsed to remove punctuation and identify each word, updating the `tweet table`. When the process of tokenizing each text entry is complete, the sentiment analysis process begins by preparing batches of tweets to be processed by the multi-core processor.

The batches are prepared during the tokenization phase where the text is stored in the data buffer along with various metadata about each text's entry point and size. Once a buffer is full, it is moved to the device where our OpenCL-based pattern matching module identifies which input words match the lexicon entries. The results are then moved back to the host's memory and each word in the `tweet table` is assigned to the score of each respective matched pattern.

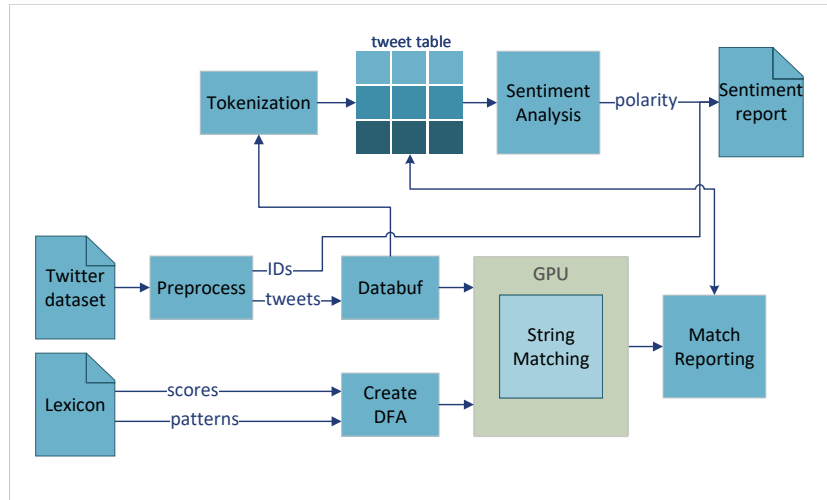


Figure 3.2: Sentiment analysis life-cycle overview.

Finally, the fine-tuned sentiment analysis is performed on the scored words, providing the *positive*, *negative*, *neutral* and *compound* scores for each text entry. These results are then used to compile the final sentiment analysis score report where each line in the file contains the tweet’s ID along with the calculated scores. An overview of the sentiment analysis life-cycle is presented in Figure 3.2.

3.3 DFA Construction

Our system performs the initial sentiment score assignment to every input word based on our modified version of Vader’s lexicon. In contrast to Vader’s original implementation, where every input word is matched against the lexicon, one at a time for every tweet in a serialized fashion, using naive string matching, our tool utilizes our highly optimized and data-parallelized version of the Aho-Corasick pattern matching algorithm, presented in Section 2.2.4. To utilize this efficient multi-pattern matching algorithm, we first have to compile the sentiment lexicon into a DFA suitable for parallel execution with multi-core processors, using the following five steps available via our API.

Reading the Lexicon File The process of reading the lexicon file is executed within the `main()` function by calling `sa_ctx_init_new()`, responsible for initializing a new sentiment analysis context. This function calls `load_patterns_and_scores()` which is responsible for reading the sentiment lexicon file and extracting the patterns and their corresponding scores. For each line read, these two elements are stored as arrays in the newly initialized sentiment analysis context (`sa_ctx->patterns` and `sa_ctx->scores`), while each pattern is also written to a new file (`patterns.txt`) which can later be used to perform other tasks. By generating a new context with each lexicon file, we are able to preserve and utilize various lexicons simultaneously (e.g., lex-

icons optimized for specific types of datasets, fine-grained lexicons for further analysis, or lexicons serving various languages).

Score Table Creation In our implementation, we utilize two arrays to store the patterns and their corresponding scores. These arrays are cell-to-cell aligned, which means that each cell in the patterns table is directly linked to the corresponding cell in the scores table. Therefore, given a pattern ID, we can easily access both the pattern and its corresponding score by using the same index value in their respective tables and perform these traversals with simple pointer arithmetics when necessary.

DFA Compilation To utilize the Aho-Corasick algorithm, we first have to compile every pattern into an NFA. Then, the NFA is transformed into a DFA which represents the Aho-Corasick state machine as a tree of nodes, with each node containing information about the state, its valid and invalid transitions, and information about the corresponding pattern. Traversal through this state machine is performed by following the appropriate pointers in each node, based on the input.

In our implementation, the patterns are read from the sentiment analysis context, constructed as described above, with each index in the pattern array representing the corresponding pattern ID. The pattern is then added to the list of patterns for the state machine, and they are all compiled into an NFA and transformed into a single DFA using the Aho-Corasick state machine compilation function `acsm_compile()`.

DFA Serialization To create an implementation of the Aho-Corasick algorithm suitable for parallel computing, we use a variation of the standard algorithm which represents the pattern state machine (DFA) as a serialization of the automaton tree to a single-dimensional integer array. This allows for efficient storage and processing of the DFA and enables easy traversal of the state machine using simple arithmetics. This array allows all states to be consecutively addressed and pre-fetched by the OpenCL-based pattern matching engine instead of following pointers to each state node, scattered across the memory.

States \ ASCII	0	...	101 (e)	...	104 (h)	105 (i)	...	114 (r)	115 (s)	...	255
0	0	0	0	0	1	0	0	0	3	0	0
1	0	0	-2	0	1	6	0	0	3	0	0
2	0	0	0	0	1	0	0	8	3	0	0
3	0	0	0	0	4	0	0	0	3	0	0
4	0	0	-5	0	1	6	0	0	3	0	0
5	0	0	0	0	1	0	0	8	3	0	0
6	0	0	0	0	1	0	0	6	-7	0	0
7	0	0	0	0	4	0	0	0	3	0	0
8	0	0	0	0	1	0	0	0	-9	0	0
9	0	0	0	0	4	0	0	0	3	0	0

Figure 3.3: Serialized Aho-Corasick DFA represented as a two-dimensional integer array with negative values indicating final states.

After the state machine is compiled, a serialized state table gets generated that will be transferred to the device (GPU) using the `acsm_gen_state_table()` function. The size of the allocated memory for both the host and device is determined by the number of states in the Aho-Corasick state machine, and the size of the ASCII set (256). The function loops through each state and its transitions and when the current state is a final state, it negates its value. By marking the final states as negatives, we can store this crucial information without requiring extra memory or indexing extra fields.

For a better understanding of this process, we illustrate in Figure 3.3 the serialized DFA derived by compiling the patterns *she*, *hers*, *he*, *his* as a two-dimensional integer array. The corresponding state machine is illustrated as a tree in Figure 2.2. This array consists of 256 columns (as many as the ASCII set) and rows equal to the number of states. Each cell contains the number of the next valid transition, corresponding to the ASCII character that the column represents, while the final states are denoted with a negative sign. In reality, this table is single-dimensional with each state row concatenated after its previous.

Move the DFA to the Device Once the DFA is serialized, it is moved to the multi-core device (e.g., GPU) where it is permanently stored for the current session. This process is performed only once per context and the DFA does not have to be moved to and from the device for each input file we wish to process. Each GPU-loaded DFA can then be addressed by the device during the pattern matching process using its corresponding context ID. A copy of the serialized DFA is also kept in host memory in case the user wishes to load and unload DFAs from the GPU for memory optimization reasons without having to recompile the state machine each time. Also, the previously generated and memory consuming NFAs and DFAs can be freed from the host memory to release the resources.

3.4 Data Pre-processing

Except for processing unstructured text files, our tool is also able to analyze huge Twitter datasets collected using the Twitter API. These datasets are well structured and contain a lot of text-based tags and special characters to wrap each tweet and supply its meta-data. Although the Twitter-extracted data files include a wide range of entities, as described in table 3.1, which can be useful for further analysis, we only need to extract the *"id"* and *"text"* fields for the sentiment analysis task. To pre-process the data, we develop a Tweet parser, which extracts the necessary information from the files obtained with the Twitter API.

The initial step is to select tweets written only in English since the lexicon that we utilize has only English words. Additionally, we keep the tweet *"id"* entity to facilitate the differentiation between all posts based on their unique identifiers. The text part is extracted depending on the type of each tweet (i.e., extended-tweet, full-text, retweeted-status, quoted-status) where the actual text can be in a different field in each case. We also handle retweets and quoted tweets, by merging retweeted text with the original text when necessary. For retweets, a portion of the last part of the text might be missing and

Table 3.1: Metadata provided with each tweet in datasets retrieved via the Twitter API.

Field	Description
created_at	ISO date of tweet action
id	Tweet object identification number
user_id	User profile ID
text	Raw tweet text
truncated	Boolean value indicating if the text is reduced by Twitter
entities	Dictionary that contains semantics of tweet like hashtags, special symbols, mentioned users and URL links
metadata	List of metadata such as ISO language code
source	Header of the application used for tweet posting
in_reply_to_status_id in_reply_to_user_id in_reply_to_screen_name	When the tweet is a reply for another post, this field contains the ID of the replied post, the ID of the user who created the original tweet and the screen_name
geo coordinates place	User geolocation data
retweeted_status	Used only when the tweet is actually a retweet . This element contain all tweet information of original tweet
is_quote_status	Boolean value that shows if the tweet is actually quote to another post
retweet_count:	Number of retweets for this post at the moment of collection
favorite_count	Number of likes of the post at the moment of collection
lang	Two letter code of text language used when the tweet is written in a single a language

is converted to ' ' from the API due to its intersection with the original text. In that case, retweeted text merges with the original text.

As for every OSN, the language on Twitter contains a lot of punctuation, emojis, slang words, and other noisy information. Therefore, we remove the *author mention* ("RT @Author" in case of retweets), new lines, emojis, Unicode symbols, non-ASCII characters and URLs from the text as they do not provide meaningful sentiment. The objective of this parser is to generate a new input for the sentiment analysis tool that exclusively includes the unique *ID* and corresponding *text* for each tweet post. This process is repeated across all files in a Twitter dataset. A list of metadata accompanying each tweet in a Twitter dataset along with the respective label descriptions is provided in Table 3.1

It is worth noting that pre-processing plays an important role in the sentiment precision of our system. By removing irrelevant information from the text and extracting only the necessary data points, we improve the accuracy of sentiment analysis results.

3.5 Parsing and Tokenizing the Input

Once the pre-processing phase is completed, the system calls the `load_tweets()` function that receives as input the pre-processed tweets and separates their ID and the filtered text, updating the relevant data structures. The next step is to process the tweets and identify each word by removing punctuation or other noise in the text. This is done with the `split_tweet()` function which takes as input the string representing a tweet, the text's length and a pointer to the `words_per_tweet` table that stores information about the words in the tweet and updates the table by parsing the tweet and splitting it into individual words. The function iterates over each character in the tweet and utilizes several counters and flags for checking the type of characters in the tweet to split the sentence at the proper point, so each token only contains the word itself without any unnecessary punctuation. For each character, the function checks:

- Whether it is an uppercase letter and increments the counter of uppercase letters in the word.
- Whether the previous and following characters are alphabetic or numeric and keeps a flag for each case.
- If the current character is an exclamation point "!" or question mark "?" and increments the corresponding counter to calculate the emphasis on punctuation later. Also, it checks if the preceding character is alphabetic or numeric so it will act as a delimiter indicating the end of the word. Similarly, if the character found after a question mark or exclamation point is alphabetic or numeric it will act as a delimiter and that denotes the beginning of a new word.
- If the current character is a "#" to determine if a hashtag begins, since we are working on Twitter texts. In that case, this indicates the beginning of a word, but we have to skip the first character and assign the start offset of that word with the position of the next character.
- If the current character is "-" and if there is a letter before and a letter/number after the symbol so the "-" symbol is not stripped (e.g., keep the word "pre-processing" as a single word).
- The delimiters that indicate the end of a word are: space " ", comma ",", full stop ".", colon ":" and in some cases the exclamation point "!" or question mark "?" as described above. In any occurrence of these symbols, the function updates the tweet table by setting the end position and length of the current word, adding the word to the word list, updating the length of that list, and checking whether the word is in all capital letters. If the next character is alphabetic or numeric, the function sets the start position of the next word.
- We finish tweet splitting by calculating the emphasis on punctuation based on the question marks and exclamation points found. Also, we determine if the text contains only words with all capital letters or not, since this information is necessary

for the sentiment analysis functions.

The structure of the table used to store the information about individual tweets is presented in 3.4. Each entry stores a single tweet and contains the following information:

1. **Word Counter:** This counter keeps track of the number of words found in the tweet.
2. **Punctuation Counters:** There are two counters that capture the number of question marks and exclamation points present in the tweet. They provide insights into the overall sentiment expressed through punctuation.
3. **"But" Flag and Position:** This flag indicates whether the word "but" is used in the tweet. Additionally, it records the position of the word "but" within the text, allowing us to assess its impact on sentiment as it is used to introduce something contrasting with what has already been mentioned.
4. **Sub-tables:** The table incorporates 10 sub-tables, where each sub-table holds various values and information about the words in the tweet, with each cell assigned to each word. For instance, the sub-table `words[i]` contains the words of the current tweet as identified during the tokenization phase described above, with "i" representing the word's index as read from left to right. Another important sub-table is `scores[i]` which stores the initial score assigned to each word based on its match with entries in the lexicon, during the OpenCL-based string matching phase. If a word matches a lexicon entry, its corresponding score is stored in this sub-table, otherwise, it remains at 0. The `senti_score[i]` is filled with values by the sentiment analysis module and represents the final sentiment score assigned to each word. We utilize the `score[i]` value as the initial valence and further fine-tune it based on our implementation of Vader's logic, refining the sentiment assessment, as we will describe in the following sections.

3.6 OpenCL-based Pattern Matching

To accelerate data processing, OpenCL and multi-core processors (e.g. GPUs) are used to match the input text against the sentiment lexicon and assign the initial scores to the input words. To achieve this, the system has to move the data to the device (GPU), scan the data for matches, and then return the results from the GPU to the host's memory.

3.6.1 Data Buffer

During the initialization phase, a data structure called `databuff`, is established with the purpose of carrying all the necessary data during the transfer between the device and the host. This structure involves various individual buffers to store the data, metadata and results for both the host and GPU. These buffers reserve memory in advance for the worst case of data volume to avoid the needless repetition of allocating and clearing the same memory addresses, while only the utilized space is transferred at each execution cycle to

word_count	but_check	but_index	words	pattern_id	starts	ends	length	scores	senti_scores	is_upper	is_booster	is_negation	qm_amps	ep_amps	punct_amps
0	0	0	w o r d 1	0	242	0	4	5	1.3	1	1	0	1	0	0
1	1	1	w o r d 2	1	6563	1	9	5	1.8	1	0	1	0	1	0
2	2	2	w o r d 3	2	7543	2	14	5	0.76	2	0	2	1	0	0
j	j	j	w o r d j	j	2878	j	78	-1.4	242	j	0	j	0	0	0

tweet 0

tweet 1

tweet i

j = word_count - 1

Figure 3.4: Tweet table overview.

optimize memory transfers. An identical empty `databuff` structure is allocated both on host and device memory.

The host stores the input data in the designated location (`h_data`) while also inserting the appropriate metadata to the auxiliary arrays such as the `h_indices`, that points to the beginning of each tweet, and `h_sizes` that indicates the size of each tweet during the text tokenization phase. The data array holds chunks of tweets written consecutively, therefore we can determine the current tweet by utilizing the `h_indices` array and by referencing the corresponding `h_sizes` array we can ascertain the endpoint of each tweet.

The process of appending data to the buffers continues until a batch is full or there is no more data to process. Once a batch of tweets (a complete `databuff`) is prepared, the utilized portion of the buffer is transferred to the GPU memory. When the processing unit is a discrete GPU, the data is transferred through the PCIe bus to the graphics card's DRAM. The `databuf_copy_host_to_device()` function is used to copy (transfer) the data buffer (and its metadata) from the host (CPU) to the equivalent buffer on the device (GPU). If a CPU or iGPU is utilized for processing the host's and device's buffers are memory mapped and there is no need to transfer the data as such kinds of processors have direct access to the host's DRAM.

Since the initialization of the bus for every small transfer results in suboptimal performance, batching multiple small transfers into larger sets is a significant improvement. Also, the SIMT and SIMD concepts that our pattern matching approach is based on, require performing the same task on multiple input data simultaneously. When the data

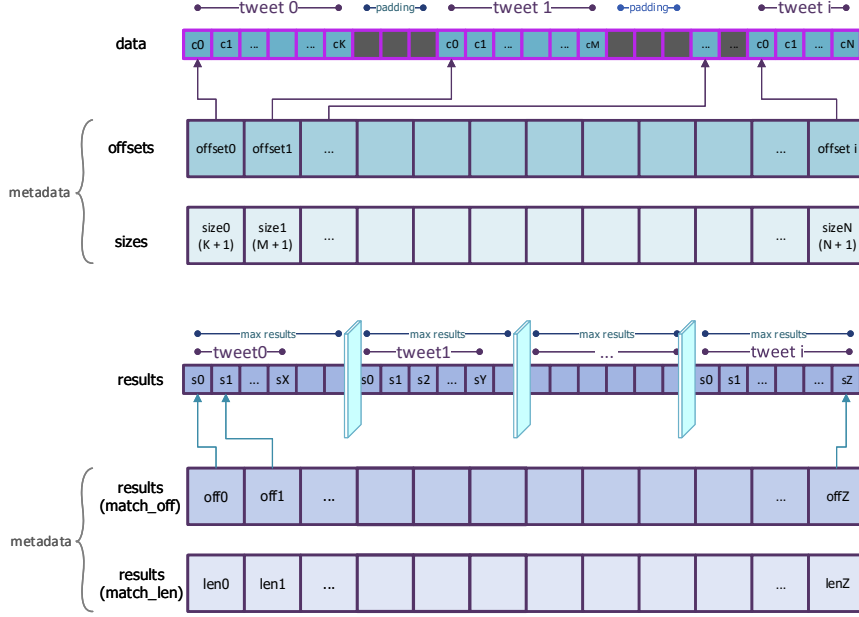


Figure 3.5: databuff structure overview.

is batched and transferred as a set, the batch can be scanned for matches and the main process that launched the processing task can construct the next batch, creating a pipeline between I/O and processing.

To achieve vectorization of memory access, we choose to not handle the input data as single-byte characters but to fetch and store them in `int4` type vectors. There is no need to pre-process the sizes and offsets of the input data. The batching of the input data into the buffer bucket is performed by storing the data bytes in the data array of the bucket. The size of each data chunk (tweet) is aligned to 16 bytes to ensure correct memory vectorization. A visual representation of the `databuff` structure is presented in Figure 3.5

3.6.2 Score Assignment

Once the buffer has been copied to the device, the data are scanned for matches against the serialized DFA derived by the lexicon, utilizing OpenCL, using the `ocl_aho_match()` function. The function spawns as many threads as the chunks (texts) in the buffer, where each thread utilizes the Aho-Corasick algorithm to locate every possible match with the lexicon with a single pass over the input text. Thus, the time required to process a buffer containing N tweets is the time required to process the largest tweet, bound to 240 bytes.

The search process begins at the initial state (state 0) and proceeds by selecting the appropriate column based on the ASCII value of the input character. The next valid state is

then determined by the value stored in the selected cell. This process continues until the end of the input string is reached or a final state is encountered. Final states are identified by a negative value and represent a match with one or more patterns in the lexicon.

Data processing is performed in a thread-per-data-chunk fashion, with each thread assigned a unique ID and a data chunk from the data buffer. The process of assigning data chunks to threads is done during the batching process, with the number of available GPU threads determining the number of data chunks that can be stored. Each thread fetches the size of the data it is assigned to process and its starting offset from the buffer's sizes and offsets arrays, respectively. The data is then fetched and processed using the `int4` vector data type, which pre-fetches 16 single-byte string characters at a time with a single memory read instruction. Data prefetching using vectorized memory accesses improves the performance of the string matching module by exploiting the SIMD characteristics of the underlying SIMT architecture. The complexity of the string matching algorithm is not increased to $O(n^2)$ due to nested loops used for data prefetching and vector unpacking.

When a match is found, the final state of the current match is stored in the `d_results` array which is preallocated in the GPU memory. Also, the position of the pattern within the tweet is saved in the `d_match_off` array, as well as the length of the pattern in the `d_match_len` array.

Finally, when the process of pattern matching is complete, the results are sent to the corresponding host memory using the `databuf_copy_device_to_host()` function. If the buffer is memory mapped (e.g., when using the CPU or iGPU), the system does not need to transfer the results. Upon completion of pattern matching on the GPU, the results are returned indicating the final states that were reached. However, some of these final states may not be valid matches and therefore should not be reported as such. To address this issue, additional checks are performed on the length and offset of the word that reached a final state. Specifically, a word is considered to be a match only if it has the same length as the pattern and if the position of the word's end coincides with the location where the final state was reached. Otherwise, the result represents a substring match, which yields an inaccurate score. If a valid match is detected, the score of the corresponding pattern in the lexicon is added to the appropriate cell in the tweet table, described above, for the matched word. Also, some flags for special word cases get initialized if they were matched.

This processing cycle of splitting the input, constructing the tweet table and data buffers, GPU-based pattern matching, and score assignment is repeated until the entire input is consumed, with each processed batch overwriting the previous one to preserve memory resources and avoid constant memory allocations.

3.7 Sentiment Analysis

After the initial scores per input word are calculated by the OpenCL-based engine, the sentiment analysis module calculates the fine-tuned sentiment scores for each word and

the overall text's score. The scores assigned during the match reporting process are updated according to our re-designed and re-implemented Vader's logic. In this section, we describe the role of each function involved in this process as well as our modifications and optimizations on Vader's initial design.

polarity() This function is the master function that performs the fine-tuned sentiment analysis on a set of input text (tweets in this example) and writes the results to the final sentiment score report. The rest of the functions described in the following paragraphs are part of the `polarity()` function. Each function is indicated with its call order and hierarchy within the `polarity()` function.

A. allcap_diff() This function is originally utilized by Vader to identify if some words are in all uppercase letters. However, we have moved this process to the tokenization phase described above. By taking advantage of the iterations that parse each tweet character-by-character we save a label for each word in the tweet table that declares if a word is spelled in all capital letters or not. Thus, when the function is called during the sentiment analysis process, it only has to return a pre-calculated value instead of re-iterating the entire input, which was a sub-optimal design of the vanilla Vader.

B. isUpper() We provide our own implementation of a function that takes a string and its length as input and checks if all the characters in the string are uppercase letters. The function iterates over each character in the string and checks if it belongs to the ASCII set of uppercase letters or if it is a digit following an uppercase letter (e.g., COVID-19). If it is, it increments the respective. At the end of the loop, if the counter is equal to the length of the string, it returns *true*, indicating that all characters are uppercase letters. Otherwise, it returns *false*. This function is very useful in text processing applications where it is necessary to determine if a word is in uppercase or not, since in our case uppercase words have more sentiment weight. This process is also performed only once during the tokenization phase, in contrast to Vader's vanilla implementation.

C. toLower() This simple function takes a string as input and converts all its characters to lowercase. The function iterates over each character in the input string and checks if it is within the ASCII range of uppercase letters (byte 65 to 90). If it is, it adds 32 to the ASCII value of the character to convert it to lowercase. In our case, this process is performed by the pattern matching engine, in contrast to Vader's original implementation, thus, all processes described above are performed with a single pass on the input data.

D. sentiment_valence() This function calculates the sentiment valence of a given tweet based on its words and their scores, as well as the presence of booster and negation words. The function takes as input a `words_per_tweet` structure containing a tweet represented as an array of words, along with information about each word, such as its score, based on the pattern matching and the lexicon, and flags indicating whether it is a booster or negation word, according to the special flags added in the lexicon. The function also takes as input the index of the current word being evaluated, `isCapdiff`, which indicates whether the sentence has a capitalization differential, and the value of `isUpper()`,

which indicates whether the current word is uppercase.

The function first initializes two float variables called "s" and "valence" to 0. These variables will be used to calculate the sentiment valence of the text. Then, it checks if the current word matches any special flag words, such as "kind" or "no", and assigns a predetermined valence to them. If the current word has matched a word in the lexicon and is not a booster or negation word, then the valence is kept to the score obtained by the pattern matching process. If the word is preceded by "no", the valence will be set to 0 or multiplied by -0.74.

If the current word is in all uppercase letters and the sentence has a capitalization differential, the valence score is adjusted by adding or subtracting 0.733 depending on the polarity of the valence at this point.

Next, the function iterates over the previous three words and checks if they are not in the regular lexicon. For each such word, the function checks if it is a booster or negation word and calls the `booster_check()` or `negation_check()` function respectively, to modify the valence score accordingly. Finally, the function applies a *least* check to the valence, which adjusts the score based on the context of the current word and its predecessor.

D.1 booster_check() This function is used to calculate the effect of booster words on the sentiment valence. It takes four arguments as input, (i) the score of the booster word, (ii) the current sentiment valence, and two integer values indicating (iii) whether the current word is uppercase and (iv) whether the sentence has a cap differential. The function returns a float value representing the effect of the booster on the sentiment valence.

The function first initializes a float variable called "scalar" to 0. If the booster word is "so" or "very", then the scalar is set to a predetermined value. Otherwise, the scalar is set to the score of the booster word in the lexicon. If the sentiment valence is negative, then the scalar is multiplied by -1. Finally, if the current word is uppercase and the sentence has a cap differential, the scalar is increased or decreased by 0.733 according to the sign of the current valence.

D.1.1. negation_check() The negation check function takes the valence score of the current word, the scores of all words in the sentence, and the positions of the current and previous words. The function first checks whether any negation word appears before the current word. If a negation word exists before the current word, the function multiplies the valence score by a negative scalar (-0.74), which inverts the polarity of the sentiment.

The function also examines two very specific possible patterns in a sentence. The first one is the case of "... never so/this <matched_word> ...". The other case is the word "never" appearing three words prior to the current word (e.g., "... never <other_word> so/this <matched_word> ..."). In these cases, the function multiplies the valence score by a scalar (1.25) to compensate for the reduction in score caused by the negation words.

D.1.2. least_check() This function is used to apply a *least* check to the sentiment valence, which adjusts the score based on the presence of negation words in the text. The

function takes the current sentiment valence, the scores of the words in the text, the pattern ID of the previous word and the index of the current word. The function returns a float value representing the adjusted sentiment valence.

It first checks if the current word is the first word in the text to determine if it has preceding words. If the previous word is "least", the sentiment valence is multiplied by -0.74. If the previous word matched to "least" but the word before that is not "at" or "very", then the sentiment valence is also multiplied by -0.74. So the phrases "at least" and "very least" will have no extra effect on the current word.

E. `but_check()` The data structure that contains information about every word of a single tweet, includes a variable indicating whether a word matches the contrastive conjunction "but" and another one representing the exact position within the text if there is a "but" in the sentence. In case of multiple occurrences of the word "but", we consider only the first one. If the `but_check` flag is set, the function iterates over each word in the tweet and multiplies the sentiment score of each word by a factor of 0.5, if the word appears before the "but", and by a factor of 1.5, if the word appears after the "but".

This contrastive effect is implemented by decreasing the sentiment scores of words before the "but" and increasing the scores of words after the "but", with the assumption that the words after the "but" are more influential in determining the overall sentiment of the tweet.

F. `amplify_scores()` (`score_valence` in Vader) The `amplify_scores()` function calculates sentiment scores including amplifiers for a given text. First, it iterates over the array of sentiment scores for each word and calculates the sum of all the scores. If the punctuation amplifier value is positive, it adds it to the sum and subtracts it if it is negative. It then normalizes the sum using the `normalize()` function, stores the result in the "compound" variable, separates the positive from the negative sentiment scores, and counts the number of neutral words using the `sift_sentiment_scores()` function. Then it calculates the `pos_sum`, `neg_sum`, `neu_count`, and `total` values. Finally, it calculates the final `pos`, `neg`, and `neu` values, rounds them to three decimal places (using our own implementation of `roundFloat()`), stores them in the sentiment dictionary, and returns it to the main `polarity()` function.

F.1. `normalize()` The `normalize()` function takes as input the sentiment score and an `alpha` value (15) that approximates the maximum expected value. It normalizes the sentiment score between -1 and 1 using the formula $norm_score = score / \sqrt{(score * score) + alpha}$. If the normalized score is less than -1, it is truncated to -1, and if it is greater than 1, it is truncated to 1.

F.2. `shift_scores()` The function sifts through an array of sentiment scores and separates them into positive, negative, and neutral. It takes as input an array of sentiment scores and the number of words in the text, and initializes three variables, (i): `pos_sum`, (ii) `neg_sum`, and (iii) `neu_count`, to zero. It then iterates over the array of sentiment scores and adds the scores to `pos_sum`, if they are positive, to `neg_sum`, if they are negative, or increments `neu_count`, if they are equal to zero. The function returns a pointer to a float array of

three cells which contains the sum of positive, negative and neutral sentiment scores, respectively.

G. store_data() The last stage of the `polarity()` function is to store the results derived from the sentiment analysis performed by all the functions described above. For this purpose, we store the tweet IDs along with the calculated sentiment scores. For each text entry (tweet), we report four sentiment scores, (i) *negative*, (ii) *neutral*, (iii) *positive*, and (iv) *compound*, next to each corresponding ID.

3.7.1 Sentiment Report

The last step is to generate a file that contains the sentiment scores for each tweet. In order to be easier to read and analyze, we compile the report so that each line corresponds to each tweet from the input file and includes five columns, (i) *ID*, (ii) *negative*, (iii) *neutral*, (iv) *positive*, and (v) *compound*. The *ID* is the unique identifier of every Twitter post. The *compound* score represents the sum of the valence scores of each word, adjusted according to our re-implementation of Vader's logic and normalized between -1 (most extreme negative) and +1 (most extreme positive). The *negative*, *neutral*, and *positive* scores are ratios indicating the proportions of text that fall in each category (so all three add up to 1).

3.8 Optimizations

In this section, we summarise the various optimizations that we performed over Vader's original logic to boost our system's overall performance as well as other design choices.

3.8.1 Single Data Preparation Pass

As described above, before sending data to the multi-core processor, we perform a text tokenization. During this phase that iterates the entire input character-by-character, it is an ideal chance to utilize the loop and perform various operations, such as converting the input to lowercase, checking if the original input is in lowercase or uppercase, calculating the capitalization differentials, etc.

In this way, once the tokenization is completed, we have all the necessary information needed later during the sentiment analysis process without re-iterating the entire input each time it is needed, as originally performed by the vanilla Vader implementation.

3.8.2 Special Word and Character Encoding

To completely avoid using the naive string comparison function, in contrast to the vanilla Vader, we encode booster and negation words and other special words and characters in the lexicon, using special extreme scores (e.g. -999), serving as flags. Every time we need to compare the input with a certain word, such as "kind", "of", "so", "very", etc., we simply check for these scores, already assigned by the OpenCL-based pattern matching

engine, instead of calling string comparisons for every word, as originally performed by Vader.

Since all the sentiment scores in the lexicon range from -4 to +4, we can use extreme values for the special words without affecting the lexicon's accuracy. For example, if we want to check if the current word is part of the phrase "kind of", we simply check if the score of the current word is 222 and if the score of the next word is 220. In this case, this would be an occurrence of the phrase "kind of". A sample of some of these special words is presented in Table 3.2.

Table 3.2: Sample of special words encoded into our modified Vader's lexicon.

word	negation	booster	score	flag
kind	no	no	2.4	222
of	no	no	0	220
no	no	no	-1.2	332
or	no	no	0	333
nor	yes	no	0	334
never	yes	no	0	442
so	no	yes(0.293)	0	443
this	no	no	0	444
very	no	yes(0.293)	0	553
at	no	no	0	554
least	no	no	0	555
but	no	no	0	666

3.8.3 Ring Buffer

If we need to utilize our system for sentiment analysis of hundreds of Gigabytes of input text with high speed I/O, a ring buffer can be utilized. Typically, the ring buffer holds multiple buckets, which allows for a pipeline between data gathering, data transferring, and data processing. Each bucket is an instance of the `databuff`, described above.

If we assume a four stage pipeline, the first bucket is the one that accepts new input data, the second bucket is full and being transferred to the device, the third bucket is being processed while the results of the fourth bucket are being transferred back to host memory. Each bucket is designed to take advantage of the SIMD capabilities of the available SIMT architecture, utilizing vector data types to increase overall performance via vectorized memory accesses, as described above. An overview of such a four stage ring buffer is presented in Figure 3.6.

3.8.4 Lexicon Automaton Memory Savings

As described above, the Aho-Corasick state machine compiled using all the patterns found in the sentiment lexicon is serialized as a DFA that can be presented as a two-dimensional

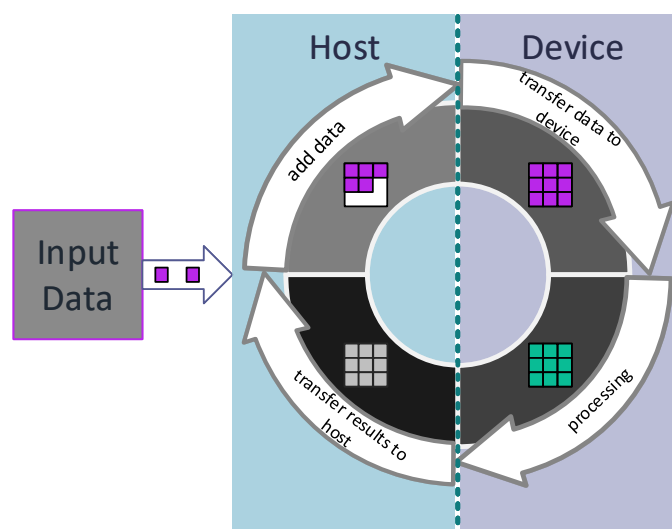


Figure 3.6: Overview of the four stage ring data buffer.

array. This array has 256 columns, as many as the characters in the ASCII set, and as many rows as the generated states. To further improve our system's memory footprint, we can discard all the unused columns before the lowest encountered character and after the highest encountered character. For example, if the lexicon only contains patterns in capital letters the array only has to be constructed with 26 columns. In our case, we can discard all columns before byte 33 (space character) and after byte 123 (" " character) as the lexicon only contains English words and printable characters. This results in a significant memory footprint reduction as more than half of the columns are discarded, multiplied by the number of states. This also has a positive effect on the automaton's caching properties and thus on the overall pattern matching performance.

Chapter 4

Evaluation

In this Chapter, we present the evaluation of our GPU-assisted sentiment analysis system with the workload described in §4.2 and Vader’s lexicon, which we modified as described in §3.8.2, using the hardware setup described in §4.1. First, in §4.4, we present the performance baseline and then we provide a thorough evaluation of our system in §4.5. We also present a comparison between our system and other similar tools in §4.6. In §4.7, we provide our system’s performance characteristics when executed on the CPU die, using OpenCL. We conclude our evaluation in §4.8 with a throughput comparison between our tool, executed on various multi-core-processors, and other tools discussed in this Chapter.

4.1 Experimental Setup

The host system utilized for the evaluation is equipped with an Intel® Core® i7-8700K CPU, clocked at 3.7GHz, with hyper-threading support that provides 12 logical cores and 16GB of dual-channel DDR3 DRAM clocked at 2400MHz. The CPU die also includes an Intel® UHD Graphics 630 iGPU. The system is equipped with an NVIDIA® GeForce RTX® 3080 Ti GPU, clocked at 1710MHz, providing 10240 CUDA Cores. The GPU is connected via the PCIe Gen 3 bus achieving 102Gbps of host-to-device and device-to-host data transfer throughput. The host is running Arch Linux with stock kernel (version 6.2.1), the vanilla NVIDIA driver (version 525.89.02) and NVIDIA OpenCL 3.0. Also, the `intel-opencl-runtime` and `intel-compute-runtime` packages provide OpenCL support for the CPU and iGPU respectively. No modifications are performed to the operating system’s kernel or the clock ratings of the various components. Unless stated otherwise, the external GPU is used for the evaluation described in this Chapter.

4.2 Datasets

To evaluate our system in terms of correctness, compared to the vanilla Vader implementation, and performance, we utilize several tweets retrieved using the Twitter API during

the Corona Virus pandemic, all related to the COVID-19 topic. Then, we pre-process the dataset, as described in the previous Chapter, and generate 21 test files. All test files contain a different amount of real tweets, written in English, with the first 9 test files containing from 1 thousand to 9 thousand tweets and the following 10 containing from 10 thousand to 100 thousand tweets. The last two test files contain 150,000 and 200,000 tweets respectively. Table 4.1 presents the files, their size, and the number of tweets each one contains. We choose to generate the data files as such to test the scalability of our system as well as identify how the system operates when processing real-world small (1K - 10K tweets), medium (10k - 100k tweets), and large topics (100K+ tweets).

Table 4.1: Test files generated by pre-processing Twitter datasets containing Tweets related to the COVID-19 topic.

File	Size	Number of Tweets
t01_1K.json	153KB	1.000
t02_2K.json	391KB	2.000
t03_3K.json	570KB	3.000
t04_4K.json	728KB	4.000
t05_5K.json	971KB	5.000
t06_6K.json	1.2MB	6.000
t07_7K.json	1.4MB	7.000
t08_8K.json	1.5MB	8.000
t09_9K.json	1.7MB	9.000
t10_10K.json	1.9MB	10.000
t11_20K.json	3.7MB	20.000
t12_30K.json	5.6MB	30.000
t13_40K.json	7.6MB	40.000
t14_50K.json	9.6MB	50.000
t15_60K.json	12MB	60.000
t16_70K.json	13MB	70.000
t17_80K.json	15MB	80.000
t18_90K.json	17MB	90.000
t19_100K.json	19MB	100.000
t20_150K.json	29MB	150.000
t21_200K.json	38MB	200.000

4.3 Lexicon

The lexicon used for our evaluation is our modified version of Vader’s original lexicon, as previously discussed. This lexicon contains only English words along with their respective scores. Also, Vader provides 2 more small lexicons, containing booster and negations words. Our system combines all this information into a single lexicon and also encodes special words with extreme score values, serving as flags. In total, the lexicon contains

7402 entries out of which, 7252 are English words, 80 are booster words, 57 are negation words and 13 are special words (flags).

4.4 Performance Baseline

To draw a baseline for the evaluation of our system, we process the workload described above using the vanilla Vader implementation as its logic is the basis of our system and Vader is considered a state-of-the-art lexicon-based sentiment analysis tool. To further compare our system’s performance with other popular systems utilizing Machine Learning algorithms and GPUs, we process the datasets on the same hardware using XML-RoBERTa and DeBERTa.

4.4.1 Vader Performance Analysis

To accurately measure Vader’s performance, we modify the original implementation to introduce fine-grained timers that allow us to monitor Vader in a per-function level and create a thorough time breakdown to identify the time consumed by each function as well as the required end-to-end time to process each dataset.

We evaluate Vader by executing 10 runs with each dataset and report the average time required to complete the sentiment analysis. The outcome of this evaluation is presented in Figure 4.1(a). For each dataset we report the time required by Vader to load the data and log the results (indicated as `dataLoad` and `dataStore` respectively) and the time required to perform the sentiment analysis (indicated as `polarity`). As we can see, the time required to process the tweets increases linearly as their number increases. Indicatively, Vader requires 1.5, 16, and 32 seconds to process 10,000, 100,000, and 200,000 tweets respectively. In Figure 4.1(b) we present this time breakdown as a percentage of the end-to-end time. We notice that, as expected, processing the tweets is the most consuming process requiring 95% of the total time.

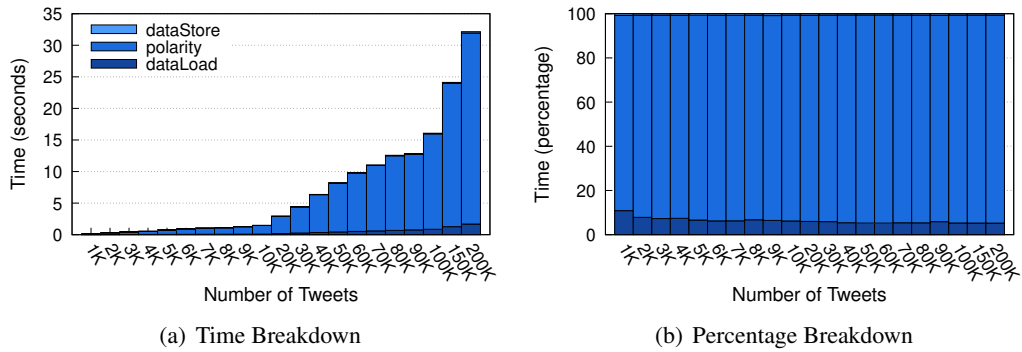


Figure 4.1: Time required by Vader to process the various datasets presented as a high-level function breakdown and as a percentage of the overall time.

Then, we further evaluate Vader’s sentiment analysis process by providing a more fine-grained time breakdown of the functions utilized in this process. The result of this analysis is presented in Figure 4.2(a) while the operation of each function utilized by the `polarity` function is described in §3.7. Figure 4.2(a) presents this breakdown as a percentage of the overall time consumed by the `polarity` function. As we can see in the figures, the most time-consuming function is `sentiVal` requiring $\sim 65\%$ of the overall time while another $\sim 5\%$ is consumed performing `toLower` and `isUpper` checks. Performing special checks to locate and handle "but" in the tweets requires $\sim 10\%$ of the overall time and another $\sim 10\%$ is `emojiStrip`.

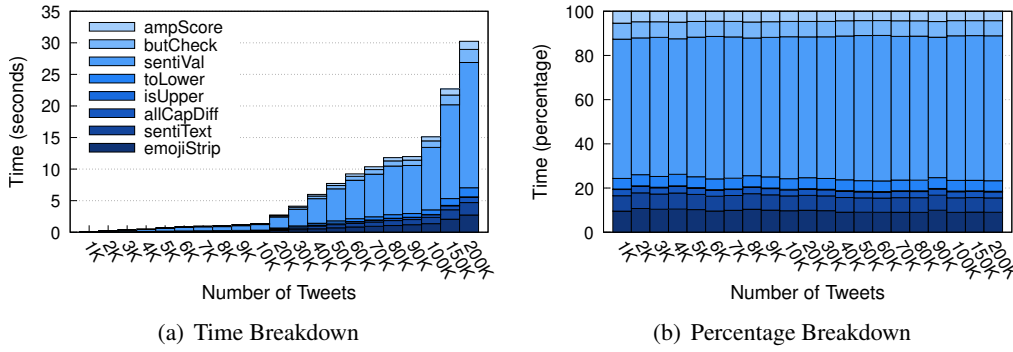


Figure 4.2: Time breakdown required by Vader’s `polarity` function to perform the sentiment analysis on the loaded data.

4.4.2 Machine-Learning Approaches

To draw a rough performance baseline for ML-based approaches, we utilize two separate models, namely XML-RoBERTa [10], a multi-language model that can provide sentiment analysis on over 110 languages, and DeBERTa [11], an implementation provided by Microsoft that only allows for English text sentiment analysis. The second model is developed to enable predictions for devices with limited hardware capabilities, such as IoT devices. As such, DeBERTa is a simple model able to achieve lower total execution time in comparison with more complex solutions like XML-RoBERTa.

We fine-tuned both models to identify their best possible configuration for our hardware. More specifically, for XML-RoBERTa we utilized a *batch size* of 12, 1 *gradient accumulation step* and set *gradient checkpointing* and *fp16* to `True` while *optim* was set to `adafactor`. For DeBERTa we chose the same configuration while the selected *batch size* was set to 62.

We evaluate both models by processing our Twitter workloads but we only measure the time that is required for each model to provide prediction and we omit data handling. Also, we only report the end-to-end time since a thorough function breakdown would not offer any valuable results that we could compare against Vader or our GPU-based system as these solutions are based on a completely different approach. The result of this analysis

is presented in Figure 4.3. We notice that DeBERTa is significantly faster than XML-RoBERTa by ~ 3.5 times. Also, the results indicate that the time required by both models to perform the analysis increases linearly with the amount of input data. However, even without measuring data handling, both models are significantly more time-consuming than Vader.

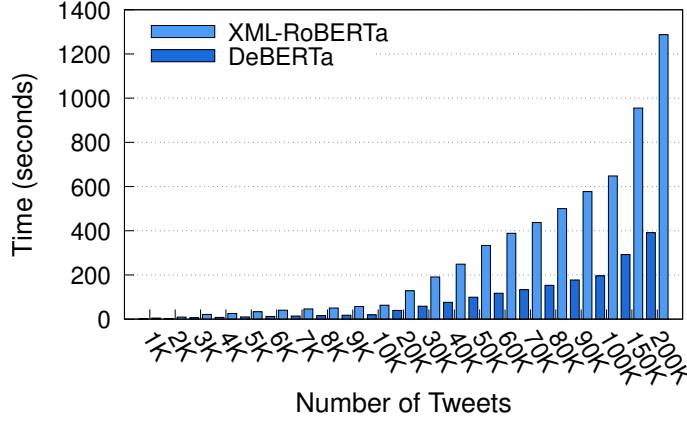


Figure 4.3: End-to-end performance comparison of DeBERTa and XML-RoBERTa when processing the Twitter workload.

4.5 GPU-Vader Performance Analysis

With the baseline identified, we proceed with the evaluation of our GPU-based sentiment analysis system. For each measurement we execute 10 runs with each dataset and report the average time required to perform the sentiment analysis, timing the functions involved in each process. We report both the time consumed by each function as well as the time consumed as a percentage of the overall processing time.

4.5.1 Overall Execution Time

We start our system’s evaluation by analyzing the overall time required to process each dataset and generate the sentiment analysis results. The result of this evaluation is presented in Figure 4.4(a). Compared to Vader, our system has an extra high-level function, named `engineInit`, that is responsible for initializing our system’s GPU engine context and the required data structures. As we can see in the figure, our system requires ~ 0.2 , ~ 0.9 and ~ 1.6 seconds to process 10,000, 100,000 and 200,000 tweets respectively. These results indicate that our system is **7.5 times faster** than Vader when processing 10K tweets, **17.7 times faster** when processing batches containing 100K tweets and almost **20 times faster** when processing batches of 200K tweets. Also, we can see that the engine’s initialization time is constant and consumes around 0.1 seconds.

Figure 4.4(b) presents this breakdown as a percentage of the overtime required to load

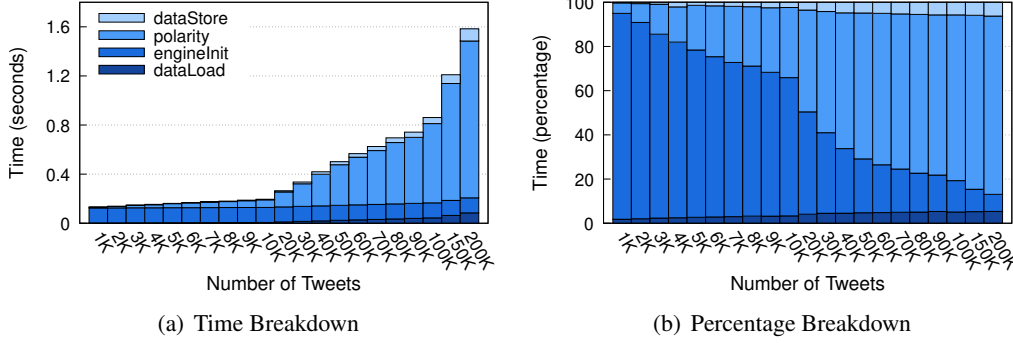


Figure 4.4: Time required by the GPU-based sentiment analysis system to process the datasets, presented as a high-level function breakdown and as a percentage of the total time.

the data, process the tweets and log the results. We notice that the engine’s initialization time consumes more than 50% of the overall time for workloads containing less than 10,000 tweets and less than 45% of the end-to-end time for workloads containing more than 10K tweets. The reason behind this behavior is that we report the average of 10 complete executions, starting from scratch, thus the engine is re-initialized between each run to obtain its average initialization time. In a real-world application, this percentage will be minimized as the system is designed to execute as a service, meaning that this initialization phase will occur only once, during the bootstrap phase, regardless of the amount of data it is going to process until it is shut down.

4.5.2 GPU-based `polarity` Performance

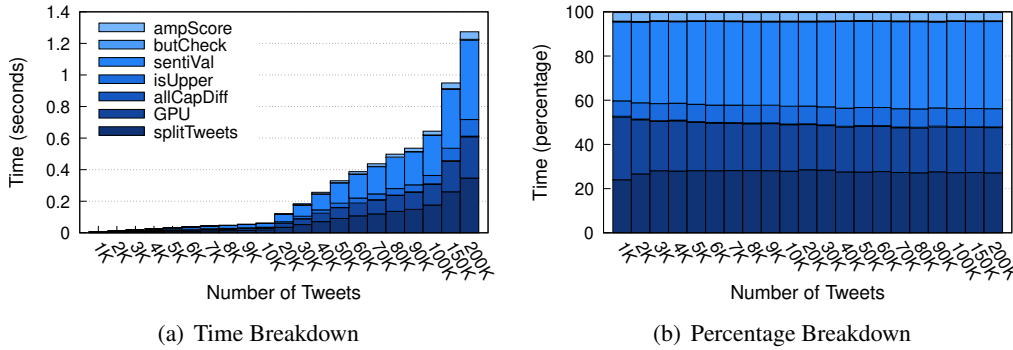


Figure 4.5: Time required by the GPU-based `polarity` function to process the workloads presented at a function-level breakdown.

We further evaluate our `polarity` function by placing fine-grained timers to each function called by `polarity`. The results are presented in Figure 4.5(b). Compared to Vader, our system does not use `sentiText`, `toLower`, and `emojiStrip` as their function-

ality is merged and handled in different functions, such as `splitTweets`. We notice that the two most time-consuming functions are `splitTweets`, performing all the necessary data transformations to prepare the data for GPU processing as well as the various data transformations performed by Vader, and `sentiVal`, consuming $\sim 30\%$ and $\sim 40\%$ of the overall time respectively. Also, due to our optimizations, `butCheck` requires only $\sim 0.2\%$ of the overall time, compared to $\sim 10\%$ required by vanilla Vader. The same result is also observed for `allCapDiff`, consuming only $\sim 0.2\%$ of the total time, compared to $\sim 5\%$ required by the original Vader implementation. The total time consumed by the GPU engine for all its operations, such as initialization, data transfers, and processing, is $\sim 20\%$ of the overall sentiment analysis time.

4.5.3 GPU Engine Performance

We complete our system’s performance analysis by presenting the time breakdown of its GPU engine. The outcome of this evaluation is presented in Figure 4.6(a). The reported functions are `addToBuffer`, responsible for filling the required GPU buffers with the input tweets, `hostToGPU` and `GPUtoHost`, responsible for transferring the input and the results to and from the GPU respectively, `scanWords` and `filterResults`, responsible for assigning scores to the words and `bufferReset` which handles the data structures after each execution.

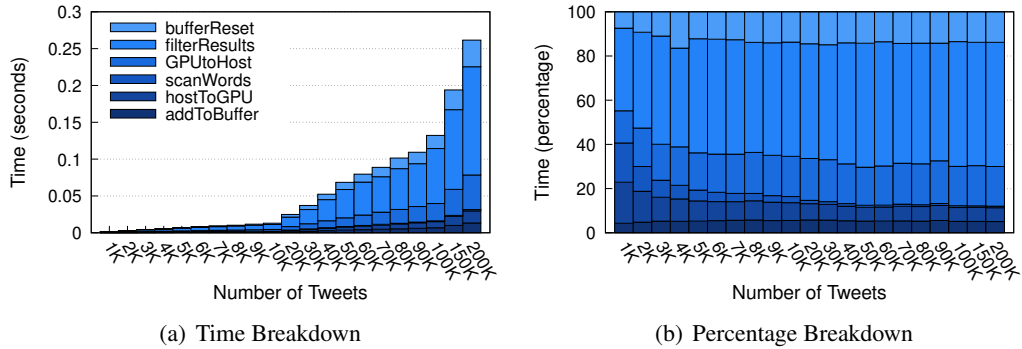


Figure 4.6: Time required by our GPU engine to process the workloads, presented at a function-level breakdown.

As we can see in the figure, the GPU engine requires ~ 0.015 seconds to process 10K tweets, including returning the results to host DRAM, and ~ 0.13 and ~ 0.26 seconds to process 100K and 200K tweets respectively. In Figure 4.6(b) we can observe that filling the data buffers and transferring data consumes $\sim 35\%$ of the overall GPU engine time. On the other hand, processing the words with our pattern matching algorithm is very fast and efficient consuming on average 15% - 5% of the total GPU time and less than 2% of the overall GPU time for workloads containing more than 10K tweets. The most time-consuming function is `filterResults`, responsible for fine-tuning the initial scores assigned by `scanWords` costing $\sim 60\%$ of the end-to-end GPU time.

4.6 End-to-End Performance Comparison

We conclude our performance analysis by comparing the total time required by all tools to perform the sentiment analysis, namely (i) vanilla Vader, (ii) our GPU-based system, (iii) DeBERTa, and (iv) XML-RoBERTa, using the same hardware host and workloads. We measure both processing and data I/O for Vader and our system while we only report the processing time for DeBERTa and XML-RoBERTa. The outcome of this analysis is presented in Figure 4.7. We point out that log-scale is used for the Y-axis due to the vast deviation of the end-to-end results for each tool. As we can see in the figure, our system is significantly faster than any other tool, requiring ~ 1.6 seconds to process the biggest workload containing 200K tweets. The vanilla implementation of Vader is the next fastest solution requiring an order of magnitude more time (~ 32 seconds) to process the 200K dataset. DeBERTa is the fastest of the ML-based approaches but requires 12x more time compared to Vader while our solution outperforms DeBERTa, being **244 times faster**. Finally, XML-RoBERTa is the slowest of all the evaluated solutions, consuming ~ 22 minutes to process the biggest workload. Our approach manages to greatly outperform XML-RoBERTa, being approximately **804 times faster** when processing batches of 200K tweets.

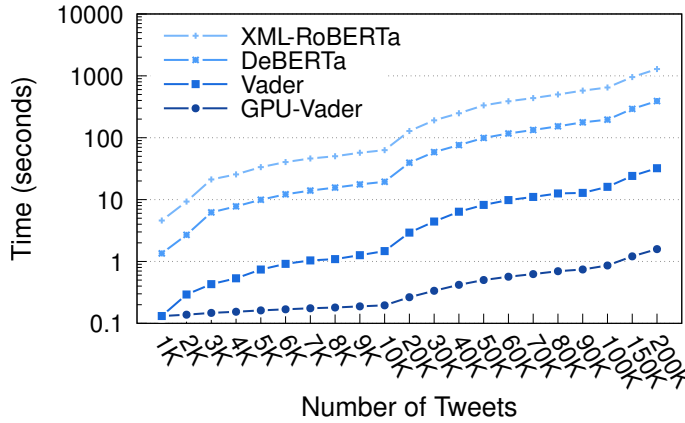


Figure 4.7: End-to-end performance comparison between XML-RoBERTa, DeBERTa, Vader and our GPU-based sentiment analysis system when processing workloads containing 1K - 200K tweets.

4.7 Execution on the CPU Die

Developing our system using C and OpenCL, allows us to execute the tool on every multi-core processor supporting the framework. This means that the system does not require an external GPU to operate. To explore its performance characteristics, we perform the same experiments described above, this time using the CPU and the iGPU, reporting the results in Figure 4.8. The results indicate that the CPU manages to process the 200K tweet workload utilizing 0.05 seconds less than the external NVIDIA GPU while the iGPU is

0.05 slower than the external GPU. At first, these results may seem counterintuitive, as the external GPU has way more processing capacity than the CPU die. However, the reason behind this behavior can be, partially, revealed by inspecting the OpenCL engine time breakdown, presented in Figure 4.9.

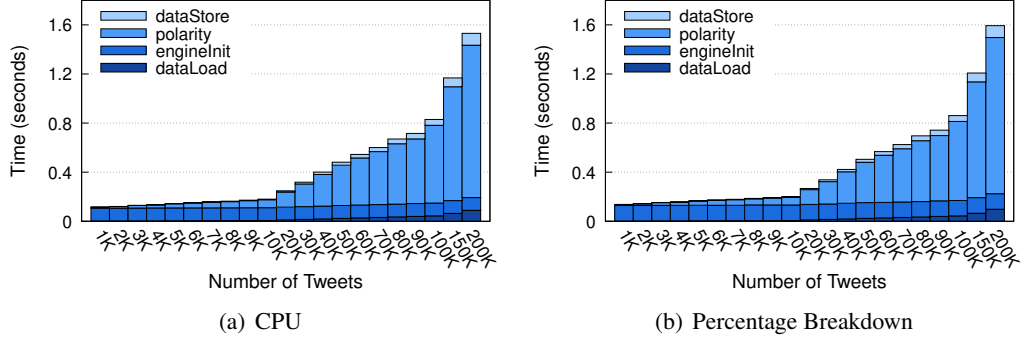


Figure 4.8: Time required by the CPU and iGPU to process the datasets using our OpenCL-based sentiment analysis tool, presented as a high-level function breakdown and as a percentage of the total time.

As we can see in the figure, both CPU and iGPU require zero time to perform `host-to-device` and `device-to-host` operations (data transfers), since both devices have direct access to the host’s memory space, as previously discussed. Thus, the `host` and `device` buffers can be memory mapped. The time gained by not requiring data to be transferred across the PCIe Bus, overshadows the slightly increased time required by the CPU and iGPU to process the results, despite having less computational capacity than the external GPU, rendering both devices equally comparable to the NVIDIA GPU.

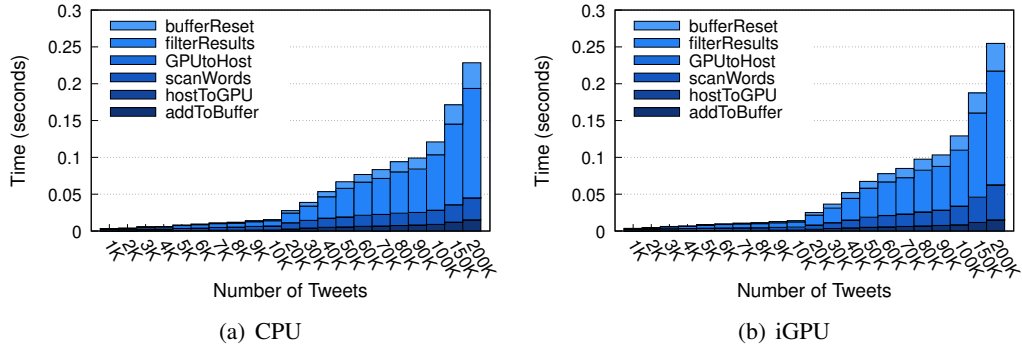


Figure 4.9: Time required by our OpenCL engine to process the workloads using the CPU and iGPU, presented at a function-level breakdown.

The performance benefits of the external GPU however will become more evident if the GPU is connected to a PCIe Gen 4 bus, which provides twice the bandwidth compared to the PCIe Gen 3 bus that we utilized in our evaluation. Also, the benefits of utilizing

an external GPU will increase as the overall load of input data increases to hundreds of thousands of texts, as the external GPU has not reached its potential maximum capacity. However, these results indicate that our system can provide very significant performance benefits, compared to other lexicon- or ML-based approaches, even when utilizing commodity CPUs/iGPUs and does not necessarily require the addition of an external GPU. However, despite the size of the input, utilizing an external GPU lifts the computation overhead from the CPU, allowing it to execute other applications or services.

4.8 Throughput Comparison

We conclude our evaluation by presenting the scalability characteristics of our system, executed on all available multicore-processors. To achieve this, we plot our system's throughput as a function of tweets processed per second and compare the results against the other three tools, vanilla Vader, XML-RoBERTa and DeBERTa. The results of this analysis are presented in Figure 4.10. As we can see in the figure, the vanilla Vader and the ML-based solutions have reached their maximum throughput from the first dataset (1K tweets) with vanilla Vader achieving $\sim 7K$ Tweets/sec, DeBERTa ~ 500 Tweets/sec and XML-RoBERTa ~ 160 Tweets/sec. On the other hand, our system keeps scaling its performance as the input size increases, reaching up to $\sim 130K$ Tweets/sec. Also, the curve's trend indicates that our system has not reached its maximum performance capacity even for workloads containing 200K tweets.

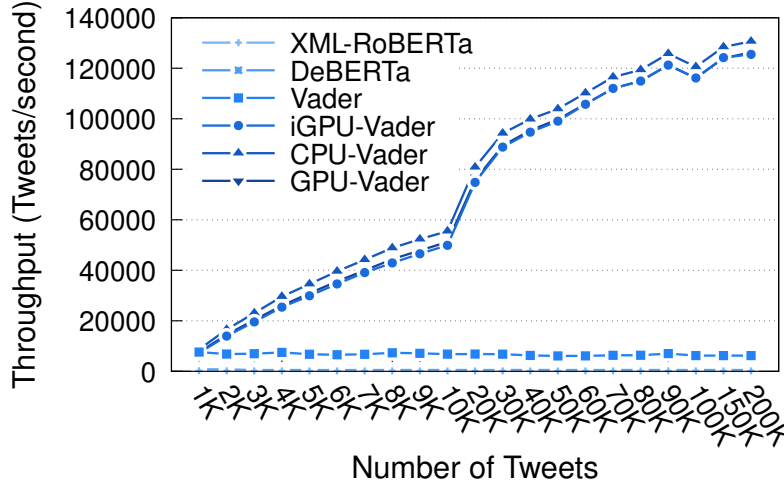


Figure 4.10: Throughput comparison between our system executed on GPU, CPU and iGPU, the vanilla Vader implementation, XML-RoBERTa, and DeBERTa.

To further explore our system's maximum sustainable throughput with the current hardware setup, we utilize all three available multi-core devices to process datasets ranging from 100K to 500K tweets. The results of this analysis are presented in Figure 4.11. We notice that our system is able to achieve even higher throughput (Tweets/second) when

processing batches containing more than 200K tweets. Also, we can see that all devices reach their maximum performance when processing datasets containing more than 400K tweets. Moreover, we notice that the GPU and iGPU yield very similar throughput, maxing at ~ 150 K Tweets analyzed per second. Also, the CPU is able to achieve $\sim 3.5\%$ higher performance, being able to process up to 155K Tweets/second.

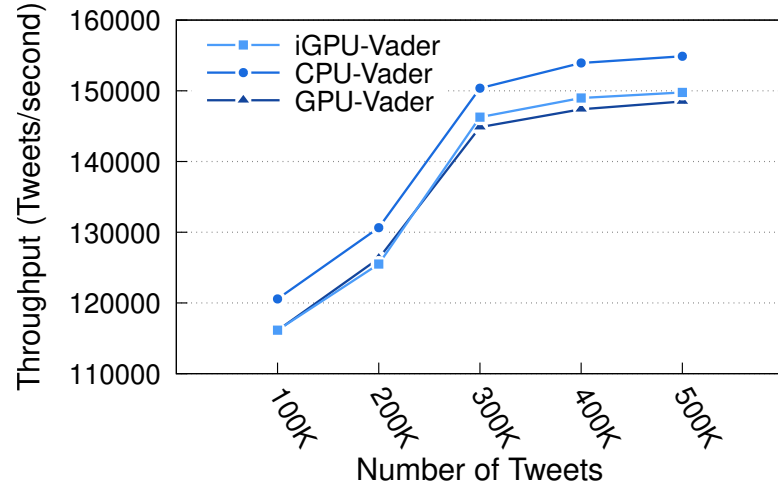


Figure 4.11: Throughput comparison between our system executed on GPU, CPU and iGPU for datasets ranging from 100K to 500K tweets.

Chapter 5

Related Work

5.1 Sentiment Analysis

Pang et al. [12] were pioneers in the field of sentiment analysis and viewed sentiment analysis as a text classification (special topic-based) problem, aiming to better understand the sentiment in the text with a more specific classification. To train their classifiers, they created a sentiment-annotated corpus which would later form the basis of supervised sentiment classification approaches. They utilized machine learning methods (NB, ME, SVMs) with a standard bag-of-features framework on a movie reviews dataset. Their experiments focused on different feature sets such as unigrams, bigrams, POS, and others. The authors observed that sentiment classification is not as easy as standard topic-based classification and concluded that using an SVM classifier with binary unigram-based features produced the best results by accounting just the feature presence. As such, they stressed the importance of feature selection in achieving good classification results.

Hu and Liu (2004) [13] proposed a two-step process for mining and summarizing customer reviews. The first step is to use text mining techniques to extract opinions from the reviews by identifying the features of a product that customers are likely to have opinions about and then extracting the opinions associated with each feature. The authors used a technique called *feature-based opinion mining approach* which utilizes the adjective synonym and antonym sets in WordNet [14] to predict the semantic orientations of adjectives. The second step involves summarizing the opinions to provide an overall sentiment of the product using a technique called *review summarization*. Their work has served as a foundation for further research in sentiment analysis on customer reviews.

Gonçalves et al. (2013) [15] conducted a comparison of eight sentiment analysis methods used for analyzing sentiment in online social networks. The authors evaluated the methods on various datasets and found that SentiWordNet [16] had the highest coverage (i.e., the proportion of text with identified sentiment), while LIWC showed the highest agreement (i.e., smallest deviations from other lexicons). However, the authors observed that no single method was consistently superior across different text sources, prompting them to

combine the methods to enhance overall performance. The resulting method achieved a high coverage of 95% and maintained relatively high accuracy and precision in sentiment analysis with an F-measure of 0.730.

5.2 Machine Learning approaches

There are efficient tools that perform automatic sentiment analysis, with minimal programming effort, such as NLTK [17] for Python. The SenticNet 7 framework [18] stands out as a unique approach that integrates neurosymbolic AI and commonsense knowledge. Neurosymbolic AI is a combination of neural networks and symbolic reasoning which allows the model to handle both numerical and symbolic data. With the incorporation of commonsense knowledge, the model is able to understand the meaning and context of words and phrases in a piece of text, allowing it to better determine the sentiment expressed in that text. The SenticNet 7 also offers an explanation for its sentiment predictions, enhancing the transparency and trust in its results and providing improved performance.

Robert Marcec and Robert Likic [19] examined the application of sentiment analysis to understand public opinions and attitudes towards three COVID-19 vaccines. They collected tweets and classified them into positive, negative, or neutral sentiments using a sentiment analysis tool. The results indicate that the overall sentiment towards AstraZeneca/Oxford vaccine was more negative compared to Pfizer/BioNTech and Moderna vaccines, and the sentiment varied based on location and source of tweets. They caution that sentiment analysis may not accurately reflect all opinions, emphasizing the need to consider the context and source of the data.

In 2013, Google introduced a series of word2vec models, which were trained on a massive corpus of 1.6 billion words and quickly gained widespread popularity for a wide range of NLP tasks [mikolov2013distributed]. Building on this success, Devlin et al. [20] introduced BERT in 2018, a bidirectional transformer-based model that currently represents the state-of-the-art in language understanding. BERT is trained on a dataset of 3.3 billion words and boasts an impressive 340M parameters, making it one of the most powerful language models in use today. Blum and Mitchell (1998) [21] proposed co-training, a semi-supervised learning method that trains two classifiers on two different sets of features, where one classifier provides additional training data for the other. He and Zhou (2011) [22] introduced a self-training method for sentiment analysis that uses labeled features to train a classifier, and then unlabeled data to improve the classifier. Al-Harbi and Rayward-Smith (2006) [23] adapted the k-means clustering algorithm for supervised clustering tasks. Moraes et al. (2013) [24] compared the performance of SVM and ANN classifiers for document-level sentiment classification.

5.3 Deep Learning Approaches

Hochreiter and Schmidhuber (1997) [25] introduced LSTM, a type of recurrent neural network architecture designed to overcome the problem of vanishing gradients in back-propagation through time. In recent years, LSTMs have been used for sentiment analysis tasks, such as in the work by Dos Santos and Gatti (2014) [26], where they used deep convolutional neural networks with LSTMs for sentiment analysis of short texts. In 2013, Socher et al. [27] proposed Recursive Deep Models for Semantic Compositionality over a Sentiment Treebank. The authors used recursive neural networks to build a model that could learn to compose the meaning of phrases and sentences in a hierarchical manner, and used it for sentiment classification on the Stanford Sentiment Treebank (SST). In a recent study, Li et al. (2020) [28] introduced a bidirectional LSTM architecture that leverages the relationship between target words and sentiment polarity words within a sentence, without the need for any external sentiment lexicons.

Another approach to sentiment analysis was proposed by Kim (2014) [29] using CNNs. With this work, Kim demonstrated that CNNs could achieve state-of-the-art performance on the SST dataset without the need for any explicit syntactic or semantic features. This model utilizes a single layer of convolution on top of word vectors obtained from an unsupervised neural language model (word2vec) and explores four distinct methods for learning word embeddings: (i) CNN-rand, which initializes all word embeddings randomly and subsequently updates them during training; (ii) CNN-static, which utilizes pre-trained word2vec embeddings that remain fixed during model training; (iii) CNN-non-static, which fine-tunes the word2vec embeddings during training for each individual task; and (iv) CNN-multi-channel, which employs two sets of word embedding vectors that are both initialized using word2vec, with one being updated during model training and the other remaining fixed.

Johnson and Zhang (2014) [30] introduced a model that makes effective use of word order in text classification with CNNs, while Zhang et al. (2015) [31] proposed a character-level convolutional neural network (CharCNN) for text classification. Both models achieved state-of-the-art results on the SST dataset. Zhou et al. (2015) [32] proposed a variant of LSTM called the Convolutional LSTM (C-LSTM) network for text classification. The authors demonstrated that the C-LSTM model outperformed both traditional LSTM and CNN models on several benchmark datasets. More recently, Johnson and Zhang (2017) [33] proposed a deep pyramid CNN model for text categorization, which achieved state-of-the-art results on several text classification tasks.

In addition to using different neural network architectures, recent works have also focused on pre-training methods for improving performance on sentiment analysis tasks. Sun et al. (2019) [34] proposed a fine-tuning method for the Bidirectional Encoder Representations from Transformers (BERT) model for text classification tasks, while Lan et al. (2019) [35] introduced ALBERT, a lite version of BERT that is specifically designed for self-supervised learning of language representations. Liu et al. (2019) [36] introduced RoBERTa, a robustly optimized pretraining approach for BERT, which achieved state-of-

the-art results on several natural language processing tasks, including sentiment analysis. Jiang et al. (2020) [37] introduced ConvBERT, a model architecture that improves upon BERT by incorporating span-based dynamic convolution modules in addition to self-attention modules. The authors note that while self-attention is effective at modeling global dependencies, it may not be as efficient at modeling local dependencies. Thus, the span-based dynamic convolution modules are used to model local dependencies, resulting in a mixed attention block. ConvBERT outperforms ELECTRA [38] using less than 1/4 of its pretraining cost.

Liao et al. (2021) [39] introduced RACSA (RoBERTa-based Aspect Category Sentiment Analysis) which utilizes RoBERTa to extract features from both the text and aspect tokens to enhance aspect-category sentiment analysis. RACSA combines RoBERTa's features with a 1D-CNN to extract phrases and incorporates a cross-attention mechanism to guide the weight allocation to relevant aspect categories. This mechanism focuses on the text fragments most pertinent to the given aspect category. Their approach showed improved results compared to other state-of-the-art methods and the study demonstrates the potential of combining BERT-based models with other techniques to improve sentiment analysis.

In a similar way, Tan et al. (2022) [40] also explored the use of RoBERTa in sentiment analysis and presented a hybrid model called RoBERTa-LSTM, which combines the transformer-based RoBERTa model with an LSTM RNN. RoBERTa-LSTM utilizes RoBERTa to encode the input text and then uses LSTM to model the sequential dependencies between the encoded representations. The authors demonstrated the effectiveness of their model on several benchmark datasets, showing that it outperformed several other state-of-the-art models. In addition, Tan et al. (2022) [41] proposed an ensemble hybrid deep learning model for sentiment analysis. This approach, called EHDLSA, combines multiple deep learning models, including RoBERTa, to improve the accuracy of sentiment analysis. Their results demonstrate that EHDLSA achieves better performance than individual models and traditional ensemble methods. Xie et al. (2020) [42] proposed unsupervised data augmentation for consistency training, which is a technique for improving the performance of neural networks by training them on both augmented and original data.

The work of Chan et al. [43] covers various techniques and architectures that have been used in sentiment analysis based on sequential transfer learning. The authors discuss the different stages of transfer learning, including pre-training, fine-tuning, and adaptation, and how they can be applied to sentiment analysis tasks. The authors identify the key challenges in sentiment analysis, including data sparsity, domain adaptation, and concept drift, and explore the various methods for addressing these challenges using sequential transfer learning. They also discuss the recent advances in deep learning models for sentiment analysis, such as recurrent neural networks (RNNs), convolutional neural networks (CNNs), and transformers, and how these models can be adapted for sequential transfer learning. They provide a detailed analysis of the different evaluation metrics used in sentiment analysis and highlight the limitations of current evaluation practices.

5.4 Lexicon-based Approaches

Hutto and Gilbert (2014) [44] introduced VADER (Valence Aware Dictionary for Sentiment Reasoning), a lexicon and rule-based approach to sentiment analysis. They developed a lexicon of over 7,500 features including words, emoticons, and acronyms, each annotated with a human-validated sentiment polarity score (positive/negative) and sentiment intensity score ranging from -4 to +4. The sentiment lexicon was designed with a focus on social media text, but it can be adapted to other domains. The sentiment valence of each feature is determined using SentiWordNet and the sentiment intensity is calculated based on various grammatical, syntactical, punctuation, and capitalization rules, as well as the examination of trigrams preceding the sentiment lexical feature. The sentiment classification is performed using word-sense disambiguation and machine learning algorithms implemented in Python using scikit-learn [45]. Hamilton et al. (2016) [46] proposed a label propagation framework to induce domain-specific sentiment lexicons from unlabeled corpora using a small set of seed words. Their method was evaluated on two different domains and demonstrated improved performance over existing sentiment lexicons, including SentiWordNet and the MPQA Subjectivity Lexicon [47].

One of the earliest works in the field is the study by Hatzivassiloglou and McKeown (1997) [48] on predicting the semantic orientation of adjectives. They proposed a method that uses the distributional similarity of adjectives to predict their polarity. In recent years, there has been growing interest in crowdsourcing techniques for sentiment analysis. Mohammad et al. (2009) [49] proposed a method for generating high-coverage semantic orientation lexicons using overtly marked words and a thesaurus. In 2013, Mohammad and Turney [50] proposed a word-emotion association lexicon that was created by crowdsourcing annotations from multiple workers.

Another approach to sentiment analysis involves building thesaurus lexicons using dictionary-based approaches. Park and Kim (2016) [51] proposed a method that uses a dictionary-based approach to build a thesaurus lexicon for sentiment classification. The study by Liu (2012) [52] provides a comprehensive overview of sentiment analysis and opinion mining. Araque et al. (2019) proposed a semantic similarity-based perspective of affect lexicons for sentiment analysis. They showed that the proposed approach outperforms existing methods on benchmark datasets. Zhang et al. (2012) proposed a weakness finder method that identifies product weaknesses from Chinese reviews using aspect-based sentiment analysis. Finally, Taboada et al. (2011) discuss the challenge of handling negation in sentiment analysis and present a manually curated list of negators and intensifiers, called Semantic Orientation CALCulator (SO-CAL), to address this issue. In their work, they compare and contrast the performance of various lexicon-based methods using a dataset of movie reviews.

5.5 Hybrid Approaches

Only a few models utilize a hybrid approach for sentiment analysis. Most of them use lexicon-based approaches to label word polarity to be further used in the sentiment analysis classifier. Gupta and Joshi (2020) [53] proposed a hybrid approach for Twitter sentiment analysis that combines a CNN and a bidirectional long short-term memory (BiLSTM) model. They also incorporated local contextual semantic information using WordNet to enhance the performance of their model. Deep Learning also can be combined with lexicons for the task of sentiment analysis. Shin et al. (2016) [54] proposed a lexicon-integrated CNN model with attention to sentiment analysis, which achieved state-of-the-art results on several benchmark datasets. Elshakankery and Ahmed (2019) [55] introduced HILATSA, a hybrid incremental learning approach for sentiment analysis of Arabic tweets that combines an unsupervised method with a supervised algorithm to improve the accuracy of sentiment classification. Asghar et al. (2018) [56] proposed T-SAF, a Twitter sentiment analysis framework that combines a set of pre-processing techniques with a hybrid classification scheme to achieve high accuracy on sentiment classification. Chikersal et al. (2015) [57] proposed SeNTU which combines a rule-based classifier with supervised learning to analyze the sentiment of tweets. Balage Filho and Pardo (2013) [58] presented NILCUSP, a hybrid system for sentiment analysis on Twitter that uses a lexicon-based approach with machine learning to improve the performance of the model.

Kouloumpis et al. (2011) [59] investigated three approaches to sentiment analysis: lexicon-based, machine learning-based, and hybrid. They evaluate several lexicons, including SentiWordNet and MPQA, and find that these approaches perform poorly on Twitter data. Additionally, they compare the performance of several machine learning algorithms, including Naive Bayes, Maximum Entropy, and SVM, and find that SVM performs the best. The authors also evaluate a hybrid approach that uses a lexicon to pre-process the data and a machine learning algorithm for classification and find that it performs better than either approach alone. However, they note that their evaluation is limited by the quality of the training data which is often noisy and may not be representative of the full range of sentiments expressed on Twitter. Overall, their work provides valuable insights into the challenges and opportunities of sentiment analysis in social media, particularly in the context of Twitter.

Kolchyna et al. (2015) [60] conducted a comparative analysis of three methods for Twitter sentiment analysis: lexicon-based, machine learning, and their combination. The authors evaluated the performance of each method using a dataset of 5,000 tweets related to the FIFA World Cup 2014 and four standard evaluation metrics: (i) precision, (ii) recall, (iii) F1-score, and (iv) accuracy. Their findings revealed that the combined approach outperformed the individual methods in all evaluation metrics. These results align with previous studies that have reported the effectiveness of combining different approaches for sentiment analysis. The study contributes to the existing literature by providing a detailed analysis of the strengths and limitations of each method and highlighting the potential of combining multiple approaches for improved performance in sentiment analysis.

5.6 Sentiment Analysis on Twitter

The first sentiment classification approach for Twitter was performed in 2009 by Go et al. [61]. The authors tackle the challenge of limited labeled data for sentiment analysis by utilizing distant supervision, where they automatically label large amounts of data from Twitter. They use a combination of n-grams, bigrams, and POS tags as features for sentiment classification, resulting in an accuracy of around 82%. This approach was more effective than the traditional supervised sentiment classification method. Their distant supervision approach provided a scalable solution for sentiment classification in the context of social media and micro-blogging platforms. Many subsequent studies have used this study as a reference point, partly due to the availability of the training dataset.

Pak and Paroubek [62] aimed to demonstrate the potential of Twitter as a corpus for sentiment analysis and opinion mining. They collected and analyzed a large-scale dataset of tweets, exploring various aspects of sentiment analysis, such as sentiment polarity, subjectivity, and sentiment-bearing words and phrases. The authors evaluated rule-based methods and machine learning techniques and compared their results with traditional sentiment analysis techniques on other datasets. The results showed the viability of using Twitter as a corpus for sentiment analysis and opinion mining and opened up new avenues for research in this field.

Wang et al. [63] presented a system for real-time sentiment analysis of tweets related to the 2012 US presidential election cycle. The authors propose a method to collect and analyze tweets in real-time and use this data to track the sentiment towards the candidates and the issues during the election. The system uses a combination of natural language processing techniques and machine learning algorithms to classify tweets as positive, negative, or neutral. The authors also evaluate the system on a dataset of tweets collected during the election and show that it achieves high accuracy in sentiment classification.

A comprehensive overview of the state of research on Twitter from its founding year in 2006 to 2020, can be found in the survey by Antonakaki et al. [64]. They introduced the basics of Twitter, its data model, and they described the graph structure of Twitter, including the relationships between users, the flow of information, and the influence of users on each other. The authors present an overview of sentiment analysis techniques applied to Twitter data, including the 8 most popular methods, lexicon-based approaches, machine learning-based approaches, and hybrid methods. Also, they present the challenges associated with sentiment analysis on Twitter, including the informal nature of the language used on the platform and the difficulty of accurately identifying sentiment in short, fragmented messages. Moreover, they discuss various types of attacks on Twitter, including spam, malware, and misinformation, highlighting the need for improved security measures and for more research into the impact of these attacks on the users and the platform itself.

In their 2010 study, Kwak et al. [65] examined the nature of Twitter, specifically whether it functions more as a social network or a news media. By analyzing the follower-followee network structure and tweet content, the authors concluded that Twitter can be considered

as both a social network and a news media, with the latter playing a more dominant role. This study provides valuable insight into the multifaceted nature of Twitter and its potential as a platform for both social interaction and information dissemination. The findings of this study are relevant to the current research as they shed light on the diverse ways in which users engage with and utilize Twitter, highlighting its significance as a communication tool in contemporary society.

Bollen et al.'s (2011) [66] study presents a unique approach to stock market prediction by examining the collective sentiment of Twitter users. The authors introduce a novel methodology that utilizes sentiment analysis of Twitter messages to forecast the movements of the Dow Jones Industrial Average. The study shows that their sentiment analysis approach outperforms existing techniques, indicating the potential value of social media data for predicting financial markets. Bollen et al.'s (2011) work represents a significant contribution to the field of computational social science, particularly in the area of applying sentiment analysis to non-traditional data sources for prediction tasks. Their study has garnered significant attention in the literature and serves as an important reference for researchers exploring the intersection of social media and financial markets. Finally, Ribeiro et al. [67] proposed Sentibench, a benchmark comparison of state-of-the-practice sentiment analysis methods. They evaluated the performance of various methods on a dataset of tweets and found that a combination of methods improves the overall performance.

5.7 GPGPU Acceleration

Pathuri et al. [68] propose a CUDA-SADBM classification model for sentiment analysis based on features extracted from social media text data, primarily from Twitter implemented in parallel computing. The authors also compare the performance of the CUDA-SADBM model to other widely used machine learning algorithms such as Naive Bayes, Support Vector Machines, and Random Forest. The results indicate that the CUDA-SADBM model outperforms these algorithms in terms of accuracy, precision, and recall, by executing it on both CPU and GPU.

Tran and Cambria [69] address the challenge of real-time multimodal sentiment analysis, by leveraging the processing speed of GPUs and ELM, and the accuracy of sentic memes. This work uses 47 YouTube video reviews as a dataset by Morency et al. and extracts visual, audio, and textual features that are classified into four different dimensions of sentiment (pleasantness, attention, sensitivity, and aptitude). To integrate the information extracted from different modalities, the authors employ both feature-level and decision-level fusion methods. The experimental results demonstrate an accuracy of 78%, which improved when they used all three modalities together.

Nirmal and Amalarethinam [70] present a parallel implementation of big data pre-processing algorithms for sentiment analysis of social networking data. The authors aimed to address the scalability issue of sentiment analysis by parallelizing the pre-processing stage of the data. The study showed that their approach could significantly reduce the execution time and improve the efficiency of sentiment analysis.

Bozkurt et al. [71] study sentiment analysis techniques for Turkish Twitter feeds using NVIDIA's CUDA technology. They implemented a CUDA-based distance kernel for k-NN algorithm to overcome the performance problems caused by the high dimensionality of feature space.

The work by Kolekar and Khanuja [72] presents a deep learning approach for sentiment analysis on airline tweet dataset using a proposed CNN model. The authors preprocess and split the Twitter dataset into training and testing datasets. They train the proposed CNN model on GPU and CPU and compare their performance, concluding that GPU speeds up computation power and is a good platform for training and testing large datasets with the proposed CNN network and does not require feature engineering.

GPUs and many-core processors have also been used to speed up the process of string pattern matching but not in the context of sentiment analysis [73, 74, 75, 76, 77] while other works utilize GPU-assisted pattern matching for Network Intrusion Detection and network packet processing [78, 79, 80]. Our proposed system is based on the approach of utilizing GPU-assisted pattern matching to speed up the process of score assignment for lexicon-based sentiment analysis.

Chapter 6

Conclusions and Future Work

In this Chapter, we present a summary of the contributions of this work (§ 6.1) as well as some insights on future work (§ 6.2). Finally, we conclude this dissertation in §6.3

6.1 Summary of Contributions

This work provides the following contributions to the field of sentiment analysis of social network data:

- We provide the first, to our knowledge, data-parallel lexicon-based sentiment analysis tool, based on Vader and implemented in OpenCL, that can execute on a wide variety of multi-core processors, such as GPUs, CPUs, and iGPUs.
- The system is able to outperform the state-of-the-art Vader sentiment analysis system by up to 20 times while achieving the same accuracy.
- We evaluate our system using real-world Twitter and lexicon, and provide a thorough comparison with other lexicon- and ML-based sentiment analysis approaches.
- We identify and implement a set of performance optimizations over Vader’s original implementation, many of which could be directly adopted by the vanilla Vader.

6.2 Future Work

As part of our future work, we plan to extend our pre-processor and execution pipeline to accept as input, datasets obtained by other popular social media networks, such as Facebook. Also, we plan to extend our system to accept multiple clients at a given time, utilizing more than one multi-core processor simultaneously, (e.g., multiple external GPUs). To further enhance the system’s performance, we will further analyze its current version to identify more data-parallel tasks that could be re-designed in OpenCL.

To be able to further increase our system’s accuracy and language coverage, we explore ways to generate fine-tuned lexicons using machine learning approaches. We believe that in this way, the system will be able to harness the benefits of both approaches (speed and accuracy) while covering a large number of languages and sentiment categories. Finally, we aim to investigate techniques to utilize this data-parallel sentiment analysis tool for other tasks, such as spam detection.

6.3 Conclusion

The exponential growth of OSNs and the information they can provide play an important role in understanding peoples’ feelings about various topics and such data are widely examined using sentiment analysis techniques. In this work, we propose an advanced data-parallel lexicon-based technique, based on Vader’s logic, to analyze and interpret a large amount of user-generated content available on social media platforms, such as Twitter, with the same accuracy but greatly increased execution performance.

The motivation behind this work stemmed from the limitations of traditional sentiment analysis methods, which were computationally expensive and unsuitable for large-scale analysis. By leveraging GPU acceleration, we aim to address these limitations and provide a more efficient and almost real-time solution for sentiment analysis.

To evaluate the effectiveness of our proposed lexicon-based sentiment analysis tool, we utilize a real COVID-19 dataset, obtained via the Twitter API, as a test case. This topic is chosen due to the enduring interest and trending nature of the COVID-19 topic, which provides a large and diverse dataset for analysis. By efficiently analyzing the sentiment of the COVID-19 dataset, we gained a deeper understanding of the potential of data-parallel lexicon-based systems in performing fast and accurate sentiment analysis using of-the-shelf multi-core processors. Our approach significantly improved processing speed, in comparison with other works, without sacrificing accuracy, yielding up to **20 times** higher computational performance compared to the original Vader implementation.

Chapter 7

List of Acronyms

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
ANN	Artificial Neural Network
BOW	Bag-of-Words
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DFA	Deterministic Finite Automaton
DNN	Deep Neural Network
ELM	Extreme Learning Machine
FPGA	Field-Programmable Gate Array
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
iGPU	Integrated Graphics Processing Unit
ILP	Instruction Level Parallelism
LIWC	Linguistic Inquiry and Word Count
LSTM	Long Short-Term Memory
ME	Maximum Entropy
MIMD	Multiple Instructions, Multiple Data
MISD	Multiple Instructions, Single Data

MPMD Multiple Programs, Multiple Data

NB Naive Bayes

NFA Non-Deterministic Finite Automaton

NLG Natural Language Generation

NLP Natural Language Processing

NLU Natural Language Understanding

OSN Online Social Network

POS Part-of-Speech

RNN Recurrent Neural Network

SA Sentiment Analysis

SIMD Single Instruction, Multiple Data

SIMT Single Instruction, Multiple Threads

SISD Single Instruction, Single Data

SPMD Single Program, Multiple Data

SST Stanford Sentiment Treebank

SVM Support Vector Machines

TLP Task Level Parallelism

Bibliography

- [1] “Natural language processing.” <https://www.ibm.com/topics/natural-language-processing>.
- [2] “Text mining vs. text analytics.” <https://www.ibm.com/topics/text-mining>.
- [3] W. Medhat, A. Hassan, and H. Korashy, “Sentiment analysis algorithms and applications: A survey,” *Ain Shams engineering journal*, vol. 5, no. 4, pp. 1093–1113, 2014.
- [4] M. Birjali, M. Kasri, and A. Beni-Hssane, “A comprehensive survey on sentiment analysis: Approaches, challenges and trends,” *Knowledge-Based Systems*, vol. 226, p. 107134, 2021.
- [5] “Machine learning - wikipedia. url:,” https://en.wikipedia.org/wiki/Machine_learning.
- [6] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. J. Miller, “Introduction to wordnet: An on-line lexical database,” *International journal of lexicography*, vol. 3, no. 4, pp. 235–244, 1990.
- [7] “Wordnet,” <https://wordnet.princeton.edu>.
- [8] C. Gilbert and E. Hutto, “Vader: A parsimonious rule-based model for sentiment analysis of social media text,” in *Eighth International Conference on Weblogs and Social Media (ICWSM-14)*. Available at (20/04/16) <http://comp.social.gatech.edu/papers/icwsml4.vader.hutto.pdf>, vol. 81, 2014, p. 82.
- [9] “The opencl framework.” <http://www.khronos.org/opencl/>.
- [10] “Xlm-roberta,” https://huggingface.co/docs/transformers/model_doc/xlm-roberta.
- [11] “Deberta,” https://huggingface.co/docs/transformers/model_doc/deberta.
- [12] B. Pang, L. Lee, and S. Vaithyanathan, “Thumbs up? sentiment classification using machine learning techniques,” *arXiv preprint cs/0205070*, 2002.

- [13] M. Hu and B. Liu, "Mining and summarizing customer reviews," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2004, pp. 168–177.
- [14] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.
- [15] P. Gonçalves, M. Araújo, F. Benevenuto, and M. Cha, "Comparing and combining sentiment analysis methods," in *Proceedings of the first ACM conference on Online social networks*, 2013, pp. 27–38.
- [16] S. Baccianella, A. Esuli, F. Sebastiani *et al.*, "Sentiwordnet 3.0: an enhanced lexical resource for sentiment analysis and opinion mining," in *Lrec*, vol. 10, no. 2010, 2010, pp. 2200–2204.
- [17] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009.
- [18] E. Cambria, Q. Liu, S. Decherchi, F. Xing, and K. Kwok, "Senticnet 7: A commonsense-based neurosymbolic ai framework for explainable sentiment analysis," in *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, 2022, pp. 3829–3839.
- [19] R. Marcec and R. Likic, "Using twitter for sentiment analysis towards astrazeneca/oxford, pfizer/biontech and moderna covid-19 vaccines," *Postgraduate Medical Journal*, vol. 98, no. 1161, pp. 544–550, 2022.
- [20] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
- [21] A. Blum and T. Mitchell, "Combining labeled and unlabeled data with co-training," in *Proceedings of the eleventh annual conference on Computational learning theory*, 1998, pp. 92–100.
- [22] Y. He and D. Zhou, "Self-training from labeled features for sentiment analysis," *Information Processing & Management*, vol. 47, no. 4, pp. 606–616, 2011.
- [23] S. H. Al-Harbi and V. J. Rayward-Smith, "Adapting k-means for supervised clustering," *Applied Intelligence*, vol. 24, no. 3, p. 219, 2006.
- [24] R. Moraes, J. F. Valiati, and W. P. G. Neto, "Document-level sentiment classification: An empirical comparison between svm and ann," *Expert Systems with Applications*, vol. 40, no. 2, pp. 621–633, 2013.
- [25] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

- [26] C. Dos Santos and M. Gatti, “Deep convolutional neural networks for sentiment analysis of short texts,” in *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*. Dublin, Ireland: COLING, 2014, pp. 69–78.
- [27] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, “Recursive deep models for semantic compositionality over a sentiment treebank,” in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1631–1642.
- [28] W. Li, F. Qi, M. Tang, and Z. Yu, “Bidirectional lstm with self-attention mechanism and multi-channel features for sentiment classification,” *Neurocomputing*, vol. 387, pp. 63–77, 2020.
- [29] Y. Kim, “Convolutional neural networks for sentence classification,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. [Online]. Available: <https://aclanthology.org/D14-1181>
- [30] R. Johnson and T. Zhang, “Effective use of word order for text categorization with convolutional neural networks,” *arXiv preprint arXiv:1412.1058*, 2014.
- [31] X. Zhang, J. Zhao, and Y. LeCun, “Character-level convolutional networks for text classification,” *Advances in neural information processing systems*, vol. 28, 2015.
- [32] C. Zhou, C. Sun, Z. Liu, and F. Lau, “A c-lstm neural network for text classification,” *arXiv preprint arXiv:1511.08630*, 2015.
- [33] R. Johnson and T. Zhang, “Deep pyramid convolutional neural networks for text categorization,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017, pp. 562–570.
- [34] C. Sun, X. Qiu, Y. Xu, and X. Huang, “How to fine-tune bert for text classification?” in *Chinese Computational Linguistics: 18th China National Conference, CCL 2019, Kunming, China, October 18–20, 2019, Proceedings 18*. Springer, 2019, pp. 194–206.
- [35] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, “Albert: A lite bert for self-supervised learning of language representations,” *arXiv preprint arXiv:1909.11942*, 2019.
- [36] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” *arXiv preprint arXiv:1907.11692*, 2019.
- [37] Z.-H. Jiang, W. Yu, D. Zhou, Y. Chen, J. Feng, and S. Yan, “Convbert: Improving bert with span-based dynamic convolution,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 12 837–12 848, 2020.

- [38] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, “Electra: Pre-training text encoders as discriminators rather than generators,” *arXiv preprint arXiv:2003.10555*, 2020.
- [39] W. Liao, B. Zeng, X. Yin, and P. Wei, “An improved aspect-category sentiment analysis model for text sentiment analysis based on roberta,” *Applied Intelligence*, vol. 51, pp. 3522–3533, 2021.
- [40] K. L. Tan, C. P. Lee, K. S. M. Anbananthen, and K. M. Lim, “Roberta-lstm: A hybrid model for sentiment analysis with transformer and recurrent neural network,” *IEEE Access*, vol. 10, pp. 21 517–21 525, 2022.
- [41] K. L. Tan, C. P. Lee, K. M. Lim, and K. S. M. Anbananthen, “Sentiment analysis with ensemble hybrid deep learning model,” *IEEE Access*, vol. 10, pp. 103 694–103 704, 2022.
- [42] Q. Xie, Z. Dai, E. Hovy, T. Luong, and Q. Le, “Unsupervised data augmentation for consistency training,” *Advances in neural information processing systems*, vol. 33, pp. 6256–6268, 2020.
- [43] J. Y.-L. Chan, K. T. Bea, S. M. H. Leow, S. W. Phoong, and W. K. Cheng, “State of the art: a review of sentiment analysis based on sequential transfer learning,” *Artificial Intelligence Review*, vol. 56, no. 1, pp. 749–780, 2023.
- [44] C. Hutto and E. Gilbert, “Vader: A parsimonious rule-based model for sentiment analysis of social media text,” in *Proceedings of the international AAAI conference on web and social media*, vol. 8, no. 1, 2014, pp. 216–225.
- [45] “scikit-learn, machine learning in python,” <https://scikit-learn.org/stable>.
- [46] W. L. Hamilton, K. Clark, J. Leskovec, and D. Jurafsky, “Inducing domain-specific sentiment lexicons from unlabeled corpora,” in *Proceedings of the conference on empirical methods in natural language processing. conference on empirical methods in natural language processing*, vol. 2016. NIH Public Access, 2016, p. 595.
- [47] “Mpqa subjectivity lexicon,” https://mpqa.cs.pitt.edu/lexicons/subj_lexicon.
- [48] V. Hatzivassiloglou and K. McKeown, “Predicting the semantic orientation of adjectives,” in *35th annual meeting of the association for computational linguistics and 8th conference of the european chapter of the association for computational linguistics*, 1997, pp. 174–181.
- [49] S. Mohammad, C. Dunne, and B. Dorr, “Generating high-coverage semantic orientation lexicons from overtly marked words and a thesaurus,” in *Proceedings of the 2009 conference on empirical methods in natural language processing*, 2009, pp. 599–608.
- [50] S. M. Mohammad and P. D. Turney, “Crowdsourcing a word–emotion association lexicon,” *Computational intelligence*, vol. 29, no. 3, pp. 436–465, 2013.

- [51] S. Park and Y. Kim, "Building thesaurus lexicon using dictionary-based approach for sentiment classification," in *2016 IEEE 14th international conference on software engineering research, management and applications (SERA)*. IEEE, 2016, pp. 39–44.
- [52] B. Liu, "Sentiment analysis and opinion mining," *Synthesis lectures on human language technologies*, vol. 5, no. 1, pp. 1–167, 2012.
- [53] I. Gupta and N. Joshi, "Enhanced twitter sentiment analysis using hybrid approach and by accounting local contextual semantic," *Journal of intelligent systems*, vol. 29, no. 1, pp. 1611–1625, 2020.
- [54] B. Shin, T. Lee, and J. D. Choi, "Lexicon integrated cnn models with attention for sentiment analysis," *arXiv preprint arXiv:1610.06272*, 2016.
- [55] K. Elshakankery and M. F. Ahmed, "Hilatsa: A hybrid incremental learning approach for arabic tweets sentiment analysis," *Egyptian Informatics Journal*, vol. 20, no. 3, pp. 163–171, 2019.
- [56] M. Z. Asghar, F. M. Kundi, S. Ahmad, A. Khan, and F. Khan, "T-saf: Twitter sentiment analysis framework using a hybrid classification scheme," *Expert Systems*, vol. 35, no. 1, p. e12233, 2018.
- [57] P. Chikersal, S. Poria, and E. Cambria, "Sentu: sentiment analysis of tweets by combining a rule-based classifier with supervised learning," in *Proceedings of the 9th international workshop on semantic evaluation (SemEval 2015)*, 2015, pp. 647–651.
- [58] P. Balage Filho and T. Pardo, "Nilc_usp: A hybrid system for sentiment analysis in twitter messages," in *Second Joint Conference on Lexical and Computational Semantics (*SEM), Volume 2: Proceedings of the Seventh International Workshop on Semantic Evaluation (SemEval 2013)*, 2013, pp. 568–572.
- [59] E. Kouloumpis, T. Wilson, and J. D. Moore, "Twitter sentiment analysis: The good the bad and the omg!" *Icwsn*, vol. 11, no. 538-541, p. 164, 2011.
- [60] O. Kolchyna, T. T. Souza, P. Treleaven, and T. Aste, "Twitter sentiment analysis: Lexicon method, machine learning method and their combination," *arXiv preprint arXiv:1507.00955*, vol. 5656, pp. 33–38, 2015.
- [61] A. Go, L. Huang, and R. Bhayani, "Twitter sentiment analysis," *Entropy*, vol. 17, p. 252, 2009.
- [62] A. Pak and P. Paroubek, "Twitter as a corpus for sentiment analysis and opinion mining," in *LREc*, vol. 10. Valletta, Malta: LREC, 2010, pp. 1320–1326.
- [63] H. Wang, D. Can, A. Kazemzadeh, F. Bar, and S. Narayanan, "A system for real-time twitter sentiment analysis of 2012 us presidential election cycle," in *Proceedings of the ACL 2012 system demonstrations*. Jeju Island, Korea: ACL, 2012, pp. 115–120.

- [64] D. Antonakaki, P. Fragopoulou, and S. Ioannidis, “A survey of twitter research: Data model, graph structure, sentiment analysis and attacks,” *Expert Systems with Applications*, vol. 164, p. 114006, 2020.
- [65] H. Kwak, C. Lee, H. Park, and S. Moon, “What is twitter, a social network or a news media?” in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.
- [66] J. Bollen, H. Mao, and X. Zeng, “Twitter mood predicts the stock market,” *Journal of computational science*, vol. 2, no. 1, pp. 1–8, 2011.
- [67] F. N. Ribeiro, M. Araújo, P. Gonçalves, M. André Gonçalves, and F. Benevenuto, “Sentibench-a benchmark comparison of state-of-the-practice sentiment analysis methods,” *EPJ Data Science*, vol. 5, pp. 1–29, 2016.
- [68] S. K. Pathuri, N. Anbazhagan, G. P. Joshi, and J. You, “Feature-based sentimental analysis on public attention towards covid-19 using cuda-sadbm classification model,” *Sensors*, vol. 22, no. 1, p. 80, 2022.
- [69] H.-N. Tran and E. Cambria, “Ensemble application of elm and gpu for real-time multimodal sentiment analysis,” *Memetic Computing*, vol. 10, pp. 3–13, 2018.
- [70] V. J. Nirmal and D. G. Amalarethnam, “Parallel implementation of big data pre-processing algorithms for sentiment analysis of social networking data,” *International journal of fuzzy mathematical archive*, vol. 6, no. 2, pp. 149–159, 2015.
- [71] F. Bozkurt, Ö. Çoban, F. Baturalp Günay, and Ş. Yücel Altay, “High performance twitter sentiment analysis using cuda based distance kernel on gpus,” *Tehnički vjesnik*, vol. 26, no. 5, pp. 1218–1227, 2019.
- [72] S. S. Kolekar and H. Khanuja, “Sentiment analysis using deep learning on gpu,” in *2018 IEEE Punecon*. IEEE, 2018, pp. 1–5.
- [73] X. Zha and S. Sahni, “Multipattern string matching on a gpu,” in *2011 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2011, pp. 277–282.
- [74] —, “Gpu-to-gpu and host-to-host multipattern string matching on a gpu,” *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1156–1169, 2012.
- [75] C.-H. Lin, S.-Y. Tsai, C.-H. Liu, S.-C. Chang, and J.-M. Shyu, “Accelerating string matching using multi-threaded algorithm on gpu,” in *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. IEEE, 2010, pp. 1–5.
- [76] M. C. Schatz and C. Trapnell, “Fast exact string matching on the gpu,” *Center for Bioinformatics and Computational Biology*, 2007.
- [77] C. S. Kouzinopoulos and K. G. Margaritis, “String matching on a multicore gpu using cuda,” in *2009 13th Panhellenic Conference on Informatics*. IEEE, 2009, pp. 14–18.

- [78] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis, “Regular expression matching on graphics hardware for intrusion detection,” in *Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009, Saint-Malo, France, September 23-25, 2009. Proceedings 12.* Springer, 2009, pp. 265–283.
- [79] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, “Gnort: High performance network intrusion detection using graphics processors,” in *Recent Advances in Intrusion Detection: 11th International Symposium, RAID 2008, Cambridge, MA, USA, September 15-17, 2008. Proceedings 11.* Springer, 2008, pp. 116–134.
- [80] G. Vasiliadis, L. Koromilas, M. Polychronakis, and S. Ioannidis, “Gaspp: A gpu-accelerated stateful packet processing framework.” in *USENIX Annual Technical Conference*, 2014, pp. 321–332.