UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

RIVERTEXT: A FRAMEWORK FOR TRAINING AND EVALUATING INCREMENTAL WORD EMBEDDINGS FROM TEXT DATA STREAMS

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

GABRIEL EMERSON ITURRA BOCAZ

PROFESOR GUÍA:
FELIPE BRAVO MÁRQUEZ

MIEMBROS DE LA COMISIÓN:
ANDRÉS ABELIUK
CLAUDIO GUTIÉRREZ
ELIANA SCHEIHING

Este trabajo ha sido parcialmente financiado por ANID FONDECYT grant 1200290, National Center for Artificial Intelligence CENIA FB210017 y ANID-Millennium Science Initiative Program - Code ICN17_002.

SANTIAGO DE CHILE
2023

RiverText: Un marco para entrenar y evaluar Vectores de Palabras Incrementales a partir de flujos infinitos de Datos de Texto

Los Word embeddings se han convertido en herramientas indispensables en varios procesos de lenguaje natural y tareas de recuperación de información, incluyendo clasificación de documentos, ranking y respuestas a preguntas. Sin embargo, los modelos de embeddings de palabras tradicionales tienen una limitación importante en su naturaleza estática, lo que dificulta su capacidad para adaptarse a los patrones de lenguaje en constante evolución que surgen en fuentes como las redes sociales y la web (por ejemplo, nuevos hashtags o nombres de marca). Para abordar este desafío, se han introducido algoritmos de embedding de palabras incrementales, que permiten la actualización dinámica de las representaciones de palabras en respuesta a nuevos patrones de lenguaje y flujos de datos continuos.

Esta tesis presenta RiverText, un framework para la entrenamiento y evaluación de word embeddings incrementales a partir de flujos de datos de texto. Nuestra herramienta proporciona un recurso valioso para la comunidad de procesamiento de lenguaje natural que trabaja con word embeddings en escenarios de streaming, como el análisis de redes sociales. La biblioteca implementa varias técnicas de word embeddings incrementales, incluyendo Skip-gram, Continuous Bag of Words y Word Context Matrix. Además, utiliza PyTorch como backend para la formación de redes neuronales, lo que permite una formación eficiente y flexible.

También hemos implementado un módulo que adapta tareas de evaluación de word embeddings estáticos intrínsecos, como la similitud y la categorización de palabras, a un entorno de streaming. Finalmente, comparamos el rendimiento de nuestro marco de trabajo utilizando diferentes configuraciones de hiperparámetros y discutimos los resultados.

Nuestra biblioteca de código abierto está disponible en `https://github.com/dccuchile/rivertext`. Incluye documentación detallada y ejemplos para ayudar a los usuarios a comenzar rápidamente y fácilmente con el marco de trabajo. Creemos que nuestro trabajo beneficiará en gran medida a investigadores y profesionales del procesamiento de lenguaje natural, especialmente a aquellos que trabajan con grandes volúmenes de datos de texto en streaming.

RiverText: A Framework for Training and Evaluating Incremental Word Embeddings from Text Data Streams

Word embeddings have become indispensable tools in various natural language processing and information retrieval tasks, including document classification, ranking, and question answering. However, traditional word embedding models have a major limitation in their static nature, which hinders their ability to adapt to the constantly evolving language patterns that emerge in sources such as social media and the web (e.g., new hashtags or brand names). To address this challenge, incremental word embedding algorithms have been introduced, enabling dynamic updating of word representations in response to new language patterns and continuous data streams.

This thesis presents RiverText, a comprehensive framework for training and evaluating incremental word embeddings from text data streams. Our tool provides a valuable resource for the natural language processing community that deals with word embeddings in streaming scenarios, such as social media analysis. The library implements various incremental word embedding techniques in a standardized framework, including Skip-gram, Continuous Bag of Words, and Word Context Matrix. Additionally, it uses PyTorch as its backend for neural network training, enabling efficient and flexible training.

We have also implemented a module that adapts intrinsic static word embedding evaluation tasks, such as word similarity and categorization, to a streaming setting. Finally, we compare the performance of our framework using different hyperparameter settings and discuss the results.

Our open-source library is available at `https://github.com/dccuchile/rivertext`. It includes detailed documentation and examples to help users get started with the framework quickly and easily. We believe that our framework will greatly benefit researchers and practitioners in natural language processing, especially those working with large-scale streaming text data.

Dedicado a mi familia y amigos, gracias por apoyarme en este viaje.
También a mi perro Rex, que en paz descanse.

Agradecimientos

Al terminar este trabajo se cierra una etapa de casi 9 años en la Facultad de Ciencias y Matemáticas de la Universidad de Chile. Un etapa donde viví muchas experiencia que me formaron y me ayudaron a llegar a este punto, de las que puedo destacar: haber pasado por dos departamentos, investigar sobre diferentes tópicos, cursar grandes asignaturas con los mejores profesores, haber formado parte de diferentes grupos estudiantiles, haber realizado docencia en múltiples cursos como profesor auxiliar y por último pero los más importante, haber conocido a una infinidad de personas valiosas.

Si bien la universidad es una parte importante en mi vida, no es la única, existe otro grupo de personas a la que quiero dedicar este trabajo. Ya que gracias a su cariño y apoyo incondicional nunca me habría convertido en la persona que soy ahora.

En primer lugar, quiero expresar mi profunda gratitud a mi madre. Mamá, tu inquebrantable confianza en mí ha sido mi motor durante esta travesía académica. No hay palabras suficientes para agradecerte todo lo que has hecho por mí. A mi padre, por tu sabiduría y constante orientación han sido invaluables en cada paso de mi educación. También quiero agradecer a mi hermana, que a lo largo de los años, hemos compartido risas, llantos, desafíos y triunfos juntos, nuestro vínculo es irrompible y estoy agradecido por tenerte en mi vida. Finalmente, a mi abuela Inés y a mis tías, Maritza, Paola y Katherine, que siempre han estado ahi para escucharme y darme palabras de aliento cuando los visitaba en Chillán.

A los amigos de la vida: Daniel Fica, Michael Jerez y Fernanda Rubio que siempre han estado ahí para compartir los momentos alegría y frustación durante este viaje, gracias por los buenos memes y las juntas que nos han acompañado todo estos años.

A lo amigos de la unversidad, en especial a mi grupo mechón: Daniel Montaner, Yasser Uarac y Bejamín Constant, gracias por por siempre estar presentes, por los buenos consejos, las idas en bici al cerro, las juntas al cine y los carretes que hemos mantenido a pesar del paso de tiempo. A la gente que conocí en Industrias, en especial a Franco Miranda, gracias por los buenos memes, las idas al cine y las cervezas que hemos compartido estos años. Por último, pero no menos importante, a mis amigos del DCC, Pablo Pizarro, Pablo Badilla, Ignacio Meza y Alonso Utreras, gracias por los buenos consejos y su amistad, sin ellos nunca habría terminado esta tesis.

También a todas las personas que de una u otra forma me acompañaron en este viaje y olvidé mencionar. Me es imposible a agregarlos a todos, pero gracias por los paseos, las cervezas, los consejos y conversaciones que hayamos compartido por la vida o en la

universidad. Quiero que sepan que su compañía me hizo muy feliz y sin ella probablemente nunca habría logrado terminar este proceso.

Llegando al plano académico, quiero agraceder a las personas del grupo de investigación ReLeLa y al departemento de Ciencias de la Computación, del cual formo parte, su formación me ha servido para darme cuenta que la investigación y la docencia son lo mío, y espero que en el futuro pueda seguir expandiendo y desarrollando las áreas que he aprendido.

Quiero agredecer directamente, a mi profesor guía y también quien ha sido mi mentor durante estos años, el profesor Felipe Bravo Márquez. Gracias por haber creído en mí sin conocerme al confiarme un tema de investigación poco explorado en la universidad. Además, su supervisión y buenos consejos han sido fundamentales en mi crecimiento como investigador, sin ellos nunca me hubiera atrevido a enviar mi trabajo a una conferencia internacional, a la cual este fue aceptado.

También me gustaría mencionar al profesor Claudio Gutiérrez, gracias por nuestros debates políticos y los consejos sobre como perseguir una carrera en la academía.

Para finalizar, agradezco el financiamiento del proyecto FONDECYT 11200290 titulado *"Tracking Social Public Opinion: A Stream-Mining Based Approach"*, al National Center for Artificial Intelligence CENIA FB210017 y al ANID-Millennium Science Initiative Program - Code ICN17_002 que hicieron este trabajo posible.

Table of Content

List of Tables

List of Figures

Chapter 1

Introduction

Human language, as an infinite source of information, has undergone numerous transformations throughout history. Moreover, with the advent of social media, the rate of change and evolution has accelerated significantly [23], making it increasingly challenging to process and comprehend all available data. Natural Language Processing [25] is a discipline that aims to provide strategies and algorithms for understanding and to process human language from various perspectives, including text analysis, speech recognition, language translation, and sentiment analysis, among other applications. Thus, NLP plays a crucial role in making sense of the vast information generated in today's digital age.

A significant challenge in the field of NLP is the representation of text in mathematical objects that are computationally tractable. Various approaches have been proposed to address this issue. However, the most prevalent method is the utilization of representations based on the distributional hypothesis [41]. The distributional hypothesis posits that words that appear in similar contexts tend to have similar meanings. This suggests that the meaning of words can be inferred through their contexts or, more specifically, through the words that co-occur with them.

Word Embedding (WE) [6], a representation of the distributional hypothesis, has become a fundamental component of NLP due to its exceptional ability to capture syntactic and semantic language relations in a vector space. These models are generally classified into two primary approaches: count-based approaches [91], also known as distributional models, and word embedding or distributed models [63]. The former approach constructs a context word matrix that counts the number of co-occurrences, while the latter relies on complex neural network structures. Subsequent research has established the equivalence of these approaches [55], demonstrating that both methods effectively capture the underlying linguistic regularities in text data.

However, the static nature of standard WE algorithms prevents them from incorporating new words, such as hashtags or new brand names, and adapting to semantic changes in existing words. For example, when unexpected events associated with an entity suddenly occur (e.g., a scandal related to a public figure or a company may change its perceived sentiment). Another scenario is when a given word acquires a new meaning in a particular event; for example, the word "ukrop" changed from "dill" to "Ukrainian patriot" during the

Russian-Ukrainian crisis [89]. To incorporate these changes into traditional WE, they must be retrained or aligned with new models to incorporate knowledge from new text sources, which is computationally inefficient.

The research community has proposed several methods for incremental learning in WE to address the drawbacks in traditional approaches. However, these methods, such as the incremental word context matrix (IWCM) [20], incremental skip-gram negative sampling (ISG) [46, 61], and incremental hierarchical softmax [76], lack a unified and transparent setup for comparison, hindering the examination and understanding of their quality and performance. This lack of information is crucial for deploying these algorithms in real-world NLP and applications.

In this thesis, we integrate these attempts into a new Python library called RiverText. This resource aims to unify and standardize the aforementioned methods into an easy-to-use toolkit. RiverText extends the interfaces provided by River [67], a machine-learning library for data streams, by enabling continuous learning of word embeddings from text streams, either from one instance at a time or from a mini-batch of [81] examples. In addition, it uses PyTorch [72] as its backend for implementing neural networks.

RiverText (as well as River) is based on the stream machine learning paradigm [31, 81, 2], which posits the following requirements for a learning algorithm [3, 69, 17]:

1. be able to process one instance (or mini-batch) at a time and inspect it (at most) once;
2. be able to process data with limited resources (time and memory);
3. be able to generate a prediction or transformation at any time;
4. is able to adapt to temporal changes.

In RiverText, we developed a standardized procedure to evaluate incremental WE methods to track the quality of embeddings throughout the stream. Our procedure performs a periodic evaluation (e.g., after processing a certain number of text instances) of existing intrinsic WE evaluation tasks for word similarity and categorization. In addition, we perform a comprehensive evaluation of three incremental WE methods: IWCM, ISG, and incremental continuous bag-of-words (ICBOW).

The next sections detail the selected models' technical problems and standardization challenges.

1.1 Problem Statement

In recent years, the development of incremental techniques for generating word embeddings [20, 46, 61, 76, 75] has garnered significant attention from researchers. These techniques are specifically designed for processing streaming text data, characterized by theoretically infinite text sequences [3]. Furthermore, unlike conventional batch learning approaches [102], incremental techniques [31] enable the creation of word embeddings, making them advantageous for real-time analysis.

Despite significant progress in incremental word embeddings, the standardization of datasets and implementation procedures remains challenging due to their continuous learning paradigm. Furthermore, conventional evaluation criteria are not suitable for streaming scenarios, emphasizing the need for transparent evaluation criteria and benchmarking for incremental representation techniques. Finally, the lack of user-friendly interfaces is another obstacle that needs to be overcome for researchers and industry practitioners to effectively utilize these techniques.

Given the increasing prevalence of streaming environments [69, 96], such as social media, the demand for efficient and accurate incremental word embedding models [20, 46, 61, 76, 75] and accessible frameworks for their application is growing. However, applying incremental word embedding techniques to stream text data requires addressing several challenges. Ensuring the privacy and security of real-time data streams [101], adapting to multilingual or code-mixed data streams [99], addressing the phenomenon of concept drift [95], and dealing with noisy or unstructured data streams are among the significant challenges that need to be considered.

That said, whether a standard and comprehensive framework can be created that incorporates all the previously outlined challenges remains uncertain.

1.2 Research Hypothesis

The thesis posits that by harnessing the shared characteristics and features of Incremental Word Embeddings, it is possible to establish a standardized framework that aligns with the instance and incremental batch paradigm. Additionally, the study proposes expanding the intrinsic evaluation process to accommodate the unique challenges streaming text data presents during the training phase.

1.3 Objectives

1.3.1 General Objectives

This thesis aims to comprehensively analyze and compare different incremental word vector methodologies utilized for streaming text data, particularly in social media, as identified in the current state-of-the-art literature [20, 46, 61, 76, 75]. The ultimate goal is to consolidate these techniques into a cohesive, user-friendly framework for researchers and practitioners. Moreover, this study proposes a periodic evaluation scheme to assess the effectiveness and robustness of the proposed framework for long-term deployment in real-world scenarios. By doing so, this thesis aims to contribute to advancing incremental word vector methodologies and their practical applications in various fields.

1.3.2 Specific Objectives

1. Propose and evaluate different incremental word embedding algorithms for training text data in streaming scenarios, to improve the efficiency and effectiveness of these models in handling large and dynamic text streams.
2. Design a unified framework that standardizes the proposed algorithms under an incremental learning paradigm.
3. Propose an extension of the intrinsic evaluation task to evaluate the performance of the incremental word embedding models in a streaming scenario, allowing for continuous monitoring of the models over time.
4. Integrate the proposed unified framework of incremental word embedding techniques and evaluation scheme into an open-source library to facilitate the reproducibility and accessibility of the experiments conducted in this study.
5. Conduct a comprehensive comparative analysis of the performance of various incremental word embedding models using the extended intrinsic NLP tasks evaluation scheme to assess the efficacy of the proposed algorithms under various streaming text data scenarios.

1.4 Methodology

This section outlines the methodology employed to achieve the specific objectives of our research. In summary, the study comprises the following steps:

1. Conducting a literature review of word embedding techniques under streaming approaches and incremental learning [20, 46, 61, 76, 75]. The selection of models will be based on source code availability and the recency of publication [20, 46].
2. Performing a comprehensive literature review of evaluation methods for word embeddings, focusing on intrinsic NLP tasks such as similarities, analogies, and related metrics [44, 33]. We will also investigate the software tools available for these tasks [86].
3. Reproducing the implementation of the existing models IWCM [20] and ISG [46] in Python. This step aims to assess the performance of these models, understand their key parameters, and analyze their outputs.
4. Familiarize ourselves with the core concepts of incremental learning, particular instance, and incremental batch learning [81] to propose a common interface encompassing incremental word embedding techniques under the streaming paradigm.
5. Designing an evaluation scheme that extends the traditional intrinsic NLP task for static word embeddings, considering the specifics of streaming scenarios.
6. Develop a module incorporating the selected models and the new evaluation scheme into a Python package.

7. Conducting a comprehensive study of the incremental word embedding techniques to understand their strengths and weaknesses under streaming scenarios, using the evaluation scheme developed in step 5.

Overall, this methodology seeks to provide a rigorous and comprehensive evaluation of the performance of incremental word embedding techniques in streaming scenarios and to develop a framework that facilitates their application and implementation.

1.5 Research Outcome

To showcase the effectiveness of our proposed framework, we sent a paper titled *"RiverText: A Python Library for Training and Evaluating Incremental Word Embeddings from Text Data Streams"* that was accepted by the committee at the *46th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR 2023)*[1]. Our paper provides an in-depth analysis of the performance of our proposed framework and highlights its key advantages over traditional WE models.

The RiverText framework is specifically designed for training and evaluating incremental Word Embeddings from text data streams. It is a departure from traditional static WE models, as it follows an incremental methodology that is more efficient, adaptable, and capable of handling large datasets.

Our paper details the key features of the RiverText framework, including its ability to process data with limited resources, generate predictions or transformations at any time, and adapt to temporal changes. We also conducted a benchmark study to showcase the framework's effectiveness, and our results demonstrate that RiverText outperforms traditional WE models in terms of accuracy and efficiency.

Furthermore, we have made the RiverText framework available for download from its official website[2]. The framework is easy to use and has been designed with scalability in mind, making it suitable for large-scale applications. We encourage other researchers to use the framework and continue to build on our work.

We believe that the RiverText framework represents a significant advancement in natural language processing. We look forward to seeing how other researchers will use and build on our work to further push the boundaries of this exciting area of research.

1.6 Outline

The rest of the thesis is organized as follows:

[1] https://sigir.org/sigir2023/

[2] https://dccuchile.github.io/rivertext/

In Chapter 2, we comprehensively discuss the theoretical background and related work that underlies our proposed methodology.

Chapter 3 presents the RiverText framework, which encompasses the models and evaluation scheme we developed for our research. We detail the foundations of our methodology, including its design principles, architecture, and model selection.

In Chapter 4, we discuss our experimental design and present the main results from our benchmark of the RiverText models. We analyze the performance of our methodology across a range of data sets and present insights gained from our experiments.

Chapter 5 delves into the implementation details of our framework and provides a comprehensive description of the code design. We discuss the libraries and tools we use and provide guidelines for future developers who seek to utilize our approach.

Finally, in the concluding Chapter 6, we summarize our research findings and provide insights into potential avenues for future work. We discuss the limitations of our approach, suggest areas for improvement, and outline directions for future research.

Chapter 2

Background and Related Work

This chapter provides an overview of the scientific disciplines related to this work. We then delve into the core concepts of word representation and streaming learning, central to this thesis. Next, we further explore the challenges the streaming problem poses in the context of the word representation algorithms. Finally, we review the related work on word representation in the streaming setting.

2.1 Scientific Disciplines

2.1.1 Artificial Intelligence

Artificial Intelligence (AI) is a scientific field that seeks to develop intelligent machines capable of simulating human-like behavior. AI technology is leveraged to construct sophisticated systems capable of solving complex problems and making real-time decisions. To this end, AI employs various techniques, including machine learning, deep learning, predictive analytics, natural language processing, and image processing, to analyze datasets and identify patterns and correlations.

To be considered artificially intelligent, a system must possess, at a minimum, the following capabilities (as described in [50]):

- Knowledge representation: the ability to store and maintain knowledge.
- Automated reasoning: the capacity to reason based on stored knowledge.
- Machine learning: the capability to learn from its environment, primarily through data analysis.

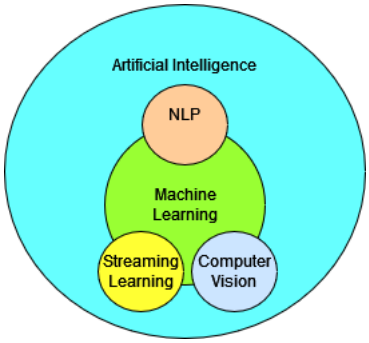

Figure 2.1: Here is a diagram illustrating the relationship between Artificial Intelligence, Machine Learning, Computer Vision, Natural Language Processing (NLP), and Streaming Learning.

2.1.2 Machine Learning

Machine learning [102] (ML) is a subfield of artificial intelligence that aims to develop algorithms and models that enable computers to learn and improve from experience. Machine learning algorithms use statistical techniques to automatically identify patterns and relationships in data without being explicitly programmed to do so. This allows machines to make predictions or decisions based on the analyzed data. The field of machine learning has rapidly evolved in recent years, driven by the availability of large amounts of data, powerful computing resources, and innovative algorithm development.

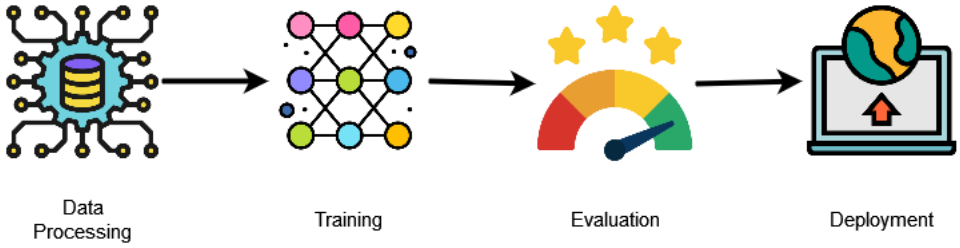

Figure 2.2: Diagram of basic Machine Learning Pipeline.

Figure 2.2 depicts the standard machine learning pipeline comprising four primary phases. The first phase is data processing, which performs all necessary transformations on raw data to obtain a standardized output. The second phase is the training phase, where the model is trained using input from the processed data to make predictions for a given task. The evaluation phase then assesses the model's effectiveness using quantitative methods. Finally, once the model can provide accurate predictions for unseen data, it is deployed for use by end-users.

The field of machine learning can be divided into several subdisciplines, including but not limited to:

- Supervised learning [24] is one of the most common forms of machine learning. A machine learning model is trained on labeled data in supervised learning, meaning an expected output or label accompanies the input data. The model then uses this

labeled data to predict new, unseen data. Supervised learning can be used for tasks such as image classification, speech recognition, and natural language processing. This approach is especially useful when there is a clear relationship between the input and output data and when the training data is well-structured.

- Unsupervised learning [11], on the other hand, involves training a model on unlabeled data and is tasked with identifying patterns or relationships within the data. This is often used when there is no clear relationship between the input and output data or when the data is unstructured. Examples of unsupervised learning include clustering, anomaly detection, and dimensionality reduction. Unsupervised learning is also used for tasks such as recommendation systems, where the goal is to identify patterns in user behavior.
- Reinforcement learning [97] is another type of machine learning that involves training a model through trial and error to make decisions in an environment where the model receives feedback through rewards or punishments. Reinforcement learning is often used in robotics, gaming, and control systems. Reinforcement learning aims to find an optimal policy or decision-making process that maximizes the reward signal. Reinforcement learning is a complex machine learning area requiring sophisticated algorithms and a deep understanding of the problem domain.

2.1.3 Deep Learning and Feedfoward Neural Network

Deep learning [53] is a subfield of machine learning that uses artificial neural networks to model and solve complex problems. It is characterized by its ability to process and learn from large volumes of data and automatically extract hierarchical features. Deep learning models comprise multiple layers of interconnected nodes, also known as artificial neurons [52], that perform mathematical operations on the input data to generate outputs.

A feedforward neural network [12], also known as a multilayer perceptron (MLP), is a type of deep learning model that consists of multiple layers of neurons arranged in a feedforward manner. The input layer receives the input data, which is then processed by the hidden layers, and the output layer generates the final output. Each neuron in a feedforward network is connected to all the neurons in the previous and subsequent layers but not to neurons in the same layer.

The neurons in a feedforward network perform a weighted sum of their inputs and apply an activation function [88] to the result to generate their output. The weights and biases of the neurons are learned during training using optimization algorithms such as stochastic gradient descent (SGD) [8]. The training goal is to find the weights and biases that minimize a given cost or loss function [94], which measures the difference between the predicted and actual output. Once the network is trained, it can be used to make predictions on new, unseen data.

Formally, it can be defined as:

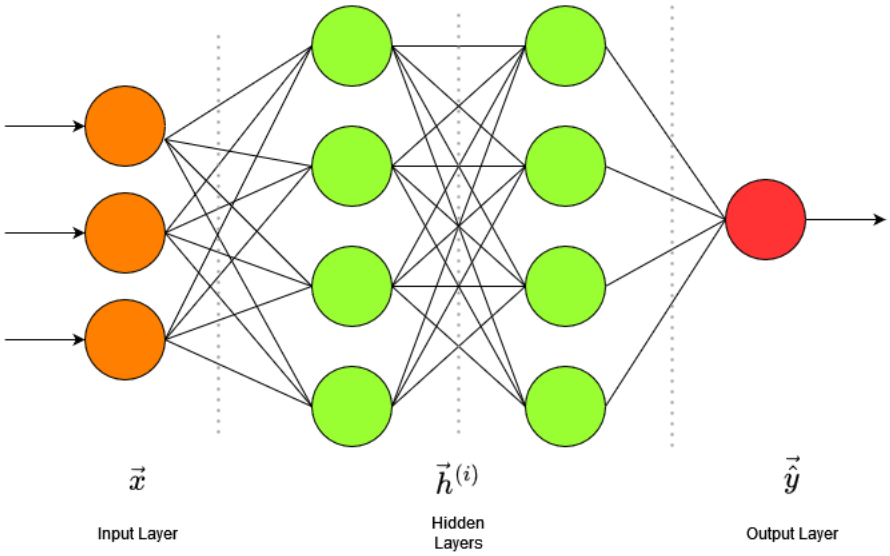

Figure 2.3: Diagram of a Multilayer Perceptron Network.

$$\begin{aligned} \vec{h}^{(0)} &= \sigma(W^{(0)} \cdot \vec{x} + \vec{b}^{(0)}) \\ \vec{h}^{(i)} &= \sigma(W^{(i)} \cdot \vec{h}^{(I-1)} + \vec{b}^{(i)}) \\ \vec{\hat{y}} &= softmax(\sigma(W^{(n)} \cdot \vec{h}^{(n)} + \vec{b}^{(n)})) \end{aligned} \tag{2.1}$$

Where σ represents an activate function, commonly relu [85], or sigmoid [79].

2.1.4 Natural Language Processing

Natural Language Processing [60] (NLP) is an interdisciplinary field that combines computer science, linguistics, and artificial intelligence to develop algorithms and models that enable machines to understand, analyze, and generate human language. NLP draws upon the principles of linguistics to understand the structures and patterns of human language. At the same time, computational techniques such as machine learning and data science are used to develop algorithms and models that can process natural language data. NLP has practical applications in chatbots, sentiment analysis, machine translation, speech recognition, and text summarization.

NLP tasks refer to the various problems that NLP seeks to solve, which can be broadly classified into three main groups:

- Text classification: An NLP task categorizes [51] a given text into predefined categories or classes based on its content. NLP models use supervised learning algorithms trained on labeled examples to learn the relationship between the input text and the corresponding categories. The trained model can then classify new, unlabeled text based on its content. Text classification has numerous practical applications, such as spam filtering, sentiment analysis, topic modeling, and language identification.

- Sequence Labeling: An NLP task [4] that involves assigning a label to each element of a sequence of tokens, such as words or characters, in a given text. Sequence labeling aims to identify and classify each token based on its context within the sequence. This task is commonly performed using machine learning models, such as Hidden Markov Models [27] and Conditional Random Fields [90], trained on labeled examples. Sequence labeling has various applications in NLP, such as named entity recognition, part-of-speech tagging, and chunking. For example, named entity recognition involves identifying and classifying entities in text, such as people, organizations, and locations. In contrast, part-of-speech tagging involves assigning a grammatical category to each word in a sentence, such as a noun, verb, or adjective.
- Sequence to sequence: an NLP task that involves mapping a sequence of tokens from one domain to another. Sequence-to-sequence aims to generate an output sequence with a different representation but maintains the same underlying meaning as the input sequence. Sequence-to-sequence models are typically implemented using Recurrent Neural Networks [53], such as Long Short-Term Memory [38] and Gated Recurrent Unit [26] networks. Sequence-to-sequence models have a wide range of applications in NLP, such as machine translation, text summarization, and conversational agents. For example, machine translation involves mapping a sequence of words in one language to a sequence of words in another. In contrast, text summarization involves generating a shorter summary of a longer text sequence.

Several approaches have been proposed to solve these NLP tasks from different perspectives:

- Rule-based systems [42] rely on predefined rules and patterns created by domain experts to understand and analyze natural language. These rules and patterns are designed to capture specific linguistic features of the input text, such as syntax, semantics, and discourse, and use them to perform various NLP tasks. One of the main drawbacks of rule-based systems is that they require significant manual effort to design and maintain the rules, making them inflexible and difficult to scale.
- Classical machine learning algorithms [45] are commonly used in NLP to build models that can learn patterns and relationships in data and perform various NLP tasks. These algorithms typically require a large amount of labeled training data, which can be time-consuming and costly. Additionally, feature engineering is often required to extract relevant features from the input text, which can be a manual and error-prone process. One of the main drawbacks of classical ML algorithms in NLP is that they may struggle with the ambiguity and variability of human language, making them less effective in certain contexts. However, these algorithms can be powerful tools when applied appropriately and achieve high accuracy in various NLP tasks.
- Deep learning [53] has revolutionized the field of NLP, with many state-of-the-art results achieved using deep learning-based architectures. These architectures include recurrent neural networks, convolutional neural networks, encoder-decoder models, and transformers. For example, domain-specific, character-level, and contextual word embeddings are often used to represent words as numerical values. Deep learning eliminates the need for hand-crafted rules or features, as the models can automatically learn

the relevant patterns and relationships from the data. However, deep learning models require large amounts of data and computing resources to achieve high performance, and their lack of interpretability can make it challenging to understand their decisions. Despite these challenges, deep learning methods continue to advance the field of NLP and offer exciting opportunities for future research.

- Large language models [64] have emerged as a recent NLP breakthrough. These models use deep learning techniques to train neural networks on massive amounts of text data, enabling them to generate human-like language and perform various NLP tasks such as translation, summarization, and question-answering. Unlike traditional deep learning models, trained on specific tasks, these models can perform multiple tasks with a single architecture, making them more versatile. However, large language models require enormous training data and computing power, making them expensive to develop and maintain. Furthermore, concerns have been raised about the potential environmental impact of training such large models and ethical considerations related to the potential misuse of the generated text. Nonetheless, using large language models is an exciting area of research in NLP and is expected to continue advancing the field.

2.1.5 Incremental and Streaming Learning

Incremental learning [2] is a machine learning technique that enables a model to learn from new data without retraining the entire model from scratch. Instead, this approach updates the existing model with new data and knowledge, allowing it to improve gradually and adapt. It is particularly useful when dealing with large or constantly changing datasets, as it reduces the computational resources required for training and updating the model.

Streaming learning [36] is a specific type of incremental learning dealing with data arriving in a continuous stream, such as sensor data or social media feeds. With streaming learning, the model is updated in real-time as new data arrives, allowing it to adapt and learn from the most recent information. This approach is especially beneficial for applications that require real-time decision-making or immediate responses to changing data patterns.

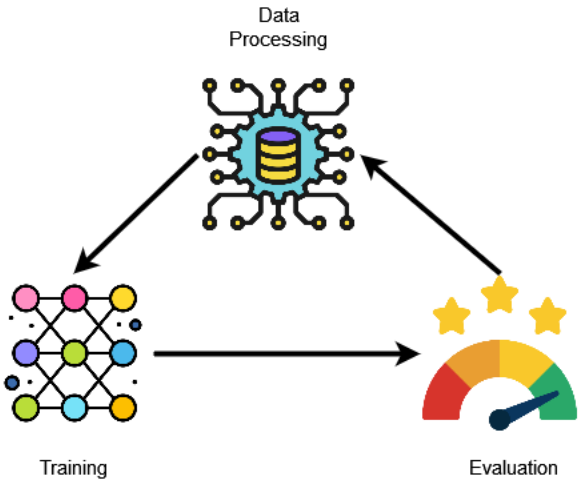

Figure 2.4: Streaming Learning Pipeline.

Figure 2.4 illustrates the streaming and incremental learning pipeline, similar to the static machine learning pipeline, in the three main data processing, training, and evaluation phases shown in Figure 2.2. The difference is that these phases work concurrently and continuously in streaming and incremental learning, as the data streams without an end.

One of the main challenges in streaming learning is dealing with the high volume and velocity of data that needs to be processed and analyzed. Specialized algorithms and techniques have been developed to address this challenge, such as online clustering [10] and feature selection [93]. These techniques enable the model to efficiently update and adapt to new data while minimizing the computational resources required.

Concept drift [95] is a phenomenon that occurs when the statistical properties of the target variable or data distribution change over time. For example, this can happen when the underlying data-generating process changes due to external factors such as seasonal variations, new trends, or shifts in customer behavior. Detecting and adapting to concept drift is crucial in incremental learning as it can significantly impact the model's performance.

Data streams [3] refers to a continuous and unbounded flow of data that arrives sequentially over time, often in high volume and velocity. Streaming learning is particularly relevant in data streams, where models must adapt and learn from new data in real time to maintain their accuracy and relevance. Formally, a data stream is any ordered pair (s, Δ) where:

- s is a sequence of tuples and
- Δ is a sequence of positive real-time intervals.

2.1.6 Instance and Batch Incremental Learning

Instance incremental learning and incremental batch learning are two approaches [81] to incremental learning.

Instance incremental learning [81, 67] is a type of incremental learning where the model is updated with each new data instance. In other words, the model is trained on each new data point as it arrives, and the parameters are updated in real-time. This approach is useful in applications where new data arrive frequently, and the model must adapt quickly. However, instance incremental learning can be computationally expensive as the model has to be retrained on each new data instance.

Batch incremental learning [22] is a type of incremental learning where the model is updated with batches of data instead of individual instances. The model is trained on a batch of data, and the parameters are updated based on the average of the gradients from the batch. This approach is more efficient than instance incremental learning as the model only needs to be updated periodically, reducing the computational resources required. However, incremental batch learning may not be as responsive to changes in the data as instance, incremental learning, as the model is updated less frequently.

2.2 Word Representation

In NLP, representing words as mathematical objects that machine learning algorithms can process is a crucial challenge due to the inherently unstructured nature of natural language text. As a result, vector representations of words and documents have become a popular solution to this problem. Nonetheless, the problem of text and document representation is an ongoing area of research with various techniques proposed to address it, each with its advantages and disadvantages.

In the subsequent sections, we review different text and document representation approaches. We commence by discussing the classic bag of words model, which represents documents as a frequency count of the words they contain. However, this model has significant semantic limitations, such as its inability to account for the order and context of words. Subsequently, we delve into how distributional representations address these issues by representing words as vectors based on the distributional hypothesis that words used in similar contexts have similar meanings.

2.2.1 One Hot Representation

One hot text representation is a binary encoding method for representing textual data. In this encoding scheme, each word or token in a corpus is represented by a unique binary vector where only one element of the vector is set to 1 while all other elements are set to 0. The length of the vector is equal to the vocabulary size, which is the total number of distinct words or tokens in the corpus. For example, if a corpus has a vocabulary of 10,000 words, each word will be represented by a vector of 10,000.

Representing a document in this model involves computing the vector representation for each word in the document and then taking the average to obtain a single vector representation. Formally, let a document be a set of words $\{w_1, w_2, ..., w_m\} \in D$, where m is the length of the document. Suppose we have a dataset of n documents $\{d_1, d_2, ..., d_n\} \in \mathcal{D}$, where $|V|$ is the number of distinct words or tokens in the vocabulary (which we will denote by V). The model consists of vectors $v \in R^{|V|}$, where each vector dimension is a one-hot encoded-word vector. For example, we can see how the word dog would be represented in formula 2.2:

$$\vec{v} = \begin{bmatrix} ant \\ ... \\ ... \\ cat \\ dog \\ dinosaur \\ ... \\ zebra \end{bmatrix} \rightarrow \begin{bmatrix} 0 \\ ... \\ ... \\ 0 \\ 1 \\ 0 \\ ... \\ 0 \end{bmatrix} \tag{2.2}$$

Then, to represent each document $d_i \in D$, the one-hot vector for each word in d_i is averaged, as shown in the formula 2.3:

$$\vec{v} = \frac{1}{|D|} \sum_{v_i \in D}^{|D|} v_i \tag{2.3}$$

One advantage of the one-hot encoding technique for representing text is its ability to accommodate documents of varying lengths in a unified vector space. However, a major drawback of this approach is its inability to capture semantic and syntactic relationships between words and documents. Furthermore, when the vocabulary size is large, the resultant high-dimensional vectors pose computational challenges for many machine learning models, as processing such vast amounts of data demands significant computational resources. These limitations have motivated the development of alternative techniques, such as word embeddings, which offer more efficient and meaningful representations of textual data, which will be discussed below.

2.2.2 Distributional Hypothesis and Distributional Representations

As we point out in the last section, the one-hot representation is a widely used technique for document representation due to its simplicity and efficiency in capturing the vocabulary of a document set. However, this technique has a fundamental limitation: the representations do not capture the meaning of words, preventing the computation of relationships between them. For instance, the one-hot representation of "cat" is entirely different from that of "dog" or "pizza," even though "cat" and "dog" are both animals. This means the model cannot recognize any underlying relationship between these words based on their meanings, thus restricting its utility for a range of natural language processing tasks.

The Distributional Hypothesis [41] has emerged as a potential solution to address this issue. This hypothesis posits that words with the same context tend to have similar meanings or connotations. Alternatively, a word can be characterized by the other words with which it is commonly associated. This idea has led to the development of Distributional Representations, which encode the meaning of words by capturing their context in vectors.

One popular method for creating Distributional Representations is using word-context matrices. In a large text corpus, these matrices capture the frequency of co-occurrences between words and their context.

2.2.3 Word Context Matrices

Word-context matrices are a class of models that aim to capture the distributional properties of words by leveraging the co-occurrences between them. Essentially, each row in the matrix represents a word, while each column represents a context word. Therefore, each entry in the matrix's (i, j) position reflects the degree of association between a word and its context. This matrix thus serves as a comprehensive record of the distributional patterns of words in a corpus. Analyzing these patterns makes it possible to derive meaningful representations of words that capture their semantic relationships with one another.

Co-ocurrence counts

One approach to computing the strength of association between a target word w_i and the words in its context c_j is to count their co-occurrences across all documents in the corpus [91]. In this method, the context is defined as a window of words surrounding the target word, and its size, k, is a parameter that the user can set. If the context's vocabulary is the same as the target words, the resulting word-context matrix will be size $|V| \times |V|$. Specifically, each word is represented as a sparse vector in a high-dimensional space, where the dimensionality corresponds to the vocabulary size. The vector represents the weighted bag of contexts in which the word appears.

For example, the Table 2.1 represents a word-context matrix for three documents:

- I like dogs.
- I like burgers.
- I have a dog.

Table 2.1: Example of a word-context matrix

	I	like	dogs	burgers	have	a	dog
I	0	2	0	0	1	0	0
like	2	0	1	1	0	0	0
dogs	0	1	0	0	0	0	0
burgers	0	1	0	0	0	0	0
have	1	0	0	0	0	1	0
a	1	0	0	0	1	0	1
dog	0	0	0	0	0	1	0

It is important to note that the number of contexts for a given target word is generally defined by a user parameter called the window size. For example, in Table 2.1, the window size was set to one, capturing the context of one word to the left and one to the right of each target word.

In the example presented in Table 2.1, the vectors associated with the words "dog" and "burgers" are the same, despite representing different things. This is a consequence of the distributional hypothesis, which suggests that words that tend to appear in similar contexts have similar meanings. However, this issue can be addressed by adjusting the window size and working with larger documents.

The count-based word-context matrix method effectively understands the semantic relationships between words based on their contexts. However, the method heavily relies on word frequencies in a corpus, which can result in unbalanced vectors. This is because common words such "and, "to," and "the" tend to capture a large number of contexts, even though they are not the most informative words in the corpus. As a result, word-context pairs like "a dog" and "the dog" receive greater importance in the model than more descriptive pairs like "big dog" and "black dog," which paradoxically convey more information.

To address this issue, we discuss the Positive Point-Wise Mutual Information.

Positive Point-Wise Mutual Information

Pointwise Mutual Information (PMI) is a statistical measure that captures the strength of the association between two words in a corpus. PMI is calculated by comparing the probability of the co-occurrence of two words with their probabilities, which is given by the formula 3.1:

$$PMI(x, y) = log_2 \left(\frac{P(x, y)}{P(x)P(y)} \right) \tag{2.4}$$

Where:

- x represents a target word.
- y represents the context associated to x.
- $P(x, y)$ the probability of the co-occurrence of x and y.
- $P(x)$ the probability of x.
- $P(y)$ the probability of y.

However, for a given corpus of text, the formula 3.1 can be reformulated to:

$$PMI(w, c) = log_2 \left(\frac{count(w, c) \cdot |D|}{count(w) \cdot count(c)} \right) \tag{2.5}$$

The values count(w, c), count(w), and count(c) represent the number of times a word-context pair (w,c), the word w, and the context word c appear in the corpus, respectively. The symbol $|D|$ represents the total number of tokens in the corpus.

Positive Pointwise Mutual Information (PPMI) is a PMI variant that addresses unbalanced vectors in the count-based word-context matrix method. PPMI assigns higher weights to rare word-context pairs and lower weights to common ones. This is achieved by subtracting the logarithm of the probability of a word-context pair occurring by chance from the logarithm of its observed frequency.

Since PPMI is a variant from the formula 2.5 is calculated as follows:

$$PPMI(w, c) = max(PMI(w, c), 0) \tag{2.6}$$

The max function ensures that negative values are set to zero, eliminating the influence of negatively correlated pairs.

Using PPMI, rare word-context pairs that convey more information about the meaning of words receive higher weights, while frequent, less informative pairs receive lower [25]. This results in a more balanced and informative word-context matrix, which can be used to train better machine learning models for natural languages processing tasks such as text classification, sentiment analysis, and machine translation.

Problems of the word-context matrix methods

While word-context matrices provide better semantic representations than one-hot vectors, they suffer from the challenge of high dimensionality. Working with and storing these matrices is memory-intensive, and self-learning classification models struggle with such high-dimensional inputs.

To address this issue, dimensionality reduction techniques such as Principal Component Analysis [1] can reduce the representations' dimensionality.

Principal Component Analysis

Principal Component Analysis (PCA) [1] is a widely used technique for dimensionality reduction in machine learning. Its goal is to find a lower-dimensional representation of a high-dimensional dataset that captures the most important information.

Given a dataset $X \in \mathbb{R}^{n \times p}$ with n data points and p features, the PCA algorithm can be described as follows:

First, the data is normalized by subtracting the mean vector $\vec{\mu} \in \mathbb{R}^p$ from each feature so that the data is centered around 0:

$$\bar{X} = X - \vec{\mu}$$

Next, the covariance matrix of the normalized data is calculated, which measures the degree to which the features co-vary:

$$C = \frac{1}{n-1} \bar{X}^T \bar{X}$$

The eigenvectors $\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_p$ of the covariance matrix C are then computed, along with the corresponding eigenvalues $\lambda_1, \lambda_2, \ldots, \lambda_p$. These eigenvectors represent the principal components of the data, and the eigenvalues represent the variance of the data in the direction of the eigenvectors:

$$\mathbb{C}u_i = \lambda_i \vec{u}_i, \quad i = 1, 2, \ldots, p$$

The eigenvectors are usually sorted in descending order based on their corresponding eigenvalues so that the first few eigenvectors capture the most important information in the data.

Finally, the lower-dimensional representation of the data can be obtained by projecting the normalized data onto the first k eigenvectors:

$$Z = \bar{X} U_k,$$

where $U_k = [\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_k]$ is the matrix containing the first k eigenvectors, and $Z \in \mathbb{R}^{n \times k}$ is the resulting lower-dimensional representation of the data.

It is important to note that traditional PCA follows a batch-learning approach, which may not be suitable for streaming environments. In such scenarios, Incremental PCA [9] plays a more prominent role in dimensionality reduction, as it considers sources of information in the form of data streams. This algorithm will be important in the next chapters.

Incremental PCA

Incremental PCA [83] is a variation of the traditional PCA algorithm that allows for efficient computation of principal components in scenarios where the data arrives in a stream or when dealing with large datasets that cannot fit entirely in memory.

In the traditional PCA, the entire dataset is required to compute the covariance matrix, which can be computationally expensive and memory-intensive for large datasets. Incremental PCA addresses this limitation by processing the data incrementally, one mini-batch or data instance at a time, and updating the principal components iteratively as new data arrives.

Generally, the incremental PCA algorithm typically follows these steps:

- **Initialize:** Start with an initial estimate of the principal components. This can be an empty set or a reduced-rank approximation obtained from a smaller subset of the data.
- **Stream data**: Process the data in small batches or individual instances. For each batch, update the estimates of the principal components based on the new data.
- **Update covariance matrix:** Compute the covariance matrix incrementally using the new batch data and update the running estimates of the principal components.
- **Compute principal components:** Use the updated covariance matrix to compute the principal components. This can be done using singular value decomposition (SVD) [40] or eigenvalue decomposition.

- **Repeat:** Stream the data, update the covariance matrix, and recompute the principal components until convergence or a desired number of iterations.

One advantage of incremental PCA is that it can handle streaming data or large datasets with limited memory resources. In addition, it avoids storing the entire dataset in memory, making it more scalable and suitable for real-time applications.

However, it's important to note that incremental PCA may introduce some approximation errors compared to the traditional PCA algorithm that operates on the entire dataset. This is because the approximation quality depends on the mini-batch size and the number of iterations performed during the streaming process.

Another direction the natural language processing community has taken is to use distributed representations. These representations encode meaning in a lower-dimensional space, with each dimension representing a specific semantic feature. Distributed representations enable the efficient processing of large amounts of text and are now commonly used in neural network-based models for various natural language processing tasks.

2.2.4 Distributed Representation or Word Embeddings

Distributed representations [63], also known as word embeddings, are a collection of models that capture the meaning of words by mapping them to dense, continuous vectors with low dimensionality. These vectors are based on the distributional hypothesis, meaning that they represent words based on their contextual usage. Therefore, words frequently appearing in similar contexts will have similar vector representations.

Word embeddings are typically trained using neural networks on large corpora of documents. During training, the semantic meaning of words is spread across the dimensions of the vectors, creating distributed representations. Although the dimensions of these vectors are not easily interpretable, these models are generally more powerful than previous count-based models.

Despite their lack of interpretability, word embeddings have become a central component in many systems due to their ability to improve performance on various NLP tasks. Their success has led to the development of many different algorithms for training word embeddings and incorporating them into NLP models.

Obtaining Word Embedding Models

There are two main approaches for obtaining word embeddings:

1. Embedding layers: This approach uses an embedding layer in a task-specific neural network architecture trained from labeled examples, such as sentiment analysis. This approach allows the embeddings to be explicitly optimized for a downstream task.

2. Pre-trained word embeddings: This approach involves creating an auxiliary predictive task from unlabeled corpora, such as predicting the next word in a sentence, in which word embeddings will naturally arise from the neural network architecture. Large-scale pre-trained language models like OpenAI GPT and BERT are particularly effective at this approach. The resulting embeddings can then be used in downstream tasks, possibly with fine-tuning.

These approaches can be used in isolation but can also be combined. For example, one can initialize an embedding layer of a task-specific neural network with pre-trained word embeddings obtained with the second approach. This can improve the model's performance, especially when limited to labeled training data.

Word2Vec

Word2Vec is a neural network-based model for learning distributed representations of words. It was developed by Mikolov et al. [63]. The model is trained on large amounts of text data to learn vector representations of words in a continuous vector space, where each dimension represents a particular feature.

Word2Vec uses two architectures for learning word embeddings: Continuous Bag-of-Words (CBOW) and Skip-gram [63]. In the CBOW architecture, the model predicts the current word given a context of surrounding words. In contrast, in the Skip-gram architecture, the model predicts the surrounding words given the current word. Both architectures use a shallow neural network with a single hidden layer to learn the vector representations.

The optimization method used in Word2Vec is based on SGD [8]. The objective function is to maximize the likelihood of the observed word-context pairs in the training data. This is achieved by minimizing the negative log-likelihood [73] of the observed pairs. The negative log-likelihood is equivalent to the cross-entropy loss [56] between the predicted and observed probabilities of the word-context pairs.

To speed up the training process and reduce memory requirements, Word2Vec uses a technique called negative sampling [35]. Negative sampling involves randomly selecting a small number of negative samples for each observed word-context pair and updating the model only on these pairs. This is more efficient than updating the model on all possible word-context pairs, which can be computationally expensive.

Another optimization method used in Word2Vec is hierarchical softmax [68]. Hierarchical softmax reduces the computational complexity of computing the softmax probability distribution by organizing the vocabulary into a binary tree. This allows for faster computation of the probability distribution and faster model training.

It is worth noting that the term "Word2Vec" is often used interchangeably to refer to both the software and the pre-trained models provided by its creators. The pre-trained Word2Vec embeddings available for use are based on the skip-gram model with negative sampling, one of the two architectures used by Word2Vec to learn word embeddings.

Skip-Gram Model

The skip-gram model [63] is a popular technique used in natural language processing to generate word embeddings. It involves training a shallow neural network that consists of a single hidden layer with no activation function. Next, the network is trained to predict the context words (words in a context window) given a central word that shifts along the training corpus. During training, the network learns co-occurrence statistics between central and context words. When the training process is finished, the resulting neural network weights are used as vectors for the token inside a vocabulary.

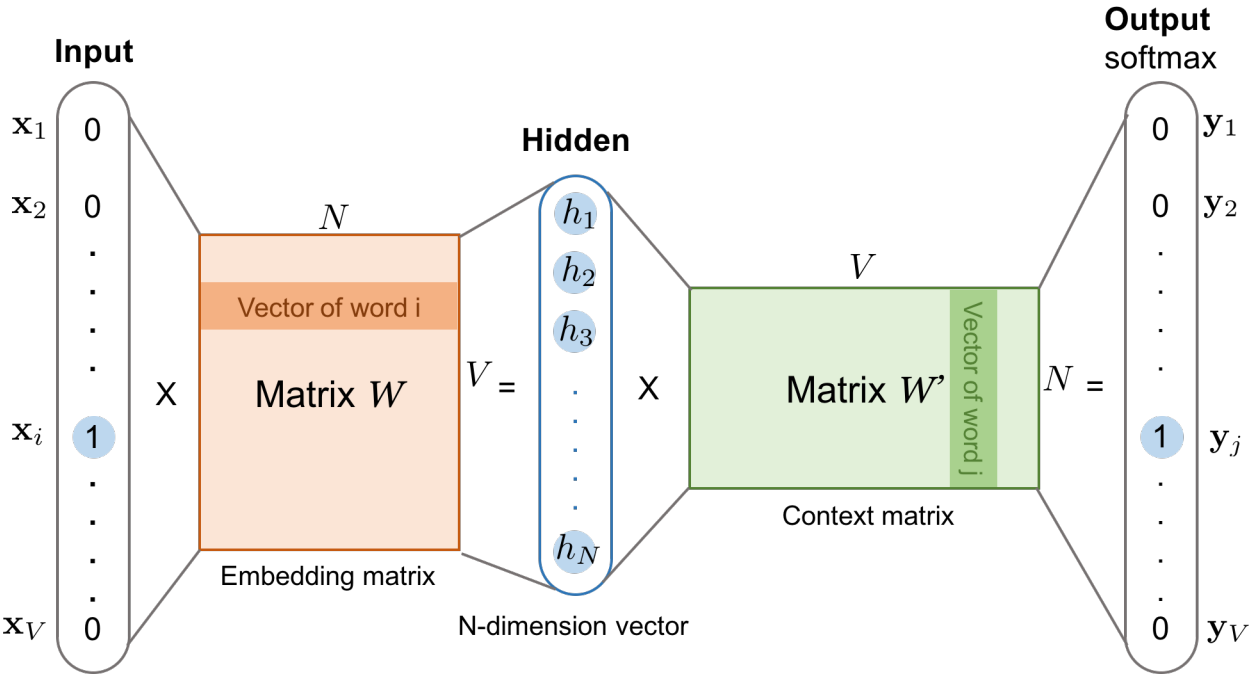

Figure 2.5: Diagram of Skip-Gram model.

In Figure 2.5, we have a high-level description of the skip-gram architecture. Subsequently, we will delve into the architecture and training process.

The neural network takes as inputs the central word and the surrounding words within a fixed window size of k. Each word in the vocabulary of size V is represented as a one-hot encoded vector, which serves as the input and output layer of the network. The hidden layer comprises N neurons with $|V|$ parameters, one for each possible input word.

The output layer has $|V|$ neurons, each with N weights, corresponding to the possible output words in the vocabulary. The activation function used in the output layer is softmax, which computes the probability distribution over all possible output words given the input word.

The entire training corpus is used to iterate over central words and their context windows to train a neural network for learning word embeddings. The central word and hidden layer information are then used to predict the context words. Finally, these predictions are compared to the actual context words to adjust the network weights via backpropagation

[8]. This training process aims to create a useful and meaningful distributed representation of the words in the corpus.

After training, the hidden layer weight matrix is extracted as the word embeddings. These embeddings capture relationships between words in the corpus and can be used as features for various natural language processing tasks, such as language modeling and text classification. Using word embeddings can significantly enhance the efficiency and accuracy of such tasks.

While the output weight matrix does contain contextual information about the words, it is often not used in this model. Instead, it is discarded or used for other purposes, such as clustering analyses or word similarity calculations.

Let's consider a document corpus consisting of a sequence of words: $w_1, w_2, w_3, ..., w_t$, and a size window k. We use the letter w to represent the target word, and the letter c to represent the context words. For instance, the words in the context $c1 : k$ of the target word w_t are denoted by $(w_{t-k/2}, w_{t-1}, w_{t+1}, ..., w_{t+k/2})$ (assuming k is even).

The Skip-gram model aims to maximize the average log probability of the context words given the target words. In other words, the model tries to learn how likely it is for each context word to appear in the vicinity of the target word, which is given by the formula 2.7:

$$\mathcal{L}_{\mathcal{SG}} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{|j|\leq c}^{n} \log p(w_{i+j}|w_i) \tag{2.7}$$

Where w_i is a target word, and w_{i+j} is a context word, the context words are obtained by shifting a sliding from left to right a window of size $2w$. The expression $\log p(w_{i+j}|w_i)$ represents the probability that the word w_{i+j} is a context of the word w_i.

$$p(w_{i+j}|w_i) = \frac{\exp(\vec{t}_{w_i} \cdot \vec{c}_{w_{i+j}})}{\sum_{w\in\mathcal{V}} \exp(\vec{t}_{w_i} \cdot \vec{c}_w)} \tag{2.8}$$

Where $\vec{t}_w$ and $\vec{c}_w$ represent the target and context words, $\mathcal{V}$ represents the vocabulary set.

While maximizing the above function is believed to lead to good embeddings, it poses a significant computational challenge. The reason is that computing $P(c_j|w)$ is computationally expensive since the summation over all context words, which is typically a large number, involves the exponential function $\exp(\vec{t}_{w_i} \cdot \vec{c}_w)$.

To overcome this challenge, the skip-gram model has two variations: hierarchical softmax and negative sampling. This thesis will only describe negative sampling, while hierarchical softmax will not be discussed.

Skip-Gram with Negative Sampling

The Skip-gram model with Negative Sampling [34, 63] is a neural network architecture that learns high-quality word embeddings from a large corpus of text data. The objective of this model is to predict the context words given a target word or vice versa.

In the Skip-gram model with Negative Sampling, a target word is represented by a vector of real-valued numbers, called an embedding. Similarly, each context word is represented by a separate embedding vector. During training, the model learns to update these embeddings to maximize the probability of correctly predicting the context words for a given target word or vice versa.

To achieve this, the model uses negative Sampling to distinguish between "good" and "bad" word-context pairs. For a given target word, the model randomly samples a small number of negative examples (i.e., words that do not appear in the context of the target word).

The probability of a word-context pair (w, c_i) being positive or negative is determined using a sigmoid function:

$$P(C = 1|w, c_i) = \frac{1}{1 + \exp(\vec{t}_w \cdot \vec{c}_{w_i})} \tag{2.9}$$

where $\vec{c}_{w_i}$ and $\vec{t}_w$ are the embeddings of the context word c and the target word w, respectively, the dot product of these embeddings is transformed into a probability score using the sigmoid function.

The objective function to be optimized is to maximize the probability of correctly classifying positive examples as positive and negative examples as negative. This function is computed using the following formula:

$$\mathcal{L}_{\mathcal{SG}} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{|j|\leq c}^{n} \psi^{+}_{w_i,w_{i+j}} + k\mathbb{E}_{v\sim q(v)}[\psi^{-}_{w_i,v}] \tag{2.10}$$

Where $\psi^{+}_{w,v} = \log\sigma(\vec{t}_w \cdot \vec{c}_v)$ and $\psi^{-}_{w,v} = \log\sigma(-\vec{t}_w \cdot \vec{c}_v)$. $\sigma(x)$ correspond to the sigmoid function. The negative samples are drawn from a probability distribution q(v) called the unigram table. This distribution $q(v)$ is built from a corpus according to the frequency $f(v)$ for each v in $\mathcal{V}$, therefore $q(v) \propto f(v)^{\alpha}$. α is a smoothing parameter between 0 and 1 $(0 < \alpha \leq 1)$.

The first term in the objective function 2.10 aims to maximize the probability of correctly classifying positive examples. In contrast, the second term aims to minimize the probability of incorrectly classifying negative examples. The embeddings are updated during training using backpropagation to maximize this objective function 2.10.

Continuous Bag of Words

The Continuous Bag-of-Words (CBOW) [34, 63] model is a neural network architecture that learns high-quality word embeddings from a large corpus of text data. The objective of this model is to predict a target word given the context words surrounding it.

In the CBOW model, each word is represented by a vector of real-valued numbers, called an embedding. For example, let the vocabulary size be V and w_i be the ith word in the vocabulary. Then, the embedding for the ith word is a d-dimensional vector denoted as v_i.

Given a target word w_t and its context words $c_{t-m}, ..., c_{t-1}, c_{t+1}, ..., c_{t+m}$, where m is the context window size, the objective of the CBOW model is to maximize the probability of correctly predicting the target word for a given context. To achieve this, the model takes the embeddings of the context words as inputs and computes the average of these embeddings. The resulting vector is then passed through a feedforward neural network with a single hidden layer, which outputs a probability distribution over all the words in the vocabulary. In Figure 2.6, we have a high-level description of the CBOW architecture.

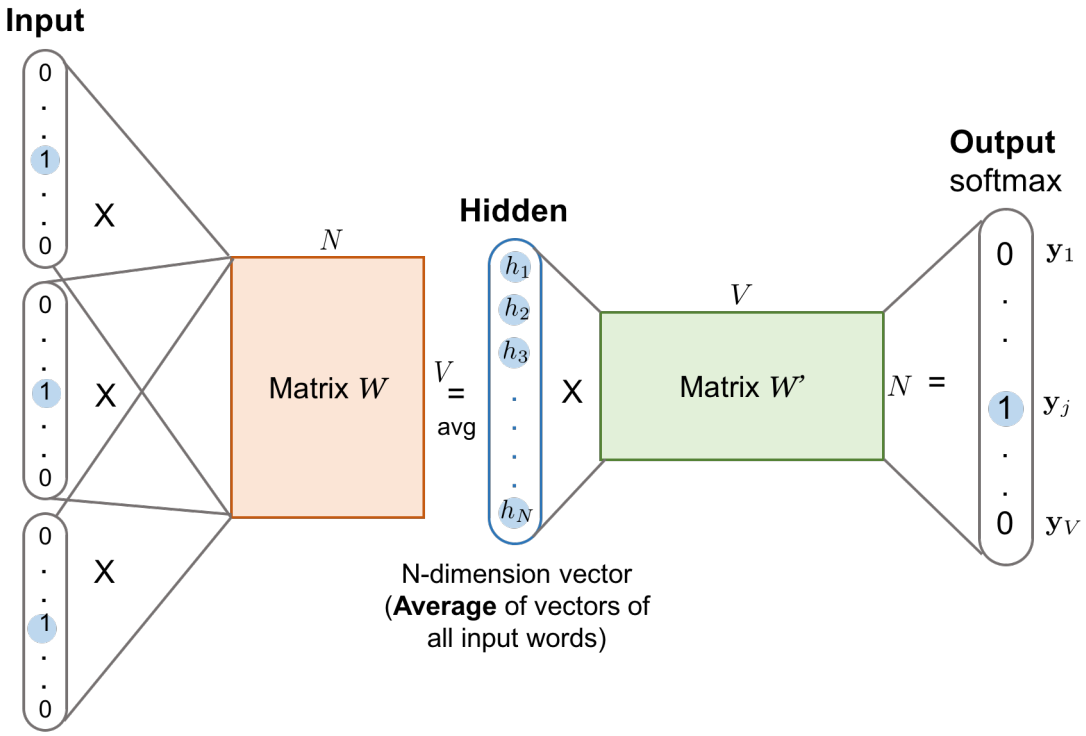

Figure 2.6: Diagram of CBOW model.

Formally, let $\vec{c}$ be the context vector computed as the average of the embeddings of the context words (i.e., $\vec{c} = \frac{1}{2m} \sum_{i=1}^{2m} v_{t-m+i}$), where v_i is the embedding vector of the ith context word. The probability distribution over the vocabulary is then computed as follows:

$$P(w_t|w_{t-m}, ..., w_{t-1}, w_{t+1}, ..., w_{t+m}) = \text{softmax}(Uc + b) \quad = \frac{\exp(u_{w_t}^T c + b_{w_t})}{\sum_{i=1}^{V} \exp(u_i^T c + b_i)} \tag{2.11}$$

Where $U \in \mathcal{R}^{|V| \times d}$ is a matrix of weights, $\vec{b} \in \mathcal{R}^{|V|}$ is a bias vector, u_i is the ith row of U, and b_i is the ith element of $\vec{b}$. The softmax function converts the neural network output into a probability distribution over all the words in the vocabulary.

The objective function to be optimized is to maximize the probability of correctly predicting the target word given its context. This function is computed using the following formula 2.12:

$$L = -\log P(w_t | c_{t-m}, ..., c_{t-1}, c_{t+1}, ..., c_{t+m}) \tag{2.12}$$

During training, the objective function is optimized through backpropagation to update word embeddings. However, CBOW and Skip-Gram models encounter similar computational issues when processing large vocabularies. Therefore, CBOW can also be optimized through Negative Sampling and hierarchical softmax discussed previously.

2.2.5 Other methods

GloVe

The GloVe (Global Vectors) [77] method is a word embedding technique that combines the advantages of both word-context matrix and distributed embedding methods. It is based on the observation that ratios of word-word co-occurrence probabilities have a semantic meaning and can be used to learn word embeddings.

The first step in the GloVe method is to create a word-context matrix that captures the co-occurrence probabilities of words in a given context window. Let X be a word-context matrix of size $V \times V$, where V is the vocabulary size. Each entry X_{ij} of the matrix is the number of times word j appears in the context of word i.

The second step is to define a word embedding function that maps each word w_i to a low-dimensional vector $\vec{v}_i$. Let $\vec{v}_i$ be the embedding vector of word i, and let $\vec{u}_j$ be the context vector of word j. The aim is to find the embeddings such that the dot product of a word and its context vector is proportional to the log of their co-occurrence probability:

$$\vec{u}_j^T \vec{v}_i \propto \log(Xij)$$

The GloVe method introduces a weighting function $f(X_{ij})$ that assigns a weight to each co-occurrence count to achieve this goal. The weighting function is defined as follows:

$$f(X_{ij}) = \begin{cases} (X_{ij}/x_{max})^\alpha & \text{if } X_{ij} < x_{max}\ 1 \\ \text{otherwise} \end{cases}$$

where x_{max} is a maximum co-occurrence count, and α is a weighting parameter that controls the weight given to rare vs. frequent co-occurrences. The choice of x_{max} and α depends on the specific application and can be determined through experimentation.

Using this weighting function, the GloVe method defines a loss function that measures the difference between the dot product of a word and its context vector and the logarithm of their co-occurrence probability:

$$J(\vec{u}_j, \vec{v}_i, \vec{b}_j, \vec{b}i) = f(Xij)(\vec{u}_j^T \vec{v}_i + \vec{b}_j + \vec{b}_i - \log(Xij))^2$$

where $\vec{b}_i$ and $\vec{b}_j$ are bias terms for word i and context word j, respectively. The goal is to minimize this loss function using gradient descent to learn the word and context embeddings.

2.3 Intrinsic NLP Tasks

Evaluating the quality of word embeddings poses a significant challenge due to various factors, and there is currently no standardized evaluation methodology in place. The field of word embedding is still open, and researchers are continuously working to develop more reliable and standardized methods for evaluating their performance. Some of the reasons why it is not straightforward to evaluate word embeddings include the following:

1. Lack of gold standards: No universally accepted gold standard for evaluating word embeddings exists. Different evaluation metrics and datasets may be appropriate for different tasks, and researchers may have different opinions on which metrics are most important. This can make it difficult to compare the performance of different word embedding models.
2. Context dependency: Word embeddings depend highly on the context in which they are learned and used. Therefore, their quality may vary depending on the text corpus used for training, the parameters used for the model, and the specific context in which they are applied. Defining what constitutes a "good" word embedding model is challenging.
3. Difficulty in defining what word embeddings should capture: There is no clear consensus on what word embeddings should capture. While some researchers focus on capturing semantic relationships between words, others may be more interested in capturing syntactic relationships. Defining what constitutes a "good" word embedding model can be difficult.
4. Subjectivity: The evaluation of word embeddings can also be subjective, as different researchers may have different criteria for evaluating the quality of word embeddings based on their research interests and goals.
5. Dependence on downstream NLP tasks: In many cases, the performance of word embeddings may be evaluated by their performance on downstream NLP tasks such as sentiment analysis or text classification. However, the performance of these tasks may be influenced by factors other than the quality of the word embeddings themselves, such as the quality of the training data or the specific algorithms used for the task.

There are two main categories of evaluation schemes for word embeddings: extrinsic and intrinsic evaluation. In extrinsic evaluation, word embeddings are utilized as input features

for a downstream task, and the resulting changes in task-specific performance metrics are measured. Examples of such tasks include part-of-speech tagging and named-entity recognition, which are not the focus of this thesis. On the other hand, intrinsic evaluation involves directly testing for syntactic or semantic relationships between words. These tasks typically entail selecting a predefined set of query terms and evaluating the corresponding semantically related target words.

This thesis focuses on evaluating word embeddings through intrinsic NLP tasks. These tasks can be classified into four categories:

- **Relatedness o word similarity**: Word similarity [32] measures the degree of relatedness or likeness between two words. It is often calculated based on the similarity of their word embeddings, which are vector representations of words learned from a large corpus of text. For example, consider the words "car" and "automobile." These two words have a high degree of similarity, as they refer to the same type of vehicle. When represented as word embeddings, the "car" and "automobile" vectors would have a high cosine similarity, indicating their semantic similarity. On the other hand, the words "car" and "banana" have a low degree of similarity, as they are unrelated in meaning and would have dissimilar word embeddings.
- **Analogy**: The word analogy task evaluates word embeddings by testing their ability to capture semantic relationships between words through arithmetic operations on their embedding representations. The task involves solving analogies like "a is to b as x is to y," where a, b, x, and y are different words. The most famous example of this task was demonstrated by Mikolov et al. [18], who showed that by operating "king - man + woman" on the embeddings of these words, the resulting vector was most similar to the embedding of "queen."
- **Categorization**: The categorization task [44] involves clustering words based on their semantic meaning to group them into different categories. This is accomplished by clustering the corresponding word vectors of all the words in a given dataset and then evaluating the purity of the resulting clusters concerning the labeled dataset.
- **Selectional preference**: The selectional preference task [33] focuses on determining the typicality of a noun for a given verb, either as a subject or as an object. For instance, in the sentence "people eat," the noun "people" is a typical subject for the verb "eat," whereas in the sentence "we rarely eat people," the noun "people" is not a typical object for the verb "eat."

2.4 Streaming in Word Embedding models

Streaming learning in word embeddings models [20, 46, 61, 76, 75] involves continuously updating the word embeddings as new data becomes available. This is achieved through incremental updates to the embedding vectors based on each new data point. This approach has several advantages over traditional, offline word embedding models such as:

- Adaptability to changing data: Streaming learning allows the model to adapt to changes in the data distribution over time. This means the embeddings can remain relevant and accurate even as new words and phrases enter the language.
- Real-time processing: Streaming learning enables real-time data processing, allowing the model to handle large volumes of data as they arrive.

However, streaming learning in word embeddings models can also pose some challenges:

1. Memory and computational requirements: As the model continuously updates its embeddings, it requires additional memory and computational resources to store and process the data. Making the model slower and more computationally intensive [22].
2. Concept drift: Streaming learning models may experience concept drift [95], where the meaning of a word changes over time. For that reason, the embeddings become outdated or inaccurate.
3. Data imbalance: Streaming learning models may also be affected by data imbalance [57], where certain words or phrases are overrepresented. Leading to bias in the embeddings and affecting their accuracy.

However, various word embeddings have been proposed to tackle the streaming and incremental paradigm challenges. Nevertheless, this is still a nascent and dynamic area of research due to the absence of standardization in evaluation methods, benchmark datasets, and software implementation. Moreover, the lack of such standardization makes it challenging to leverage incremental word embeddings to their full potential in real-world natural language processing applications in both academic and industrial settings.

2.5 Related Work

In this section, we review the literature on the three main aspects on which this work is based: 1) incremental WE models, 2) stream machine learning libraries, and 3) intrinsic evaluation of WE. First, we cover the models implemented in our framework and others, such as Incremental GloVe [77], which is not added to our library but will be included in the next version. Second, we review the main libraries the research community uses for machine learning of data streams. Finally, we discuss intrinsic evaluation approaches for WE.

2.5.1 Incremental Word Embedding Models

As we pointed out previously, WE can be divided into count-based approaches, and distributed methods based on the distributional hypothesis [41] (i.e., words appearing in the same contexts tend to have similar meanings). According to this classification, we implemented the following models discussed below:

The Incremental Word Context Matrix model (proposed by Bravo-Marquez et al. [20]) is a count-based method that constructs a word-context matrix of size $V \times C$, where V is the number of words contained in the vocabulary and C is the number of contexts around the target words (obtained from a surrounding window of fixed size). Each matrix cell encodes the association between a target word and a context, computed using a smoothed positive point-wise mutual information (PPMI) score [60]. To keep memory usage constant throughout the stream, it is necessary to keep fixed the number of words composing the vocabulary, the contexts, and the counters calculating the PPMI weights. When there is no space for a new word or context, the existing ones are replaced according to a given criterion (less frequent word/context, older word/context, among others).

Incremental skip-gram with negative sampling (ISG) is based on the neural network architecture proposed by Kaji and Kobayashi [46]. This model is inspired by the original skip-gram from the Word2Vec library (Mikolov et al. [63]). Kaji and Kobayashi develop an incremental version of negative sampling; their algorithm builds a unigram table that incrementally updates the words frequencies and the noise distribution. The authors use the Misra Gries algorithm [65] to allocate words dynamically in a finite vocabulary using constant memory throughout the stream.

Other methods not implemented yet but to be added in the future are:

SpaceSaving Word2Vec (SSW) is a work similar to ISG but developed independently by May et al. [61]. The main differences are:

- ISG uses the Misra Greis algorithm [65] for dynamic word allocation, while SWW employs the Space Save algorithm [62], which counts the most frequent elements in a data stream.
- SSW uses the original unigram sampling table to estimate the negative distribution; Kaji and Kobadashi proposed an original algorithm [46] for this purpose.

The incremental Glove [76] model follows the same idea as the original GloVe [77]; calculates the global statistics with a word-context matrix of co-occurrences, and reduces the matrix dimensionality by training a square-root loss function. The main difference is that the incremental version modifies the loss function into a recursive scheme that depends on old and new data.

2.5.2 Stream Machine Learning Libraries

As discussed above, in the stream machine learning setting, models learn continuously from data streams that evolve over time. This learning can typically be done in two ways: 1) training one example at a time or 2) training by mini-batches of examples. An important difference with the standard machine learning paradigm is that stream models cannot perform data preprocessing operations that require full access to the data (e.g., vocabulary extraction). Note also that stream machine learning is very similar to the incremental or

online learning paradigms in machine learning, but incorporates some additional constraints, such as those listed in Section 1.

Massive Online Analysis (MOA), developed by Bifet et al. [15], is a Java software package that implements numerous machine learning algorithms for training and evaluation from evolving data streams. In addition, Bifet et al. developed MOATweetReader [16], an extension to MOA for analyzing tweets in real time, detecting changes in word distribution, performing summary statistics, and sentiment analysis.

River [67] was formed from the union of two similar predecessor projects, Creme [39], and scikit-multiflow [66], which provides Python implementations of the main machine learning algorithms for data streams for tasks such as classification, regression, and clustering, as well as other functionalities. In standard machine learning, multidimensional arrays are typically used as the primary data structure for data representation. However, since streaming data can come up at any time, River uses dictionaries as a more flexible and faster alternative. To optimize mathematical operations between dictionaries, River relies on its own dictionary data structure, called `VectorDict`, which is implemented in Cython [13].

Note that none of these libraries are designed to perform representation learning for unstructured data, such as word embeddings, in an incremental fashion.

2.5.3 Intrinsic Evaluation

WE intrinsic evaluation is a family of evaluation techniques for measuring the syntactic and semantic properties captured by these vectors that include three types of tasks: word similarity (i.e., whether the similarity between two words vectors correlates with a human judgment of relatedness), analogies (i.e., when relations in the form of "a is to b as c is to d" can be obtained from arithmetic operations on the vectors), and categorization (i.e., when groups of words are aligned with predefined categories, such as animals). These evaluations are often combined to benchmark different WE algorithms, the corpora on which they are trained, and the hyperparameter settings [86]. The Word Embeddings Benchmark[1] [44] is an open-source package that brings together all intrinsic evaluations into a unified interface to facilitate the evaluation and comparison of these resources. However, this evaluation approach has its detractors, Gladkova and Drozd [33] argued that intrinsic evaluation ignores key features of distributional semantics (e.g., polysemy), and does not always correctly determine how a word embedding would perform in a downstream application.

It is important to note that these evaluations are designed for a standard machine learning setting (i.e., the evaluation is performed after the training is completed). In this work, we attempt to adapt them to a streaming setting.

[1] https://github.com/kudkudak/word-embeddings-benchmarks

Chapter 3

RiverText Foundations

As mentioned in previous chapters, incremental WE training is the process of learning dynamic word vectors from continuously arriving streams of text, such as tweets. The overall process in which these vectors are trained in our framework is as follows:

1. Connect to a continuous source of a text data stream (e.g., Twitter).
2. Tokenize the text and traverse its words.
3. If a new word is found, it is added to the vocabulary and a new vector is assigned to it.
4. If the word is known, its corresponding vector is updated according to its context (i.e., its surrounding words).
5. At any time during training, getting the vector associated with a vocabulary word is possible.

RiverText users can modify the incremental word embedding learning algorithm (depending on available implementations) and select a sketching algorithm to dynamically allocate new words to the vocabulary. This is an important component of our framework since it is impossible in a streaming setting to determine the vocabulary size in advance, as it is usually done in standard machine learning. The current version of the software only implements Misra Gries' algorithm [65], but we plan to add more algorithms in the future, such as the Space Saving algorithm [62].

This chapter discusses the principal components utilized to standardize and adapt the incremental WE algorithms, such as the Misra Gries algorithm, and the incremental learning paradigms employed. We then present our evaluation scheme for assessing the performance of the incremental WE model using intrinsic NLP tasks. Finally, from a conceptual perspective, we review the specific characteristics of each incremental WE model used to adapt our library.

3.1 Misra Greis Algorithm

The Misra-Gries algorithm is a streaming algorithm for frequent item counting in a data stream, which Misra and Gries [65] proposed. The algorithm maintains a set of counters, each associated with an item in the stream. When a new item is read from the stream, if the item is already in the set of counters, its corresponding counter is incremented. Otherwise, if the counters are not yet full, a new counter is added to the set with an initial value of 1. If the counters are already full, all counters are decremented by 1, and any counters that reach 0 are removed from the set.

At the end of processing the stream, the algorithm returns the set of counters, which should contain only the items that frequently occur in the stream.

The Misra-Gries algorithm has a time complexity of $O(nk)$, where n is the length of the input stream and k is the number of distinct items in the stream. This algorithm is commonly used in distributed systems where the data is too large to fit in a single machine's memory. However, a set of machines can each maintain a subset of the counters, which can later be combined to get the final result.

Algorithm 1: IWCM model method.

```
   Input: vocab size V, Vocab vocab, Count counter
   Output: An updated counter
 1 while batch in ST do
 2     for tweet in batch do
 3         tokens = tokenize(tweet)
 4         for token in tokens do
 5             if token not in vocab and |vocab| < v then
 6                 addToVocab(vocab, token)
 7                 updateCount(counter, token)
 8             else if token in vocab then
 9                 updateCount(counter, token)
10             if |vocab| = v then
11                 restByOneToAllCounts(counter)
12                 words = removeElemEqualOne(counter)
13                 removeKeys(vocab, words)
```

In our implementation, the Misra Greis algorithm tracks the most frequent words in the entire data stream once the vocabulary becomes full. If a token is not already in the vocabulary, we add it and set its count to one. If the word is already in the vocabulary, we increment its count by one. When the number of words in the vocabulary reaches the maximum size allowed, we decrease the count of all words by one and remove them from the vocabulary if their count becomes one, ensuring that there is always room for new words. The algorithm used for this implementation can be found in Algorithm 3.1.

As previously mentioned, we plan to incorporate additional options and algorithms for

tracking new words in our implementation, including the Space Saving algorithm [62] and Sketching techniques [71], which are known to effectively track new words in data streams.

3.2 Incremental Learning Approaches

RiverText implements two incremental learning approaches: 1) instance incremental and 2) batch incremental learning, which are discussed below.

In Instance Incremental learning, our WE parameters are updated with every training instance (e.g., a tweet) and discarded after training, as shown below:

Listing 3.1: Example of `learn_one` method using the Incremental WCM model. The parameters of the `WordContextMatrix` class are the vocab size, window size, and context size, respectively.

```
from rivertext.models import WordContextMatrix
from rivertext.utils import TweetStream

from torch.utils.data import DataLoader

ts = TweetStream("/path/to/tweets.txt")
wcm = WordContextMatrix(
    vocab_size=100000,
    window_size=3,
    context_size=1000
)
dataloader = DataLoader(ts, batch_size=1)

for tweet in dataloader:
    wcm.learn_one(tweet)
```

In this case, the text stream is simulated from a file of tweets (one tweet per line and separated by a broken line) and read from the buffer one at a time using PyTorch `DataLoader` with a batch size of 1. Then, our learning algorithm (IWCM in this case) only has to call the `learn_one` method to update its parameters accordingly.

This approach suffers from efficiency problems due to the overhead of processing one instance at a time. In addition, in the case of neural network-based models, such as ISG and ICBOW, which are based on gradient descent, loss calculations can result in inaccurate gradients [34], which can lead to requiring too many instances to obtain good word representations.

Incremental batch learning, on the other hand, gathers a small batch of instances before training. This allows neural network-based models to benefit from the increased efficiency of specialized computing architectures such as GPUs, which replace vector-matrix operations with matrix-matrix operations for forward and backward network passes. The difference with traditional batch learning is that batches can only be processed once and must be deleted once processed.

Listing 3.2: Example of `learn_many` method using the Incremental WCM model. The parameters of the `WordContextMatrix` class are the vocab size, window size, and context size, respectively.

```
from rivertext.models import WordContextMatrix
from rivertext.utils import TweetStream

from torch.utils.data import DataLoader

ts = TweetStream("/path/to/tweets.txt")
wcm = WordContextMatrix(
    vocab_size=100000,
    window_size=3,
    context_size=1000
)
dataloader = DataLoader(ts, batch_size=32)

for batch in dataloader:
    wcm.learn_many(batch)
```

As shown in Listing 3.2, our learning models only need to call the `learn_many` method to process and train a batch of instances. The text stream is also read in batches of w tweets using PyTorch's `DataLoader` with a batch size of w.

An appropriate batch of w usually depends on the available GPU memory capacity. It is important to note that the word vectors will not be updated with this approach until the batch has been processed.

3.3 Periodic Evaluation

The proposed method for evaluating our incremental WE performance is called Periodic Evaluation. This method applies a series of evaluations to the entire model, using a test dataset associated with intrinsic NLP tasks after a fixed number, p, of instances, have been processed and trained. The algorithm takes as input the following arguments:

- The parameter p represents the number of instances between the evaluation series.
- The incremental WE model, referred to as M, is to be evaluated.
- The input text data stream, referred to as TS, used to train the incremental WE model.
- A test dataset, GR, associated with intrinsic NLP tasks.

The Periodic Evaluation algorithm aims to offer a structured evaluation scheme of an incremental word embedding model throughout the training process. It provides a mechanism for continuously assessing the model's performance, thereby enabling the identification of any potential issues and offering valuable insights into the model's progress. However, it should be noted that while traditional evaluation methods for NLP tasks have been applied in static

Algorithm 2: Periodic Evaluation Algorithm. The evaluator function takes the words and their mapped vectors, and an intrinsic dataset.

```
   Input: Stream ST, Incremental WE model, Intrinsic Dataset GR, int p
1  c = 0
2  while batch in ST do
       // train the model
3      learn_many(model, batch)
       // evaluate the model during certain periods
4      if c ≠ 0 ∧ c mod p then
5          result = evaluator(model.wv, GR)
           // the result is stored in a JSON file
6          save(result)
7      c += length(batch)
```

settings, the Periodic Evaluation represents a novel approach by extending their functionality to the dynamic scenario of text streams, where the models can be trained indefinitely.

In Algorithm 2, we can observe how the periodic evaluation is implemented. In line 5, there is a function referred to as "evaluator," which takes as input the vocabulary structure, the mapped vectors, and the test dataset associated with intrinsic NLP tasks and reduces the quality of word embeddings formed into a scalar value. This function provides a quantitative measure of the quality of the word embeddings generated by the incremental WE model, allowing for a more accurate assessment of its performance.

The intrinsic tasks implemented in the proposed method are similarity, analogies, and categorization. The following evaluation metrics measure these tasks:

- **Similarity**: The Spearman correlation coefficient [98], denoted as ρ, is used to calculate the degree of association between the similarity scores calculated from the word embeddings and the scores obtained from a human-annotated dataset.
- **Analogies**: Accuracy is used to count the number of correctly obtained words from an analogy equation, comparing the set of analogy words obtained from the word embeddings with the set of analogy words from the human-annotated dataset.
- **Categorization**: Purity clustering [59] is used to count the total number of correctly classified words, comparing the categories obtained from the word embeddings with the categories from the human-annotated dataset.

The results of the evaluation metrics are stored as a JSON file that the user can access and examine at any time.

We have delegated the implementation of the evaluator for intrinsic tasks to an external library called Word Embedding Benchmark [44]. This library provides a comprehensive collection of test datasets (e.g., MEN [21], MTURK [80], AP [7]) and the corresponding methods for measuring the quality of word embeddings. The intrinsic task in question determines the specific evaluator function to be utilized. For instance, the similarity task requires using

the `evaluate_similarity` function provided by the library. For further information on this library's functionality and usage, refer to the GitHub repository[1].

It is important to note that the Periodic Evaluation method only assesses the quality of word embeddings for words present in the model's vocabulary at the evaluation time. As the vocabulary is subject to changes due to the application of the Misra-Gries algorithm for discarding infrequent words, the word embeddings for discarded words are not evaluated unless they are subsequently reintroduced to the vocabulary.

Another important consideration is that in cases where some words in the test dataset provided by the evaluator are not present in the model's vocabulary, the average embedding of the words in the vocabulary is assigned to these out-of-vocabulary words. This process is used to evaluate the quality of the model. It is crucial to note that this approach can impact the model's overall performance, and the results obtained should be interpreted with caution.

3.4 Implemented Methods

For this work, we adapted and modified three models: the Incremental Word Context Matrix (IWCM), Incremental SkipGram with Negative Sampling (ISG) and Incremental Continuous Bag of Words with Negative Sampling (ICBOW), whose details are explained next.

3.4.1 Incremental Word Context Matrix

Our implementation of the IWCM model is based on the algorithm described in the work of Bravo-Marquez et al. [20]. The IWCM model utilizes a co-occurrence matrix of dimension V x C, where V represents the number of words present in the vocabulary and C represents the number of context words associated with each target word. It is essential to note that, as opposed to its static counterpart, the co-occurrence matrix in the IWCM model may not be square due to the incremental nature of the algorithm. The relationship between a target word and its context is weighted by the Positive Pointwise Mutual Information (PPMI) score [25], a commonly used measure of association in NLP.

$$PPMI(w, c) = max\left(0, log_2\left(\frac{count(w, c) \times D}{count(w) \times count(c)}\right)\right) \tag{3.1}$$

In Equation 3.1, the variable D represents the total number of words in the text streams. The counters, $count(w_i, c_j)$, $count(w_i)$, $count(c_j)$, and D, which are used to calculate the probabilities of the word-context pairs for the PPMI score, are efficiently stored in `VectorDict` objects provided by the River packages [67]. These objects function as a sparse data structure, enabling efficient mathematical operations and incremental updates of the word-context matrix.

[1] https://github.com/kudkudak/word-embeddings-benchmarks

Algorithm 3: IWCM model method.

Input: Stream ST, window size W, vocab size V, context size C
Output: Matrix Mat V x C

```
d = 0
while batch in ST do
    for tweet in batch do
        tokens = tokenize(tweet)
        for token in tokens do
            if token not in vocab then
                addToVocab(vocab, token)
            d += 1
            contexts = getContexts(token, tokens, W)
            updateDictCounter(token, contexts)
            for cont in contexts do
                Mat(token, cont) = max(0, log((count(token,cont)·d) / (count(token)·count(cont))))
        // reduce the embedding dimension by incremental PCA
        reduceEmbDimByIPCA(tokens)
```

In Algorithm 3, a sliding window of 2W tokens is utilized to extract context information from the tokenized tweets in the text stream. The center of the window is aligned with a target word, and all surrounding tokens within the window are considered context tokens. For unseen target words and contexts, new entries are dynamically allocated in the `VectorDict` objects and initialized with a count value of zero. For existing words and contexts, the corresponding counters are updated incrementally. This approach allows for efficient storage and manipulation of the word-context matrix while maintaining an acceptable level of accuracy.

One limitation of this method is that, similar to its static counterpart, the IWCM model produces sparse and high-dimensional vectors. To address this issue, we employ the incremental Principal Component Analysis (PCA) [9] technique, to reduce the dimensionality of the generated embeddings. This algorithm does not require multiple passes over the entire set of embeddings to achieve dimensionality reduction, as it processes the data as a vector stream. In addition, our IWCM implementation selectively applies dimensionality reduction to recently added or updated embedding vectors to optimize computational efficiency.

3.4.2 Incremental Word2Vec

The incremental Word2Vec architecture comprises two ISG and ICBOW models based on the static version proposed by Mikolov et al. [63]. The ISG model predicts the context words for a given target word, and the ICBOW model aims to predict the target word using its context words.

Our implementation is based on the Skip Gram model with Negative Sampling, as proposed by Kaji and Kobayashi [46]. This implementation extends the traditional unigram

table, typically created as a static word array, to an incremental approach. Instead of performing multiple passes over the entire dataset to complete the unigram table, the model updates the table incrementally, making the process more efficient and scalable.

Algorithm 4: Adaptive Unigram Table.

Input: Array word_indexes, Array T, int size_T, Array Freqs, float α
Output: Array T

```
1  z = 0
2  for index in word_indexes do
3      Freqs[index] += 1
4      F = Freqs[index] - (Freqs[index] - 1)^α
5      z += F
6      if |T| < size_T then
7          add F copies of index to T
8      else
9          for j = 1, ..., (size_T·F)/z do
10             T[j] = index with probability F/z
```

In Algorithm 4, we present the adaptive unigram table proposed by Kaji and Kobayashi [46]. Given a fixed-size unigram table T with a capacity of $size\_T$, an array $Freqs$ representing the frequencies of the words in the vocabulary, a tuple of word indexes representing a tweet, and a smoother parameter α. The algorithm proceeds as follows:

- If the number of elements in T, $|T|$, is less than $size\_T$, F copies of the word index are added to T, where the word index corresponds to the indexes mapped to the words that compose the vocabulary.
- Otherwise, the number of copies of the word index added to T is calculated as $\frac{size\_T \cdot F}{z}$, and the new additions to T may overwrite current values with a probability proportional to $\frac{F}{z}$. This process updates the distribution of words represented in T.

It is important to note that the frequency of each word is proportional to the number of its indexes stored in T.

In Algorithm 5, the adaptive unigram method is implemented through the functions `updateTokenFreq` and `updateUnigramTable`. The two incremental word2vec models utilize this algorithm: ISG and ICBOW, with the only difference being the neural architecture used. However, a crucial preprocessing step is necessary before performing the stochastic gradient descent in line 14. This step involves converting the word indexes into the appropriate input format for the specific ISG or ICBOW models and is essential for the proper functioning of the algorithm.

The RiverText package incorporates the implementation of the neural network backend for both the ISG and ICBOW models using the PyTorch framework.

Algorithm 5: Incremental Word2Vec method

Input: Stream ST, Vocab size V, Unigram Table Size T,int num_ns

```
vocab = Vocab(V)
ut = UnigramTable(T)
while batch in ST do
    for tweet in batch do
        tokens = tokenize(tweet)
        for w in tokens do
            if w not in vocab then
                addToVocab(vocab, w)
            updateTokenFreq(w)
            updateUnigramTable(w)
            contexts = getContexts(w, tokens)
            for c in contexts do
                draw num_ns indexes from ut: ns_1, ns_2, ..., ns_num_ns
                // convert the word indexes to the neural model input
                v_w, v_c, ns_1, ns_2, ..., v_ns_num_ns = preprocessing(w, c, ns_1, ns_2, ...,
                  v_ns_num_ns)
                // performs SGD to update the word embedding
                SGD(v_w, v_c, v_ns_1, v_ns_2, ..., ns_num_ns)
```

Chapter 4

Experiments and Results

In this section, we present our benchmark results divided into three subsections. In the first part, we explain the dataset used in this work, the second part describes the experimental setup and main hyperparameters, and the last part shows our main findings.

4.1 Data

Our experiment uses a dataset of unlabeled tweets to simulate a text stream of tweets. Twitter provides an excellent source of text streams, given its widespread use and real-time updates from its users. We draw a set of ten million tweets in English from the Edinburgh corpus [78]. This dataset is a collection of tweets from different languages for academic purposes and was downloaded from November 2009 to February 2010 using the Twitter API [1]. We hypothesize that using this dataset of tweets as a text stream would allow us to evaluate the performance of incremental WE methods in a realistic scenario, given the nature of social media text and its dynamic and evolving nature.

4.2 Experimental setup

In our experimental investigation, we executed the Periodic Evaluation using diverse datasets and hyperparameter settings. Since there is not exists any methodology or benchmark dataset in the literature that establishes how to measure the performance of the incremental WE, we decided to focus on understanding how the models behave in front of different hyperparameters settings in streaming environments as a starting point. The evaluation was conducted on multiple architectural configurations (IWCM, ISG, and ICBOW) and intrinsic test datasets [44].

The hyperparameters under consideration were the size of the embedding, the window size, the context size, and the number of negative samples since they are the most com-

[1] https://developer.twitter.com/en/docs/twitter-api

mon hyperparameters among the three models proposed. The results of this evaluation provide valuable insights into the performance of the different architectural configurations and hyperparameter settings, offering a comprehensive understanding of the subject under examination.

For the intrinsic test datasets, we used two datasets from the similarity tasks (MEN [21] and Mturk [80]) and one from the categorization task (AP [7]).

4.2.1 Hyperparameter settings

The main hyperparameter configurations that we studied were:

1. We evaluated the impact of three hyperparameters on neural network embedding:
   - Embedding size: refers to the dimensionality of the vector representation associated with each vocabulary word. Our configurations considered three different embedding sizes, including 100, 200, and 300.
   - Window size: This refers to the number of neighboring tokens used as the context for a target token. Our configurations utilized three different window sizes, including 1, 2, and 3.
   - The number of Negative samples: This refers to the number of negative instances that maximize the probability of a word being in the context of a target word. Our configurations considered three different numbers of negative samples, including 6, 8, and 10.

   Therefore, our experimental investigation considered a total of 27 configurations, comprising all combinations of the hyperparameters ($emb\_size \in 100, 200, 300$, $window\_size \in 1, 2, 3$, and $num\_ns \in 6, 8, 10$) and for each of the architectural configurations and intrinsic test datasets. Table 4.1 presents all the configurations for the incremental WE models based on neural network architectures.

2. For the word context matrix embedding:
   - We leveraged the same configurations of the embedding size and window size as we did for the neural network embedding
   - Context size: represents the number of words associated with a vocabulary word based on the distributional hypothesis. The study involved three context sizes, including 500, 750, and 1000.

   Therefore, 27 configurations were executed, incorporating all the possible combinations of ($emb\_size \in 100, 200, 300$, $window\_size \in 1, 2, 3$, and $context\_size \in 500, 750, 1000$) for each intrinsic test dataset. Table 4.2 presents all the configurations for the incremental WE models based on word context matrices.

   It is important to mention that the vocabulary size in all configurations was set to capture 1,000,000 words. Additionally, the period value, p, utilized in our experiments was set to 320,000 instances, with a batch size of 32. This period value was selected

as it represents the point at which the evaluator was called after processing 320,000 tweets. These parameters were carefully selected to effectively analyze the performance of the different incremental word embedding models.

Table 4.1: Hyperparameter configuration for the ISG and ICBOW models.

Embedding size	Window size	Num. Neg. Samples
100	1	6
100	1	8
100	1	10
100	2	6
100	2	8
100	2	10
100	3	6
100	3	8
100	3	10
200	1	6
200	1	8
200	1	10
200	2	6
200	2	8
200	2	10
200	3	6
200	3	8
200	3	10
300	1	6
300	1	8
300	1	10
300	2	6
300	2	8
300	2	10
300	3	6
300	3	8
300	3	10

4.2.2 Results and discussion

Our analysis thoroughly evaluated various configurations (243 in total), considering the combination of three architectures, multiple hyperparameter values, and intrinsic evaluation tasks. As an illustration, we present two examples of the executed configurations:

- A configuration comprised of the ICBOW model, with hyperparameters set to $emb\_size = 300$, $window\_size = 3$, and $num\_ns = 10$, was employed for the similarity evaluation task using the MEN dataset.

Table 4.2: Hyperparameter configuration for IWCM model.

Embedding size	Window size	Context size
100	1	500
100	1	750
100	1	1000
100	2	500
100	2	750
100	2	1000
100	3	500
100	3	750
100	3	1000
200	1	500
200	1	750
200	1	1000
200	2	500
200	2	750
200	2	1000
200	3	500
200	3	750
200	3	1000
300	1	500
300	1	750
300	1	1000
300	2	500
300	2	750
300	2	1000
300	3	500
300	3	750
300	3	1000

- Another configuration involved the ISG model, with hyperparameters defined as $emb\_size = 100$, $window\_size = 1$, and $num\_ns = 6$, was utilized for the AP dataset's categorization evaluation task.

As mentioned in Section 3.3, the Periodic Evaluation is conducted as an intrinsic evaluation task after processing p instances during the training loop of any incremental WE model. Table 4.3 presents the results of applying the Periodic Evaluation to all hyperparameter configurations listed in Table 4.1, using the ICBOW model and the MEN dataset based on similarity tasks. Unfortunately, the complete table cannot be displayed in this document due to its size, but it is available in the repository[2] of our library.

Table 4.4 presents the results of the Periodic Evaluation, which are sorted in descending order based on the mean Spearman correlation obtained during the training process. Upon

[2]https://github.com/dccuchile/rivertext/tree/main/experiments

analyzing the table, it can be observed that the best results are obtained for the larger window and embedding sizes with the ICBOW model in the similarity task using the MEN dataset. However, this table just gives an insight into the best hyperparameter setting for a specific model, omitting a concise comparison of how the other models behave in front of the different hyperparameter settings and evaluation tasks.

The tables containing the results of our experiments for the remaining models can be found in the Section B of this document. However, due to their size, they may not be fully visible on a single page. Therefore, we recommend referring to the documentation page for a complete view of these tables.

Table 4.3: The table shows the results of the periodic evaluation of the ICBOW model and the MEN dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ICBOW	**300**	**3**	**6**	**0,5075**	**0,3546**	**...**	**0,5236**	**...**	**0,5194**
ICBOW	300	3	8	0,507	0,3381	...	0,5298	...	0,5275
ICBOW	300	3	10	0,505	0,3366	...	0,5291	...	0,5327
ICBOW	100	3	8	0,4891	0,3499	...	0,5233	...	0,5013
ICBOW	200	3	10	0,4884	0,3433	...	0,4971	...	0,5112
ICBOW	100	3	6	0,4879	0,3128	...	0,4985	...	0,5108
ICBOW	200	3	8	0,4828	0,3562	...	0,5041	...	0,5081
ICBOW	100	3	10	0,4828	0,3288	...	0,5058	...	0,5168
ICBOW	300	2	6	0,4827	0,3312	...	0,4988	...	0,5242
ICBOW	200	3	6	0,4778	0,3359	...	0,4892	...	0,5097
ICBOW	300	2	8	0,4758	0,3474	...	0,4967	...	0,4895
ICBOW	300	2	10	0,4677	0,3446	...	0,4762	...	0,4854
ICBOW	200	2	6	0,4555	0,3067	...	0,4865	...	0,4969
ICBOW	100	2	8	0,4541	0,3102	...	0,4979	...	0,4613
ICBOW	200	2	10	0,4516	0,3264	...	0,4597	...	0,4784
ICBOW	200	2	8	0,4476	0,2761	...	0,436	...	0,4626
ICBOW	100	2	6	0,4475	0,3197	...	0,4547	...	0,4961
ICBOW	100	2	10	0,4402	0,3261	...	0,4465	...	0,4557
ICBOW	300	1	8	0,4226	0,2779	...	0,438	...	0,4406
ICBOW	300	1	6	0,4195	0,2763	...	0,4465	...	0,4501
ICBOW	300	1	10	0,4164	0,2947	...	0,4276	...	0,4422
ICBOW	200	1	6	0,4007	0,2796	...	0,4076	...	0,407
ICBOW	200	1	8	0,3942	0,3085	...	0,4155	...	0,3932
ICBOW	200	1	10	0,3836	0,2657	...	0,3858	...	0,425
ICBOW	100	1	6	0,3822	0,2717	...	0,3594	...	0,4178
ICBOW	100	1	8	0,3784	0,2671	...	0,388	...	0,4095
ICBOW	100	1	10	0,3721	0,2719	...	0,3797	...	0,3916

To determine the optimal hyperparameter configuration for each architecture and across all tasks, we employed a ranking system based on the democratic voting procedure, Borda

Count [28]. The steps involved in this ranking system are as follows:

- First, the mean value of each hyperparameter configuration and test dataset is computed based on the results obtained from the time series analysis.
- Secondly, for each evaluated test dataset, the average mean value is calculated across all intrinsic tasks.
- Finally, we ordered the obtained average, with the lower position indicating the optimal configuration.

By employing this ranking system, we aim to analyze the best hyperparameter configurations for each model and test dataset, considering that the intrinsic tasks' results are unrelated.

In Table 4.4, we illustrate each model's top three ranked hyperparameter configurations. The complete ranking, including all configurations, can be found on the documentation page[3]. It is crucial to mention that while the example Table showcases the best three configurations for each model, the full ranking encompasses a broader range of results.

Table 4.4: The Overall Ranking of the benchmark results are based on the average of the Periodic Evaluation applied across the text stream. The result tasks are calculated by finding the mean of the evaluation, and the overall mean is determined by taking the average of these result tasks. This overall mean then determines the position in the ranking.

		Hyperparameters				Result tasks			
Position	Model	Emb. size	Win. size	Num. N.S	Context size	Mean MEN	Mean Mturk	Mean AP	Overall mean
1	**ICBOW**	**100**	**3**	**6**	**-**	**0.488**	**0.439**	**0.294**	**0.407**
2	ICBOW	300	3	8	-	0.507	0.428	0.284	0.406
3	ICBOW	300	3	6	-	0.508	0.416	0.289	0.404
4	**ISG**	**100**	**1**	**8**	**-**	**0.44**	**0.4**	**0.321**	**0.387**
5	ISG	100	1	6	-	0.443	0.393	0.312	0.383
6	ISG	100	2	10	-	0.421	0.399	0.309	0.376
7	**IWCM**	**100**	**3**	**-**	**1000**	**0.44**	**0.343**	**0.319**	**0.367**
8	IWCM	200	3	-	1000	0.438	0.351	0.307	0.366
9	IWCM	300	3	-	1000	0.439	0.35	0.307	0.365

As can be seen from the results, the neural network models ICBOW and ISG demonstrate superior performance, on average, compared to the non-neural network IWCM model. Notably, the ICBOW models attain better results with larger embedding and window sizes. In contrast, the ISG models perform optimally with smaller embedding and window sizes. In the case of the IWCM model, the effect of embedding and window sizes on performance is unclear. However, a trend towards improved performance with larger context sizes is observable.

Considering these findings in the context of the chosen evaluation metrics and the intrinsic tasks involved is important. The results suggest that the neural network architecture of the ICBOW and ISG models may significantly impact the performance, particularly concerning capturing semantic relationships between words. Additionally, the varying optimal configurations for the ICBOW and ISG models highlight the need for thorough experimentation and

[3]https://dccuchile.github.io/rivertext/benchmark/

Figure 4.1: Best setting models for MEN, Mturk, and AP datasets. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

analysis when selecting hyperparameters in these models. Further research may also consider exploring the underlying mechanisms and reasons for the observed performance differences between the models.

According to Table 4.4, we can state that the best hyperparameter setting for each model are:

- Best setting configuration for ICBOW model is $emb\_size = 100$, $window\_size = 3$, and $num\_ns = 6$.
- Best setting configuration for ISG model is $emb\_size = 100$, $window\_size = 1$, and $num\_ns = 8$.
- Best setting configuration for IWCM model is $emb\_size = 100$, $window\_size = 3$, and $context\_size = 1000$.

The results of the optimal hyperparameter configurations for each model are displayed

in Figure 4.1. This figure showcases the performance dynamics across different periods for the MEN, Mturk, and AP datasets. The figure highlights the crucial role of hyperparameter tuning in optimizing the performance of each model. The ICBOW model appears to perform more in the similarity task than the categorization task. Conversely, the ISG and IWCM models perform better in the categorization task and outperform the ICBOW model. However, it is noteworthy that the results of a model for a specific intrinsic evaluation task and dataset can vary significantly and are not always related. Thus, a model may perform well in one task but poorly in another.

The appendix contains the remaining graphs showing the relationship between hyperparameter settings in the three architectures and intrinsic NLP tasks.

Chapter 5

RiverText Library

The RiverText[1] library is an open-source implementation of various incremental word embedding techniques found in literature, adhering to the streaming learning paradigm. Developed in Python, the library is seamlessly integrated with several popular data science libraries and can be easily installed using pip[2]. The project is publicly hosted on Github[3] under the BSD 3-Clause license and is structured according to the design and code patterns recommended by the river[4] community.

The library has extensive documentation, including tutorials, an API definition, and a guide on contributing to the project, executing tests, and compiling documentation. The RiverText library is a valuable resource for researchers and practitioners looking to implement incremental word embeddings in their projects and contribute to advancing the field.

The RiverText library was developed with the following objectives in mind:

- Standardizing and encapsulating existing incremental word embedding algorithms and designing new ones to advance the field.
- Proposing a preliminary evaluation scheme for incremental word embedding models using intrinsic evaluation NLP tasks to measure their effectiveness and robustness.
- Providing an easily accessible toolkit for both the research community and industry to access these models and facilitate their practical applications.

It also provides support for the following:

- Loading and iterating over files that cannot be stored on disk.
- Saving metric values in JSON files across the training phase of the models.

[1] https://dccuchile.github.io/rivertext/
[2] https://pypi.org/project/rivertext/
[3] https://github.com/dccuchile/rivertext
[4] https://riverml.xyz

- Implementing a vocabulary class that can be used for any incremental method.

The following section provides a detailed description of the library and its implementation. We begin by discussing the motivation behind the development and publication of this library. Next, we present the design of the main components and processes involved in incremental word embedding techniques. Finally, we highlight some of the advanced processes of the library, such as loading larger files, training a model with one or many instances, running periodic evaluations, and saving the results.

5.1 Motivation

One of the key prerequisites for our benchmark analyses was implementing the incremental WE algorithm used in previous studies. Therefore, we attempted to reuse all publicly available resources and recreate any missing ones. However, upon completing the development of our framework, which uses the incremental learning types of instance and batch, it became apparent that unifying the different implementations was not feasible. Instead, each code and resource was developed to meet the specific requirements of individual studies, which greatly restricted their reuse and extension.

Consequently, we decided to re-implement the incremental WE algorithms according to our framework's guidelines and software decisions. The high level of standardization provided by RiverText enabled us to implement the algorithms and develop them into a well-designed and highly extensible code that follows best practices in Python development.

Furthermore, the absence of standardized tools for streaming embeddings, the limitations of our benchmark analyses, and the advanced level of development we were conducting prompted us to publish our code as an open library accessible to members of the research community and industry. Ensuring the code's broader usage and facilitating the standardization of incremental WE algorithms for future research.

5.2 Components

In this section, we will present the main components of our framework and discuss how the algorithms were standardized, as described in Chapter 3.

5.2.1 Word Embedding Model

This section will explore the various classes and sub-packages responsible for implementing incremental WE models in our library. As we have mentioned earlier, the primary objective of these methods is to update WE in real-time as new data arrives rather than retraining the entire model on the entire dataset every time new data is added. Our implementation

of incremental WE leverages the techniques proposed in the literature to update traditional WE algorithms efficiently.

From a technical perspective, our choice to use the river library for our implementations was influenced by its expanding community and the prevalence of its use among users. Although the river library predominantly emphasizes instance learning via the `learn_one` method, several essential classes provide a `learn_many` method that facilitates incremental batch learning. This approach holds great importance to us, as it allows us to implement incremental word2vec models using PyTorch as a backend for the neural network layer. It is worth noting that the PyTorch library is optimized for batch operations utilizing native tensor objects for vector operations.

We organized the implementation into the following Python classes:

`IWVBase` class

The `IWVBase` class is foundational for implementing incremental WE methods in the RiverText library. In addition, by extending the Transformer and VectorizeMixin classes from the River library, the `IWVBase` class provides a powerful framework for processing and analyzing text data.

The `Transformer` class is a key component of the river library and provides several methods for learning from incoming data, including the `learn_one` and `learn_many` methods. These methods allow the `IWVBase` class to update its word embeddings incrementally based on new input data. Additionally, the Transformer class includes several data standard methods, such as `predict_one` and `predict_many`, that enable the `IWVBase` class to make predictions and classify text data.

The `VectorizeMixin` class contains several attributes and methods that facilitate text data processing, such as tokenization and vectorization. These methods are crucial for generating word embeddings from text data and transforming them into a format that machine learning models can use.

To facilitate the implementation of new incremental WE methods in the RiverText library, it is recommended to extend the `IWVBase` class. By doing so, the new methods can inherit the common attributes, methods, and characteristics of the `IWVBase` class, which can help to reduce development time and increase code efficiency.

Extending the `IWVBase` class provides several benefits. First, it ensures that the new methods have access to the core functionalities of the `IWVBase` class, including the Transformer and VectorizeMixin classes, which are essential for text data processing and machine learning. Second, it promotes code reuse and modularity, as the common features of the `IWVBase` class can be easily shared among the different incremental WE methods. Furthermore, by extending the `IWVBase` class, new methods can leverage existing code and incorporate new features without disrupting the codebase.

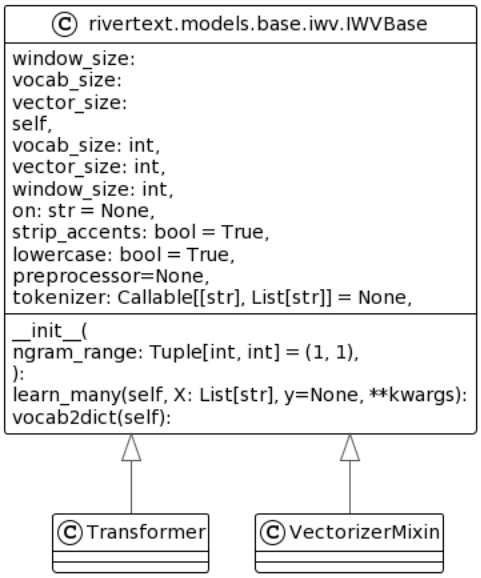

Figure 5.1: The class diagram for `IWVBase`, representing a base incremental WE algorithm, should include the attributes and methods specific to this class. As `IWVBase` extends the `Transformer` and `VectorizeMixin` classes, it can inherit their respective methods for processing textual data streams and for applying the incremental learning paradigm.

In diagram 5.1, we can see the attributes and methods of the `IWVBase` class, which extends the Transformer and VectorizeMixin classes.

`IWCM` class

The `IWordContextMatrix` is a class designed for capturing the meaning of words based on their contexts. It implements the Incremental Word Context model, which analyzes the co-occurrence of words within a given corpus to comprehensively represent the relationships between words.

To achieve this, the `IWordContextMatrix` class extends the `IWVBase` class and provides methods for instance and batch incremental learning, including the `learn_one` and `learn_many` methods. The model also employs a streaming version of weighted correlations between the target and its contexts to ensure accuracy in capturing the meaning of words using PPMI values.

There are two options for obtaining resulting embeddings from the model: the first is to use the word context matrix method to obtain sparse vectors, while the second is to reduce the embedding using the Incremental PCA algorithm, the user can decide which option wants to use configuration the `reduce_dim` parameter of the class `IWordContextMatrix`. Both options allow for flexibility and customization depending on the user's needs.

Throughout the iteration process over the source of data streams, the model must store the counter for computing the streaming PPMI values. These values are stored in the `Vocab` object, and the model only updates vectors for words that have had their counts incremented.

Additionally, the vocabulary is updated to include new words appearing in the text streams, and this update is performed using the Misra Gries algorithm, as mentioned before. Finally, the word context matrix is stored in the sparse matrix object of the `scipy` library, providing efficient and scalable processing of large volumes of data.

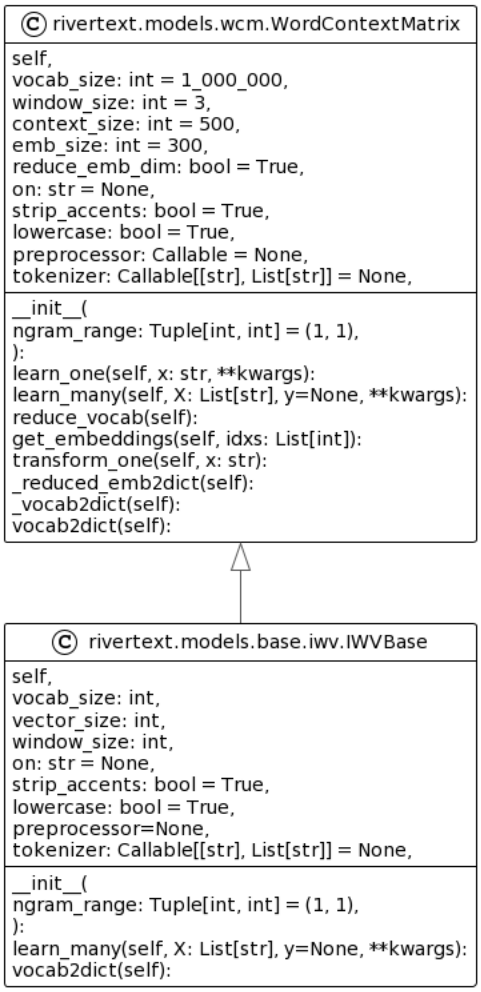

Figure 5.2: Attributes and methods for the `IWordContextMatrix` class that represents the IWCM method.

Diagram 5.2 presents the `IWordContexMatrix` class and its associated attributes and methods. This class is designed for generating word embeddings using the word-context matrix approach, which involves computing the co-occurrence statistics of words and constructing a matrix that encodes the relationships between them.

One important feature of the `IWordContexMatrix` class is its dynamic vocabulary reduction capability. This is accomplished through the `reduce_vocab` method, which frees up space for new words when the vocabulary becomes full. This is particularly useful in streaming environments with large vocabulary sizes and constantly changing. It allows the model to adapt and continue learning without being constrained by a fixed vocabulary size. Additionally, the `IWordContexMatrix` class includes the `reduce_emb2dict` method, which reduces the dimensionality of the generated word embeddings using the Incremental PCA

algorithm.

`IWord2Vec` class

The `IWord2Vec` class is a highly efficient implementation of the Incremental SkipGram and Continuous Bag of Words (CBOW) models for generating word embeddings. It builds on the `IWVBase` class, which serves as a base structure for the Word2Vec algorithm and any incremental embedding method. Users can choose the desired model by setting the value of the sg parameter to 1 for SkipGram or 0 for CBOW.

One of the key advantages of `IWord2Vec` is its use of the PyTorch deep learning framework as its internal neural network backend. This allows for using PyTorch's numerous benefits, such as leveraging GPUs for faster training and selecting from a wide range of optimization algorithms to enhance the model's performance. The `learn_one` and `learn_many` methods depend on a preprocessor object provided by the `iword2vec_utils` model to transform the input text into a format suitable for neural network training. After preprocessing, the inputs are passed to the forward method of the respective embedding architecture. The selected model determines the specific type of preprocessor used.

The `IWord2Vec` class has several important parameters, including the size of the vocabulary, the size of the unigram table, the size of the embedding, the number of negative samples to use during training, and the model to use. The size of the vocabulary and the size of the unigram table can impact the quality of the word embeddings generated by the model. Larger values of these parameters can lead to better performance, but they also increase the computational cost of training. The size of the embedding controls the dimensionality of the word vectors that the model generates. The number of negative samples is used during training to improve the quality of the word embeddings. It can significantly impact the model's performance, but too many negative samples can lead to overfitting.

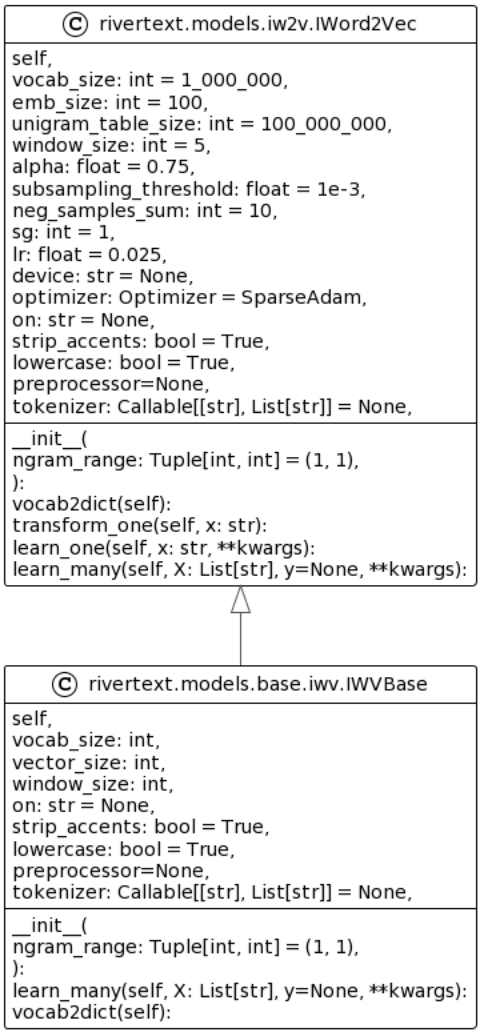

Figure 5.3: Attributes and methods for the `IWord2Vec` class that represents the ISG and ICBOW methods.

Diagram 5.3 displays the attributes and methods associated with the `IWord2Vec` class, a neural network-based model for generating word embeddings. One notable difference between Diagram 5.3 and Diagram 5.2, which also deals with word embeddings, is that the former includes the `iwordvec_utils` sub-package, which is responsible for processing the vocabulary and inputs for the neural network.

`iword2vec_utils` sub package

The `iword2vec_utils` subpackage constitutes an integral component of the `IWord2Vec` implementation. Its purpose is to provide a collection of classes that facilitate the incremental update of the unigram table and offer preprocessing algorithms that transform textual data into positive and negative samples suitable for training the model.

The unigram table is an important data structure utilized during negative sampling in the word embedding training. It represents the frequency distribution of words within the input corpus and is instrumental in sampling negative examples during training. The `iword2vec_utils` package includes a class, called `UnigramTable`, that efficiently updates the unigram table during incremental training, with methods such as `samples` for randomly selecting negative examples, `build` for constructing the unigram table structure, and `update` for updating the table using the algorithm proposed by Kaji and Kobayashi. The first figure of Diagram 5.4 illustrates the `UnigramTable` and its methods.

Besides the unigram table implementation, the `iword2vec_utils` package provides several preprocessing algorithms that convert textual data into positive and negative samples suitable for feeding into the model during training. The specific preprocessing algorithm depends on the selected embedding model, such as SkipGram or CBOW. For instance, one preprocessing algorithm converts a sequence of words into a tuple comprising the target and context words. In contrast, another converts a sequence of words into a tuple containing the target word and a negative sample. Diagram 5.4 depicts the `Preprocessor` as the base class for implementing the inputs preprocessing algorithm for textual streams. Given that the ISG and ICBOW models differ in their input formats, the `PrepSG` and `PrepCBOW` classes extend the `Preprocessor` and differ in the input format specified within the `__call__` method when the preprocessor objects are instantiated.

Another significant set of classes provided by the `iword2vec_utils` subpackage pertains to the PyTorch backend, which facilitates the implementation of the neural network component of the incremental neural models. Analogous to the preprocessing classes, the `WordVec` class implements common functionality for the two methods, ISG and ICBOW. For the specific vector calculations of the methods, the `WordVec` class is extended by the `SG` and `CBOW` classes, differing in their `forward` method, which implements the loss function for each method. The final figure in Diagram 5.4 illustrates the `WordVec` class and its child classes.

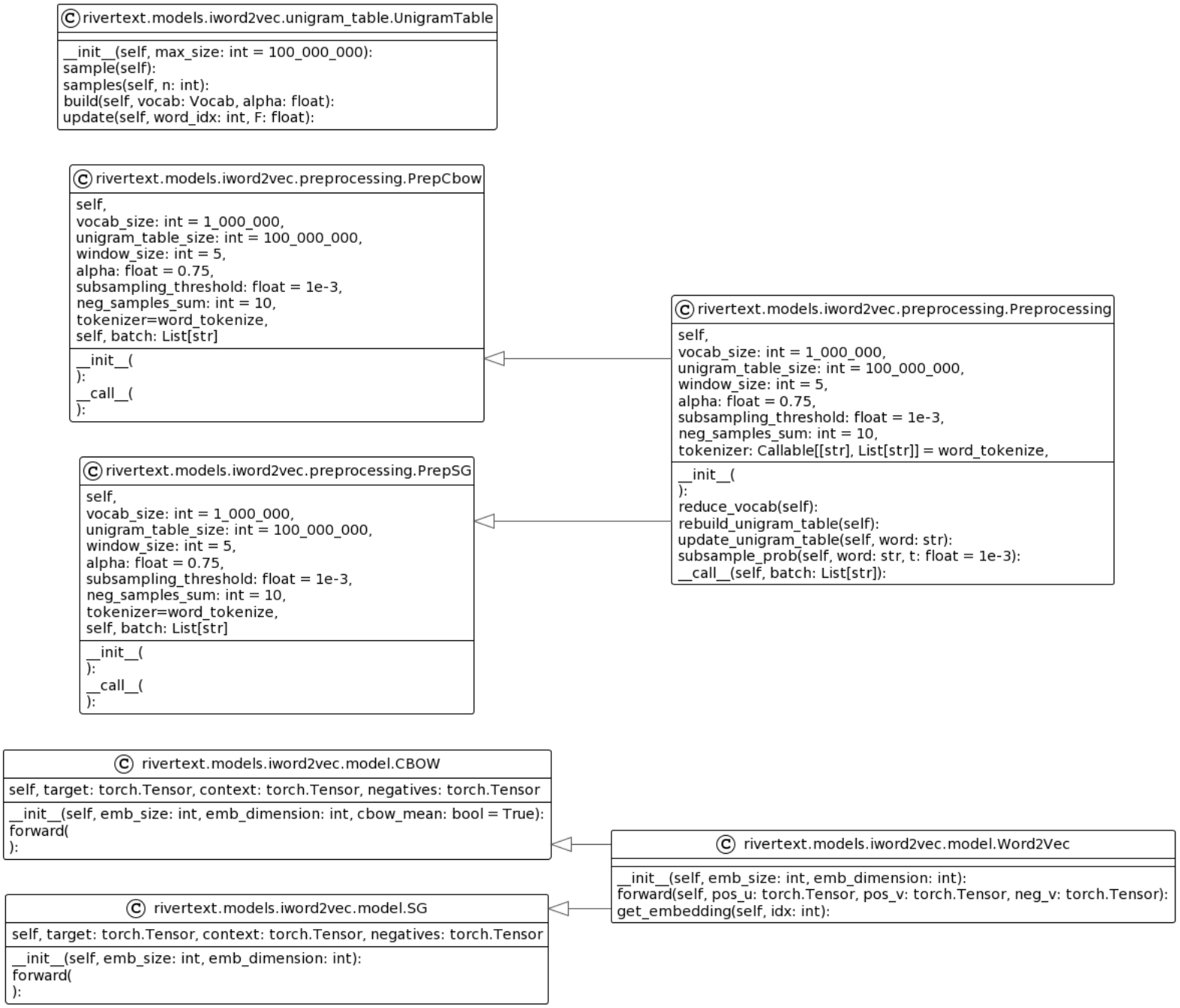

Figure 5.4: The `iword2vec_utils` module includes three crucial classes for its functionality, which are visualized in three separate diagrams. The first diagram displays the `UnigramTable` class, which creates and maintains a table of unigram frequencies used in the ISG and ICBOW models. Its primary methods include generating a table, sampling words from the table, and updating it after a new word is processed. The second diagram shows the `Preprocessor` class, responsible for converting the input stream of text into a neural network representation. It uses techniques such as subsampling and negative sampling to prepare the input for the neural network. Finally, the third diagram depicts the PyTorch implementation for the neural network backend, which includes input and output embedding layers and hidden layers that perform the neural network computations. The diagrams visually represent the classes and their interactions within the `iword2vec_utils` module.

5.2.2 Periodic Evaluation

The `PeriodicEvaluator` class assesses the performance of an incremental WE model using an intrinsic NLP task and a related-test dataset after a set number of instances have been processed and trained. The class takes as input the dataset to train on, the model to evaluate, the number of instances to process before evaluating the model, a golden dataset containing relations, an evaluation function, and a path to an output file to store the evaluation results.

In diagram 5.5, we depict the attributes and methods of the `PeriodicEvaluation` class. The `run` method executes periodic assessments of the entire model every p instances, where p is the number of instances to process before evaluating the model. The method first processes a batch of data using the model's `learn_many` method. Then, suppose the number of processed instances is divisible by p. In that case, the model's embeddings are extracted using the `vocab2dict` method, and the `evaluator` function is called to evaluate the model's performance on the test dataset. The result is appended to `store_results` list, and if an output file path is provided, it is saved in a JSON file.

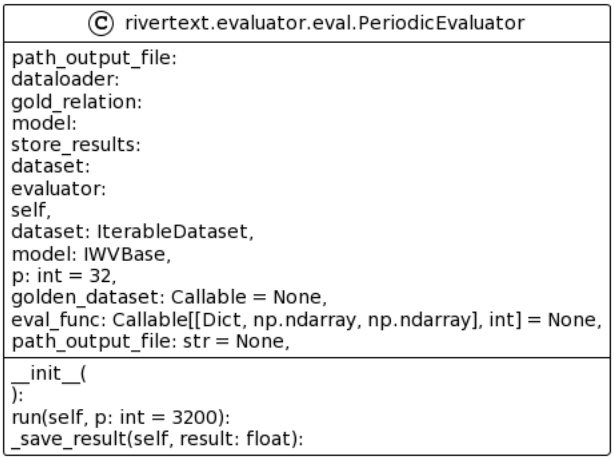

Figure 5.5: Diagram of class that shows the attributes and methods for the `PeriodicEvaluation` class that represent the Periodic Evaluation.

5.2.3 Utils

The `utils` package implements utility classes and functions for code execution. The main classes are:

Vocabulary

The container class is a fundamental component in storing the words used as vocabulary from a text stream. This vocabulary implementation is used across all incremental word embedding algorithms. In other words, the container class acts as a central repository for all the words processed by the incremental word embedding algorithms.

As the text stream grows and new words are encountered, the container class dynamically updates the vocabulary to ensure it accurately represents the full range of words, such as the Misra Greis algorithm.

The `Vocab` class is a critical container that stores the vocabulary utilized in incremental word embedding algorithms. Its inner structure consists of two VectDict attributes. The first attribute stores the words that make up the model's vocabulary, while the second attribute is responsible for keeping track of how many times a word in the vocabulary is seen in the text stream.

Additionally, in Diagram 5.6, we present how the `Vocab` class features an `add_token` method that adds new tokens to the vocabulary by checking whether they are already contained in the `VectDict` object. If the token is absent in the vocabulary, the `add_token` method adds the word and increments its count by one. However, the method updates the word count if the token is already in the vocabulary.

Furthermore, when the vocabulary structure becomes full, the class has a remove method that eliminates tokens. Finally, the Misra Greis algorithm is executed to make space for new words in the vocabulary. With this algorithm, the class decrements the count of all words by one and removes the ones with a count of one.

TweetStream Dataloader

The `TweetStream` class is an important feature in our implementation that allows for the efficient loading of large text files. This class extends the `IterableDataset` class of the PyTorch API, which integrates seamlessly with PyTorch's data-loading utilities.

Using the `TweetStream` class enables us to overcome the memory constraints associated with loading entire datasets into memory. Instead, the class reads and processes the text stream iteratively, loading only one tweet at a time, making it possible to work with much larger datasets that exceed our system's memory capacity. However, it's worth noting that datasets must have a specific format, with one tweet per line separated by a break line.

Moreover, in Diagram 5.6 we shows how the `TweetStream` class provides a wide range of helpful methods to preprocess the text stream before feeding it into our incremental learning algorithms. For instance, we can use the class to filter out stop words or perform tokenization, stemming, or lemmatization to enhance the quality of the training data.

The class includes two methods: the `preprocess` method, which formats the data according to the user's specifications, and the `__iter__` method, which loads the next line or chunk of text. These methods can be customized to meet specific requirements and improve the performance of text stream processing.

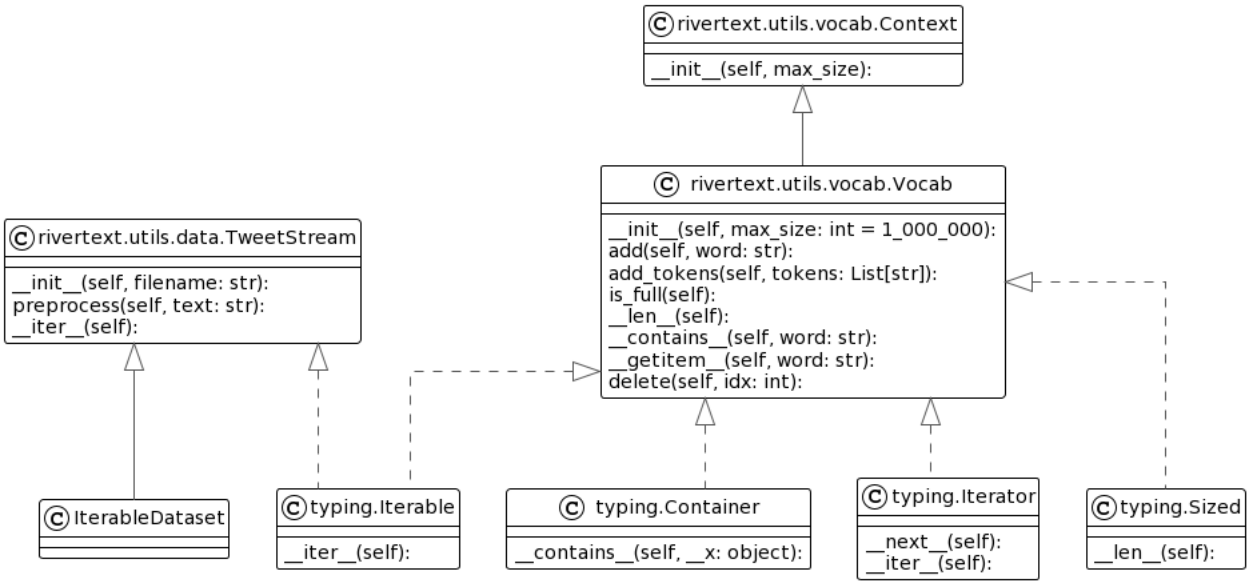

Figure 5.6: The `utils` package contains two important classes: `TweetStream` and `Vocab`. The `TweetStream` class is responsible for loading and iterating through files that may not be stored on disk, providing a convenient interface for processing large volumes of text data stored in memory or streamed from an external source. The `Vocab` class stores the words associated with the vocabulary for the incremental word embedding methods.

5.2.4 Training Process

In this section, we will discuss two of the most important processes implemented by the library: the training process and the periodic evaluation process discussed in previous sections.

Training model workflow

In a more technical context, the training process provided by RiverText involves a series of steps that can be subdivided into inputs, actions, and outputs.

The input steps involve defining the main hyperparameters of the model, such as `vocab_size`, `context_size`, and `num_ns`, among others. Additionally, the user must instantiate the model object and enable the source of text streams, for which it is necessary to define the iterable object `TweetStream`. This object allows for the opening and iteration of files that may not be stored on disk.

After defining the model and enabling the source of text streams, the next step is to create a data loader object from the PyTorch API. This object is used to iterate over the text the data streams provide. Once the data loader is ready, the chosen model can train and update the vectors generated by the text streams. It is worth noting that these actions are continuously executed, ensuring the embeddings are continually updated during training.

The resulting embedding vectors generated by the model and text streams serve as the

output of the training process. Importantly, these actions are continuously executed, allowing the embeddings to be continuously updated throughout the training process.

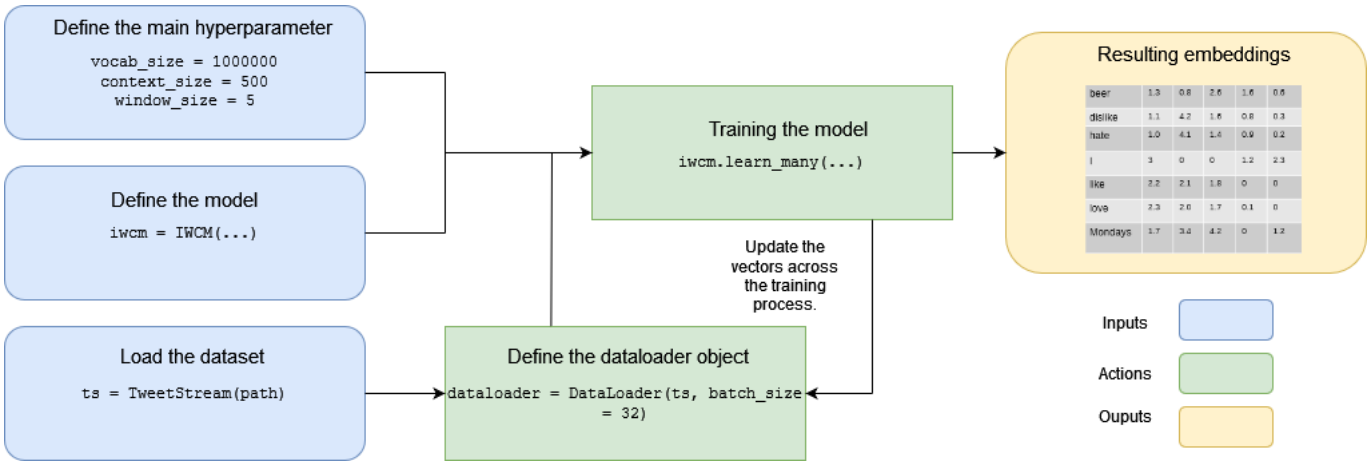

Figure 5.7: Training scheme for the IWCM model.

Diagram 5.7 illustrates the training workflow for the IWCM model. The celestial squares in the diagram represent the input steps, while the green squares indicate the subsequent actions performed once the inputs are received. The yellow squares represent the embedding outputs. Notably, the leftward arrow from the training model square to the data loader indicates that the learning process occurs continuously while the vectors are being updated.

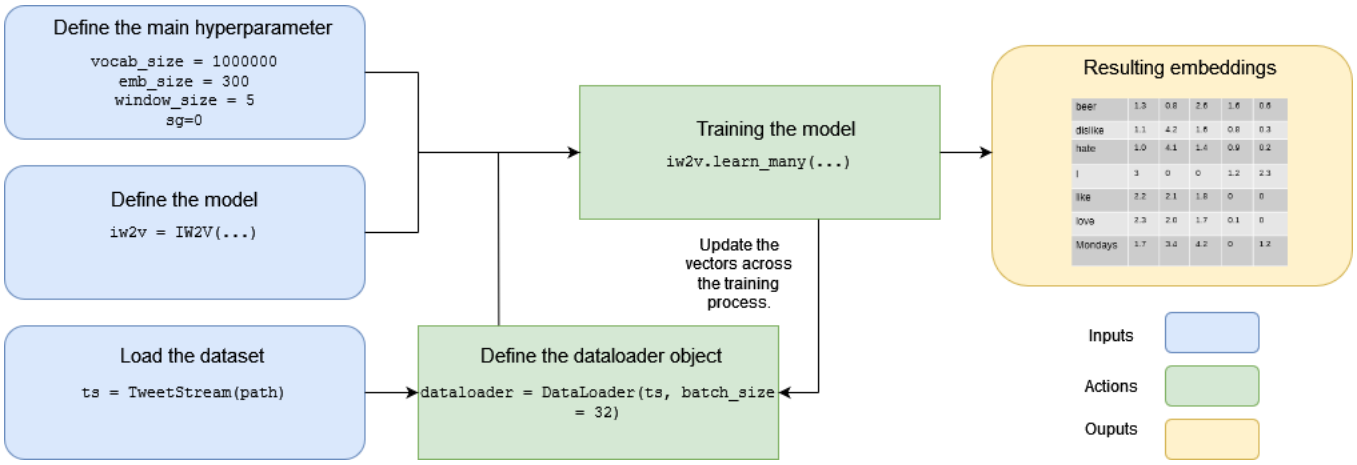

Figure 5.8: Training scheme for the incremental Word2Vec models.

Diagram 5.8 portrays the training process for the word2vec models. Within RiverText, the `IWord2Vec` object encapsulates the ISG and ICBOW models and serves as an interface for training dynamic word vectors from text streams. It is noteworthy that the choice of the word2vec model for training the text streams is contingent on the value of the sg parameter, with a value of 1 implying the utilization of the ISG model and the ICBOW model being implemented otherwise.

At the core of the training process is the `IWord2Vec` object, which generates and updates

the word vectors based on the input text streams. The main hyperparameters, such as `vocab_size`, `context_size`, and `num_ns` must be defined to begin the training process, as they underpin the model's configuration. The model object is then instantiated, and the iterable object `TweetStream` is employed to facilitate the input of text streams. The remaining steps closely resemble those employed in the IWCM model, as we aim to develop a unified interface for each incremental method.

Periodic Evaluation Execution

The Periodic Evaluation scheme is a useful tool for assessing the performance of incremental word embedding models in streaming scenarios using intrinsic NLP tasks. It is implemented in the evaluator package of the RiverText library.

The Periodic Evaluation scheme workflow involves several steps, which are illustrated in Diagram 5.9:

1. The user selects the hyperparameters for the incremental word embedding model and instantiates an object of that model.
2. The text streams are loaded into a `TweetStream` object for iteration.
3. The user selects a golden relation dataset related to an intrinsic NLP task.
4. To evaluate the model, the user instantiates a function related to the task chosen in the third step.
5. The inputs from the previous steps are passed into a `PeriodicEvaluation` object.

The actions performed by the `PeriodicEvaluation` object consist of applying the evaluation function from step four to the inputs of every `p` observation using the `run` method, enabling continuous evaluation of the model's accuracy and identifying areas for improvement.

The outputs from the Periodic Evaluation scheme consist of a `JSON` file that stores information about the assessment of the model. This includes the model name, hyperparameters, and the intrinsic NLP task metric results. This information can be used to monitor the model's performance over time and make improvements as necessary.

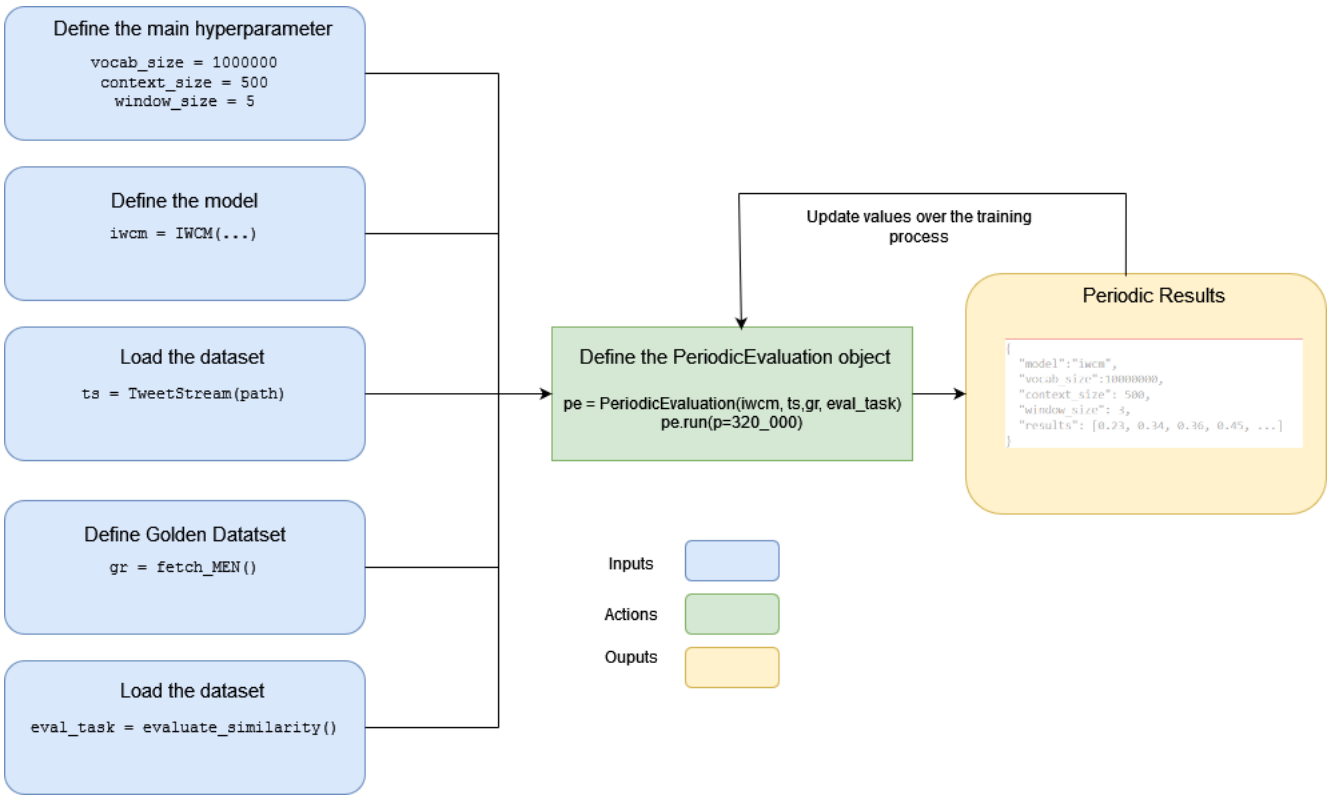

Figure 5.9: Workflow scheme for running the Periodic Evaluation using an incremental WE model.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

The landscape of human communication has rapidly changed with the advent of social media [23]. The increasing availability of real-time data streams poses significant challenges to the traditional word embedding models that rely on large corpora of static data. The ever-evolving nature of language in these dynamic environments makes it necessary to retrain the models continuously. This requirement is computationally expensive, making the development of incremental WE models an essential research area in NLP.

In this context, the RiverText framework significantly contributes to the field. RiverText provides a systematic approach for training and evaluating IWE models from text data streams. It enables the implementation of standardized models and provides a unified methodology for comparing and evaluating them. Furthermore, it offers a robust evaluation method based on intrinsic NLP tasks adapted to a streaming environment.

To assess the effectiveness of the framework, the authors conducted a benchmark study that evaluated three incremental WE architectures: ICBOW, ISG [46, 61], and IWCM [20]. The benchmark study evaluated the models using periodic intrinsic evaluations of word similarity and categorization tasks. The study revealed that hyperparameter tuning is essential for the optimal performance of the models. For instance, larger embedding sizes improve ICBOW's performance, while ISG benefits from smaller embedding sizes.

One disadvantage of the periodic evaluation approach is that it cannot detect concept drift [95], a common problem in streaming data. Concept drift refers to the changes in the data distribution that may occur over time, which can significantly affect the performance of the models. Although the proposed approach allows for visualizing the model's performance throughout the training process, it fails to capture the effect of concept drift on the models. Unfortunately, no standard methodology or benchmark dataset can simulate concept drift in IWE models. Therefore, we strongly recommend carefully applying these algorithms' results in streaming environments.

We have made RiverText available as an open-source library, which includes a compre-

hensive set of tutorials and API documentation. This makes it easy for researchers and practitioners to use the framework and replicate the benchmark study. With RiverText, the authors aim to promote reproducibility and further research in incremental WE models. The framework's standardization and evaluation methodology make it an important contribution to the field and facilitate further research in this exciting area. The authors welcome contributions from the community to enhance the library and improve the understanding of IWE models.

6.2 Future Work

Our study focuses on streaming learning under the word embeddings [20, 46, 61, 76, 75] algorithm and its evaluation. However, the field of study has advanced since we based our framework, opening up opportunities for machine learning from data streams in various areas, particularly in NLP.

Moving forward, we plan to expand our system's capabilities for text representation and evaluation in the context of the time-evolving text. Our primary goal is to incorporate more incremental text representation methods, such as incremental Glove [76], to increase the flexibility and adaptability of word and phrase representation as they evolve.

Additionally, we aim to develop an evaluation methodology that considers concept drift, which pertains to the semantic changes of words over time [7]. As we point out, our current periodic evaluation approach assumes that golden relations, such as word pair similarities or categories, remain static during the stream, which is not an adequate assessment of the word vectors' ability to adapt to change. Therefore, we plan to extend the concept drift idea developed in [20] to simulate the semantic change in synthetic tweets for all intrinsic tasks of WE evaluation

We also intend to improve the functionality of our software by incorporating other sketching techniques outlined in previous studies, such as [37], which enable efficient updating of the vocabulary with minimal memory usage. Additionally, we aim to integrate incremental detection of collocations or phrases, as described in recent research such as [43]. This would enhance the representation of multi-word expressions such as "New Zealand" or "New York" in our vocabulary.

Another critical area we aim to focus on in our future work is implementing a data loader that connects to the Twitter API and streams topic-specific tweets for training. This would enable users to monitor social media and extract relevant information efficiently, expanding the potential of our system and enabling it to analyze the most recent and pertinent data.

Finally, we hope our system will inspire further investigation in NLP and incremental learning, particularly in the representation of words and documents in the context of the time-evolving text. With the exponential growth of social media and the web, our system has the potential to be a valuable tool for extracting insights and knowledge from vast amounts of text data.

Bibliography

[1] Hervé Abdi and Lynne J Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[2] RR Ade and PR Deshmukh. Methods for incremental learning: a survey. *International Journal of Data Mining & Knowledge Management Process*, 3(4):119, 2013.

[3] Charu C Aggarwal. *Data streams: models and algorithms*, volume 31. Springer, 2007.

[4] Adnan Akhundov, Dietrich Trautmann, and Georg Groh. Sequence labeling: A practical approach. *arXiv preprint arXiv:1808.03926*, 2018.

[5] Anton Alekseev and Sergey Nikolenko. Word embeddings for user profiling in online social networks. *Computación y Sistemas*, 21(2):203–226, 2017.

[6] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[7] Abdulrahman Almuhareb and Massimo Poesio. Concept learning and categorization from the web. In *proceedings of the annual meeting of the Cognitive Science society*, volume 27, 2005.

[8] Shun-ichi Amari. Backpropagation and stochastic gradient descent method. *Neurocomputing*, 5(4-5):185–196, 1993.

[9] Matej Artac, Matjaz Jogan, and Ales Leonardis. Incremental pca for on-line visual learning and recognition. In *2002 International Conference on Pattern Recognition*, volume 3, pages 781–784. IEEE, 2002.

[10] Wesam Barbakh and Colin Fyfe. Online clustering algorithms. *International journal of neural systems*, 18(03):185–194, 2008.

[11] Horace B Barlow. Unsupervised learning. *Neural computation*, 1(3):295–311, 1989.

[12] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *Ieee Potentials*, 13(4):27–31, 1994.

[13] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

[14] Albert Bifet and Eibe Frank. Sentiment knowledge discovery in twitter streaming data. In *International conference on discovery science*, pages 1–15. Springer, 2010.

[15] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Philipp Kranen, Hardy Kremer, Timm Jansen, and Thomas Seidl. Moa: Massive online analysis, a framework for stream classification and clustering. In *Proceedings of the first workshop on applications of pattern analysis*, pages 44–50. PMLR, 2010.

[16] Albert Bifet, Geoffrey Holmes, and Bernhard Pfahringer. Moa-tweetreader: real-time analysis in twitter streaming data. In *International conference on discovery science*, pages 46–60. Springer, 2011.

[17] Albert Bifet, Ricard Gavalda, Geoffrey Holmes, and Bernhard Pfahringer. *Machine learning for data streams: with practical examples in MOA*. MIT press, 2018.

[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146, 2017.

[19] Mohamed Reda Bouadjenek, Hakim Hacid, and Mokrane Bouzeghoub. Social networks and information retrieval, how are they converging? a survey, a taxonomy and an analysis of social information retrieval approaches and platforms. *Information Systems*, 56:1–18, 2016.

[20] Felipe Bravo-Marquez, Arun Khanchandani, and Bernhard Pfahringer. Incremental word vectors for time-evolving sentiment lexicon induction. *Cognitive Computation*, 14 (1):425–441, 2022.

[21] Elia Bruni, Nam-Khanh Tran, and Marco Baroni. Multimodal distributional semantics. *Journal of artificial intelligence research*, 49:1–47, 2014.

[22] Scott H Clearwater, Tze-Pin Cheng, Haym Hirsh, and Bruce G Buchanan. Incremental batch learning. In *Proceedings of the sixth international workshop on Machine learning*, pages 366–370. Elsevier, 1989.

[23] Evandro Cunha, Gabriel Magno, Giovanni Comarela, Virgilio Almeida, Marcos André Gonçalves, and Fabricio Benevenuto. Analyzing the dynamic evolution of hashtags on twitter: a language-based approach. In *Proceedings of the workshop on language in social media (LSM 2011)*, pages 58–65, 2011.

[24] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. Supervised learning. *Machine learning techniques for multimedia: case studies on organization and retrieval*, pages 21–49, 2008.

[25] Jurafsky Daniel, Martin James H, et al. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition.* prentice hall, 2007.

[26] Rahul Dey and Fathi M Salem. Gate-variants of gated recurrent unit (gru) neural networks. In *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*, pages 1597–1600. IEEE, 2017.

[27] Sean R Eddy. Hidden markov models. *Current opinion in structural biology*, 6(3): 361–365, 1996.

[28] Peter Emerson. The original borda count and partial voting. *Social Choice and Welfare*, 40(2):353–358, 2013.

[29] Atefeh Farzindar and Diana Inkpen. Natural language processing for social media. *Synthesis Lectures on Human Language Technologies*, 8(2):1–166, 2015.

[30] Bin Gao, Jiang Bian, and Tie-Yan Liu. Wordrep: A benchmark for research on learning word representations. *arXiv preprint arXiv:1407.1640*, 2014.

[31] Xin Geng and Kate Smith-Miles. Incremental learning., 2009.

[32] Sahar Ghannay, Benoit Favre, Yannick Esteve, and Nathalie Camelin. Word embedding evaluation and combination. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 300–305, 2016.

[33] Anna Gladkova and Aleksandr Drozd. Intrinsic evaluations of word embeddings: What can we do better? In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 36–42, 2016.

[34] Yoav Goldberg. Neural network methods for natural language processing. *Synthesis lectures on human language technologies*, 10(1):1–309, 2017.

[35] Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

[36] Heitor Murilo Gomes, Jesse Read, Albert Bifet, Jean Paul Barddal, and João Gama. Machine learning for streaming data: state of the art, challenges, and opportunities. *ACM SIGKDD Explorations Newsletter*, 21(2):6–22, 2019.

[37] Amit Goyal, Jagadeesh Jagarlamudi, Hal Daumé III, and Suresh Venkatasubramanian. Sketching techniques for large scale nlp. In *Proceedings of the NAACL HLT 2010 Sixth Web as Corpus Workshop*, pages 17–25, 2010.

[38] Alex Graves and Alex Graves. Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45, 2012.

[39] Max Halford, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, and Adil Zouitine. creme, a python library for online machine learning, 2019.

[40] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[41] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[42] Frederick Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, 1985.

[43] Sam Henry, Clint Cuffy, and Bridget T McInnes. Vector representations of multi-word terms for semantic relatedness. *Journal of biomedical informatics*, 77:111–119, 2018.

[44] Stanisław Jastrzebski, Damian Leśniak, and Wojciech Marian Czarnecki. How to evaluate word embeddings? on importance of data efficiency and simple supervised tasks. *arXiv preprint arXiv:1702.02170*, 2017.

[45] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[46] Nobuhiro Kaji and Hayato Kobayashi. Incremental skip-gram model with negative sampling. *arXiv preprint arXiv:1704.03956*, 2017.

[47] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, pages 462–466, 1952.

[48] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[49] Svetlana Kiritchenko, Xiaodan Zhu, and Saif M Mohammad. Sentiment analysis of short informal texts. *Journal of Artificial Intelligence Research*, 50:723–762, 2014.

[50] Joost N Kok, Egbert J Boers, Walter A Kosters, Peter Van der Putten, and Mannes Poel. Artificial intelligence: definition, trends, techniques, and cases. *Artificial intelligence*, 1:270–299, 2009.

[51] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. Text classification algorithms: A survey. *Information*, 10(4):150, 2019.

[52] Anders Krogh. What are artificial neural networks? *Nature biotechnology*, 26(2): 195–197, 2008.

[53] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553): 436–444, 2015.

[54] John A Lee and Michel Verleysen. Unsupervised dimensionality reduction: Overview and recent advances. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2010.

[55] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. *Advances in neural information processing systems*, 27, 2014.

[56] Li Li, Miloš Doroslovački, and Murray H Loew. Approximating the gradient of cross-entropy loss function. *IEEE Access*, 8:111626–111635, 2020.

[57] Rushi Longadge and Snehalata Dongre. Class imbalance problem in data mining review. *arXiv preprint arXiv:1305.1707*, 2013.

[58] Agnes Lydia and Sagayaraj Francis. Adagrad—an optimizer for stochastic gradient descent. *Int. J. Inf. Comput. Sci*, 6(5):566–568, 2019.

[59] Christopher D Manning. *Introduction to information retrieval*. Syngress Publishing,, 2008.

[60] James H Martin. *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall, 2009.

[61] Chandler May, Kevin Duh, Benjamin Van Durme, and Ashwin Lall. Streaming word embeddings with the space-saving algorithm. *arXiv preprint arXiv:1704.07463*, 2017.

[62] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*, pages 398–412. Springer, 2005.

[63] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[64] Bonan Min, Hayley Ross, Elior Sulem, Amir Pouran Ben Veyseh, Thien Huu Nguyen, Oscar Sainz, Eneko Agirre, Ilana Heinz, and Dan Roth. Recent advances in natural language processing via large pre-trained language models: A survey. *arXiv preprint arXiv:2111.01243*, 2021.

[65] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[66] Jacob Montiel, Jesse Read, Albert Bifet, and Talel Abdessalem. Scikit-multiflow: A multi-output streaming framework. *The Journal of Machine Learning Research*, 19(1): 2915–2914, 2018.

[67] Jacob Montiel, Max Halford, Saulo Martiello Mastelini, Geoffrey Bolmier, Raphael Sourty, Robin Vaysse, Adil Zouitine, Heitor Murilo Gomes, Jesse Read, Talel Abdessalem, et al. River: machine learning for streaming data in python. 2021.

[68] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *International workshop on artificial intelligence and statistics*, pages 246–252. PMLR, 2005.

[69] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends® in Theoretical Computer Science*, 1(2):117–236, 2005.

[70] Makbule Gulcin Ozsoy. From word embeddings to item recommendation. *arXiv preprint arXiv:1601.01356*, 2016.

[71] Odysseas Papapetrou, Minos Garofalakis, and Antonios Deligiannakis. Sketching distributed sliding-window data streams. *The VLDB Journal*, 24:345–368, 2015.

[72] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[73] WM Patefield. On the maximized likelihood function. *Sankhyā: The Indian Journal of Statistics, Series B*, pages 92–96, 1977.

[74] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[75] Hao Peng, Jianxin Li, Yangqiu Song, and Yaopeng Liu. Incrementally learning the hierarchical softmax function for neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[76] Hao Peng, Mengjiao Bao, Jianxin Li, Md Zakirul Alam Bhuiyan, Yaopeng Liu, Yu He, and Erica Yang. Incremental term representation learning for social network analysis. *Future Generation Computer Systems*, 86:1503–1512, 2018.

[77] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[78] Saša Petrović, Miles Osborne, and Victor Lavrenko. The edinburgh twitter corpus. In *Proceedings of the NAACL HLT 2010 workshop on computational linguistics in a world of social media*, pages 25–26, 2010.

[79] Heny Pratiwi, Agus Perdana Windarto, S Susliansyah, Ririn Restu Aria, Susi Susilowati, Luci Kanti Rahayu, Yuni Fitriani, Agustiena Merdekawati, and Indra Riyana Rahadjeng. Sigmoid activation function in selecting the best model of artificial neural networks. In *Journal of Physics: Conference Series*, volume 1471, page 012010. IOP Publishing, 2020.

[80] Kira Radinsky, Eugene Agichtein, Evgeniy Gabrilovich, and Shaul Markovitch. A word at a time: computing word relatedness using temporal semantic analysis. In *Proceedings of the 20th international conference on World wide web*, pages 337–346, 2011.

[81] Jesse Read, Albert Bifet, Bernhard Pfahringer, and Geoff Holmes. Batch-incremental vs. instance-incremental learning in dynamic and evolving data.

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

[83] D Ross, J Lim, RS Lin, and MH Yang. Incremental learning for robust visual tracking. internat. j. *Computer Vision*, 25(8):1034–1040, 2008.

[84] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[85] Johannes Schmidt-Hieber. Nonparametric regression using deep neural networks with relu activation function. 2020.

[86] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 298–307, 2015.

[87] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[88] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *Towards Data Sci*, 6(12):310–316, 2017.

[89] Ian Stewart, Dustin Arendt, Eric Bell, and Svitlana Volkova. Measuring, predicting and visualizing short-term change in word representation and usage in vkontakte social network. In *Eleventh international AAAI conference on web and social media*, 2017.

[90] Charles Sutton, Andrew McCallum, et al. An introduction to conditional random fields. *Foundations and Trends® in Machine Learning*, 4(4):267–373, 2012.

[91] Peter D Turney and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of artificial intelligence research*, 37:141–188, 2010.

[92] Pauli Virtanen, Ralf Gommers, Evgeni Burovski, Travis E Oliphant, David Cournapeau, Warren Weckesser, Pearu Peterson, Stefan van der Walt, Josh Wilson, Nikolay Mayorov, et al. Scipy/scipy: Scipy 1.1. 0. *Zenodo*, 2018.

[93] Jialei Wang, Peilin Zhao, Steven CH Hoi, and Rong Jin. Online feature selection and its applications. *IEEE Transactions on knowledge and data engineering*, 26(3):698–710, 2013.

[94] Qi Wang, Yue Ma, Kun Zhao, and Yingjie Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, pages 1–26, 2020.

[95] Geoffrey I Webb, Roy Hyde, Hong Cao, Hai Long Nguyen, and Francois Petitjean. Characterizing concept drift. *Data Mining and Knowledge Discovery*, 30(4):964–994, 2016.

[96] Andreas Weiler, Michael Grossniklaus, and Marc H Scholl. Situation monitoring of urban areas using social media data streams. *Information Systems*, 57:129–141, 2016.

[97] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[98] Clark Wissler. The spearman correlation formula. *Science*, 22(558):309–311, 1905.

[99] Siddharth Yadav and Tanmoy Chakraborty. Unsupervised sentiment analysis for code-mixed data. *arXiv preprint arXiv:2001.11384*, 2020.

[100] Michael Zhai, Johnny Tan, and Jinho Choi. Intrinsic and extrinsic evaluations of word embeddings. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[101] Bin Zhou, Yi Han, Jian Pei, Bin Jiang, Yufei Tao, and Yan Jia. Continuous privacy preserving publishing of data streams. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 648–659, 2009.

[102] Zhi-Hua Zhou. *Machine learning*. Springer Nature, 2021.

Annex A

Experiment Results by Task and Model

In this section, we present the results of our experiments with the RiverText framework, which involved varying the architectures, hyperparameter configurations, and intrinsic NLP tasks. However, due to the tables' size, we only present summary tables in this paper. The full tables can be found on our GitHub repository at `https://github.com/dccuchile/rivertext/tree/main/experiments`.

Table A.1: The table shows the results of the periodic evaluation of the ICBOW model and the Mturk dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ICBOW	300	3	10	0.4185	0.2705	...	0.4099	...	0.4583
ICBOW	300	3	8	0.4276	0.2758	...	0.3805	...	0.4484
ICBOW	300	3	6	0.4163	0.2386	...	0.4278	...	0.4711
ICBOW	300	2	10	0.4144	0.1621	...	0.4052	...	0.3672
ICBOW	300	2	8	0.4141	0.231	...	0.3971	...	0.4898
ICBOW	300	2	6	0.423	0.1731	...	0.4164	...	0.3876
ICBOW	300	1	10	0.3976	0.2604	...	0.3978	...	0.4382
ICBOW	300	1	8	0.37	0.1825	...	0.3449	...	0.4416
ICBOW	300	1	6	0.404	0.2788	...	0.3621	...	0.4516
ICBOW	200	3	10	0.4226	0.2222	...	0.4257	...	0.3856
ICBOW	200	3	8	0.4108	0.2027	...	0.3932	...	0.466
ICBOW	200	3	6	0.414	0.2353	...	0.4686	...	0.4233
ICBOW	200	2	10	0.3953	0.257	...	0.4008	...	0.3962
ICBOW	200	2	8	0.4123	0.2143	...	0.3317	...	0.4428
ICBOW	200	2	6	0.3925	0.2597	...	0.3579	...	0.3832
ICBOW	200	1	10	0.3573	0.1651	...	0.3731	...	0.376
ICBOW	200	1	8	0.3787	0.2696	...	0.3557	...	0.3598
ICBOW	200	1	6	0.3442	0.2141	...	0.3197	...	0.3639
ICBOW	100	3	10	0.419	0.2185	...	0.4334	...	0.4806
ICBOW	100	3	8	0.3888	0.1693	...	0.4502	...	0.4675
ICBOW	100	3	6	0.4394	0.1941	...	0.4224	...	0.5122
ICBOW	100	2	10	0.3883	0.1683	...	0.3832	...	0.4959
ICBOW	100	2	8	0.3869	0.2379	...	0.4358	...	0.4332
ICBOW	100	2	6	0.4067	0.1926	...	0.4104	...	0.4451
ICBOW	100	1	10	0.3574	0.1841	...	0.4272	...	0.3393
ICBOW	100	1	8	0.3501	0.2177	...	0.3319	...	0.3309
ICBOW	100	1	6	0.3683	0.1909	...	0.3478	...	0.409

Table A.2: The table shows the results of the periodic evaluation of the ICBOW model and the AP dataset for the categorization task, measured with purity clustering. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ICBOW	100	3	10	0.3024	0.2065	...	0.301	...	0.3383
ICBOW	100	3	8	0.2976	0.204	...	0.3109	...	0.3333
ICBOW	100	3	6	0.2942	0.2114	...	0.2935	...	0.3358
ICBOW	100	2	8	0.2922	0.2065	...	0.306	...	0.3109
ICBOW	200	3	8	0.291	0.2164	...	0.2711	...	0.3159
ICBOW	300	2	6	0.2902	0.1915	...	0.2861	...	0.306
ICBOW	300	3	6	0.289	0.204	...	0.2861	...	0.3159
ICBOW	300	2	10	0.2859	0.204	...	0.2861	...	306
ICBOW	300	3	8	0.2845	0.199	...	0.2985	...	0.3259
ICBOW	300	3	10	0.2843	0.194	...	0.2935	...	0.2985
ICBOW	100	2	10	0.2842	0.204	...	0.2836	...	0.3284
ICBOW	200	3	6	0.2838	0.1965	...	0.2811	...	0.3209
ICBOW	100	2	6	0.2836	0.2164	...	0.2612	...	0.306
ICBOW	200	3	10	0.2825	0.1891	...	0.2861	...	0.3358
ICBOW	300	2	8	0.2819	0.1915	...	0.2811	...	0.3234
ICBOW	200	2	6	0.2777	0.2189	...	0.2836	...	0.2985
ICBOW	300	1	6	0.2768	0.2015	...	0.2786	...	0.3383
ICBOW	200	2	8	0.2767	0.1866	...	0.2836	...	0.2736
ICBOW	300	1	8	0.2764	0.204	...	0.2736	...	0.3259
ICBOW	200	2	10	0.2732	0.194	...	0.2711	...	0.3134
ICBOW	300	1	10	0.2685	0.209	...	0.291	...	0.2786
ICBOW	200	1	6	0.2651	0.2139	...	0.2537	...	0.2886
ICBOW	200	1	8	0.2642	0.2065	...	0.2761	...	0.2811
ICBOW	100	1	10	0.2608	0.2065	...	0.2612	...	0.291
ICBOW	200	1	10	0.2607	0.209	...	0.2761	...	0.2612
ICBOW	100	1	6	0.2577	0.2065	...	0.2637	...	0.2637
ICBOW	100	1	8	0.2567	0.2264	...	0.2736	...	0.2761

Table A.3: The table shows the results of the periodic evaluation of the ISG model and the MEN dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ISG	300	3	10	0.3646	0.2975	...	0.3771	...	0.3796
ISG	300	3	8	354	0.3008	...	0.364	...	0.3534
ISG	300	3	6	0.3681	0.2843	...	0.4094	...	0.3927
ISG	300	2	10	0.3809	0.3111	...	0.3764	...	0.3825
ISG	300	2	8	0.3821	0.3048	...	0.3846	...	0.3963
ISG	300	2	6	0.3792	0.275	...	0.3823	...	0.3886
ISG	300	1	10	0.3947	0.3009	...	0.3954	...	0.3946
ISG	300	1	8	0.3982	0.3018	...	0.4131	...	0.4158
ISG	300	1	6	0.4073	0.3137	...	0.4048	...	0.4281
ISG	200	3	10	0.3828	0.2991	...	0.4018	...	0.3954
ISG	200	3	8	0.3721	0.3176	...	0.3574	...	0.371
ISG	200	3	6	0.3851	0.3194	...	0.3906	...	0.3903
ISG	200	2	10	0.3954	0.3133	...	0.4009	...	0.4118
ISG	200	2	8	0.3982	0.3312	...	0.4107	...	0.4092
ISG	200	2	6	0.3974	0.3169	...	0.3999	...	0.4165
ISG	200	1	10	0.41	0.3019	...	0.408	...	0.4241
ISG	200	1	8	0.408	0.3022	...	0.4201	...	0.4347
ISG	200	1	6	0.4133	0.2893	...	0.4071	...	0.4406
ISG	100	3	10	0.392	0.3069	...	0.3858	...	0.3974
ISG	100	3	8	0.4121	0.3128	...	0.4227	...	0.4432
ISG	100	3	6	0.4055	0.3248	...	0.4174	...	0.4182
ISG	100	2	10	0.421	0.3207	...	0.4339	...	0.4355
ISG	100	2	8	0.4227	0.3315	...	0.4102	...	0.4383
ISG	100	2	6	0.4267	0.3411	...	0.4377	...	0.4462
ISG	100	1	10	0.4418	0.3368	...	0.4635	...	0.4697
ISG	100	1	8	0.4396	0.3299	...	0.4421	...	0.4714
ISG	100	1	6	0.4427	0.3178	...	0.4543	...	0.4759

Table A.4: The table shows the results of the periodic evaluation of the ISG model and the Mturk dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ISG	300	3	10	0.317	0.2161	...	0.2819	...	0.3279
ISG	300	3	8	0.3117	0.1782	...	0.2709	...	0.2704
ISG	300	3	6	0.3406	0.1838	...	0.3452	...	0.2674
ISG	300	2	10	0.3372	0.221	...	0.4176	...	0.3231
ISG	300	2	8	0.3452	0.1428	...	0.3973	...	0.3189
ISG	300	2	6	0.3353	0.2175	...	0.3276	...	0.3201
ISG	300	1	10	0.3809	0.2527	...	0.3859	...	0.317
ISG	300	1	8	0.3694	0.1444	...	0.3511	...	0.3729
ISG	300	1	6	0.364	0.2389	...	0.3603	...	0.3993
ISG	200	3	10	0.3452	0.2448	...	0.327	...	0.2905
ISG	200	3	8	0.3555	0.277	...	0.3232	...	0.326
ISG	200	3	6	0.3399	0.2248	...	0.387	...	0.2831
ISG	200	2	10	0.3459	0.2506	...	0.354	...	0.2978
ISG	200	2	8	0.3412	0.2498	...	0.2948	...	0.3875
ISG	200	2	6	0.3676	0.2476	...	0.3806	...	0.365
ISG	200	1	10	0.3705	0.2486	...	0.3176	...	0.3605
ISG	200	1	8	0.3716	0.1687	...	0.3769	...	0.4266
ISG	200	1	6	0.3923	0.2681	...	0.4274	...	0.4347
ISG	100	3	10	0.3585	0.1762	...	0.3202	...	0.2756
ISG	100	3	8	0.3712	0.2358	...	0.4196	...	0.372
ISG	100	3	6	0.3452	0.154	...	0.3542	...	0.3738
ISG	100	2	10	0.3987	0.2549	...	0.3707	...	0.3531
ISG	100	2	8	0.3916	0.293	...	0.4189	...	0.3488
ISG	100	2	6	0.3815	0.308	...	0.399	...	0.3824
ISG	100	1	10	0.4163	0.3213	...	0.4711	...	0.3737
ISG	100	1	8	0.3996	0.1543	...	0.4413	...	0.4194
ISG	100	1	6	0.3932	0.206	...	0.3831	...	0.392

Table A.5: The table shows the results of the periodic evaluation of the ISG model and the AP dataset for the categorization task, measured with purity clustering. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
ISG	300	3	10	0.317	0.2161	...	0.2819	...	0.3279
ISG	300	3	8	0.3117	0.1782	...	0.2709	...	0.2704
ISG	300	3	6	0.3406	0.1838	...	0.3452	...	0.2674
ISG	300	2	10	0.3372	0.221	...	0.4176	...	0.3231
ISG	300	2	8	0.3452	0.1428	...	0.3973	...	0.3189
ISG	300	2	6	0.3353	0.2175	...	0.3276	...	0.3201
ISG	300	1	10	0.3809	0.2527	...	0.3859	...	0.317
ISG	300	1	8	0.3694	0.1444	...	0.3511	...	0.3729
ISG	300	1	6	0.364	0.2389	...	0.3603	...	0.3993
ISG	200	3	10	0.3452	0.2448	...	0.327	...	0.2905
ISG	200	3	8	0.3555	0.277	...	0.3232	...	0.326
ISG	200	3	6	0.3399	0.2248	...	0.387	...	0.2831
ISG	200	2	10	0.3459	0.2506	...	0.354	...	0.2978
ISG	200	2	8	0.3412	0.2498	...	0.2948	...	0.3875
ISG	200	2	6	0.3676	0.2476	...	0.3806	...	0.365
ISG	200	1	10	0.3705	0.2486	...	0.3176	...	0.3605
ISG	200	1	8	0.3716	0.1687	...	0.3769	...	0.4266
ISG	200	1	6	0.3923	0.2681	...	0.4274	...	0.4347
ISG	100	3	10	0.3585	0.1762	...	0.3202	...	0.2756
ISG	100	3	8	0.3712	0.2358	...	0.4196	...	0.372
ISG	100	3	6	0.3452	0.154	...	0.3542	...	0.3738
ISG	100	2	10	0.3987	0.2549	...	0.3707	...	0.3531
ISG	100	2	8	0.3916	0.293	...	0.4189	...	0.3488
ISG	100	2	6	0.3815	0.308	...	0.399	...	0.3824
ISG	100	1	10	0.4163	0.3213	...	0.4711	...	0.3737
ISG	100	1	8	0.3996	0.1543	...	0.4413	...	0.4194
ISG	100	1	6	0.3932	0.206	...	0.3831	...	0.392

Table A.6: The table shows the results of the periodic evaluation of the IWCM model and the MEN dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/.. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
IWCM	300	3	1000	0.439	0.2346	...	0.458	...	0.500
IWCM	300	3	750	0.4275	0.2211	...	0.4507	...	0.4825
IWCM	300	3	500	0.4314	0.2364	...	0.4551	...	0.4786
IWCM	300	2	1000	0.4203	0.2251	...	0.4357	...	0.4846
IWCM	300	2	750	0.4042	0.2176	...	0.4204	...	0.4660
IWCM	300	2	500	0.4033	0.2216	...	0.4187	...	0.4592
IWCM	300	1	1000	0.3917	0.1828	...	0.4118	...	0.4573
IWCM	300	1	750	0.3625	0.1627	...	0.3811	...	0.4261
IWCM	300	1	500	0.3564	0.1647	...	0.374	...	0.408
IWCM	200	3	1000	0.4384	0.2217	...	0.455	...	0.5007
IWCM	200	3	750	0.4276	0.2197	...	0.4512	...	0.4866
IWCM	200	3	500	0.4325	0.2345	...	0.4571	...	0.4781
IWCM	200	2	1000	0.4193	0.2133	...	0.4345	...	0.4830
IWCM	200	2	750	0.4032	0.2013	...	0.4153	...	0.4654
IWCM	200	2	500	0.4043	0.2208	...	0.4243	...	0.4532
IWCM	200	1	1000	0.3868	0.1796	...	0.4049	...	0.4473
IWCM	200	1	750	0.3597	0.1722	...	0.3729	...	0.4205
IWCM	200	1	500	0.3511	0.1705	...	0.3658	...	0.4023
IWCM	100	3	1000	0.4396	0.2211	...	0.4607	...	0.4996
IWCM	100	3	750	0.4289	0.2112	...	0.4475	...	0.4871
IWCM	100	3	500	0.4302	0.216	...	0.4539	...	0.4766
IWCM	100	2	1000	0.4175	0.2143	...	0.4313	...	0.4804
IWCM	100	2	750	0.4043	0.2006	...	0.4208	...	0.4627
IWCM	100	2	500	0.4000	0.205	...	0.4203	...	0.4489
IWCM	100	1	1000	0.3787	0.1762	...	0.4013	...	0.4402
IWCM	100	1	750	0.3543	0.1703	...	0.3717	...	0.4136
IWCM	100	1	500	0.3442	0.1690	...	0.3568	...	0.3916

Table A.7: The table shows the results of the periodic evaluation of the IWCM model and the Mturk dataset for the similarity task, measured with Spearman's correlation. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
IWCM	300	3	1000	0.3496	0.2388	...	0.3631	...	0.3926
IWCM	300	3	750	0.3386	0.2582	...	0.3346	...	0.3598
IWCM	300	3	500	0.3235	0.2239	...	0.3395	...	0.3519
IWCM	300	2	1000	0.3343	0.1887	...	0.3351	...	0.3674
IWCM	300	2	750	0.3352	0.2128	...	0.3414	...	0.3627
IWCM	300	2	500	0.3231	0.2396	...	0.3452	...	0.2989
IWCM	300	1	1000	0.3462	0.2097	...	0.3656	...	0.36
IWCM	300	1	750	0.3737	0.2188	...	0.3925	...	0.3587
IWCM	300	1	500	0.3577	0.2534	...	0.353	...	0.3006
IWCM	200	3	1000	0.3508	0.262	...	0.3379	...	0.3836
IWCM	200	3	750	0.3392	0.2513	...	0.3285	...	0.3617
IWCM	200	3	500	0.331	0.2284	...	0.3355	...	0.3404
IWCM	200	2	1000	0.3434	0.2469	...	0.3338	...	0.3973
IWCM	200	2	750	0.3389	0.2351	...	0.3367	...	0.3727
IWCM	200	2	500	0.3337	0.2216	...	0.3613	...	0.315
IWCM	200	1	1000	0.3409	0.1869	...	0.3694	...	0.3422
IWCM	200	1	750	0.3722	0.1966	...	0.3906	...	0.3467
IWCM	200	1	500	0.3618	0.2327	...	0.3478	...	0.3265
IWCM	100	3	1000	0.343	0.2461	...	0.3412	...	0.3739
IWCM	100	3	750	0.3359	0.2859	...	0.3386	...	0.373
IWCM	100	3	500	0.3381	0.2619	...	0.3478	...	0.3487
IWCM	100	2	1000	0.3402	0.2437	...	0.3387	...	0.3725
IWCM	100	2	750	0.3397	0.2635	...	0.3165	...	0.3689
IWCM	100	2	500	0.3413	0.3014	...	0.3706	...	0.3215
IWCM	100	1	1000	0.3292	0.2256	...	0.3333	...	0.3411
IWCM	100	1	750	0.344	0.2512	...	0.3402	...	0.3475
IWCM	100	1	500	0.3471	0.2714	...	0.3642	...	0.3501

Table A.8: The table shows the results of the periodic evaluation of the IWCM model and the AP dataset for the categorization task, measured with purity clustering. Although the full table is too large for this paper, it can be accessed on the documentation page at https://dccuchile.github.io/rivertext/. The bold remark represents the best result on average.

Model	Emb. size	Window size	Num. N. S.	Mean	T1	...	T15	...	T31
IWCM	300	3	1000	0.3069	0.2139	...	301	...	0.3209
IWCM	300	3	750	0.3032	0.2114	...	0.2985	...	0.3184
IWCM	300	3	500	0.2855	0.1915	...	296	...	0.3134
IWCM	300	2	1000	0.3084	0.2015	...	0.3234	...	0.3483
IWCM	300	2	750	0.3003	0.2264	...	0.3035	...	0.3557
IWCM	300	2	500	0.2811	0.2015	...	0.2861	...	0.301
IWCM	300	1	1000	0.2974	0.2114	...	0.3184	...	0.3458
IWCM	300	1	750	0.2764	0.2139	...	0.2836	...	0.2836
IWCM	300	1	500	0.2614	0.194	...	0.2562	...	0.2836
IWCM	200	3	1000	0.3074	0.2264	...	0.3333	...	0.3284
IWCM	200	3	750	0.3107	0.2139	...	0.3234	...	0.3284
IWCM	200	3	500	0.29	0.199	...	0.2985	...	0.3507
IWCM	200	2	1000	0.3083	0.2139	...	0.3209	...	0.3184
IWCM	200	2	750	0.309	0.199	...	0.3259	...	0.3582
IWCM	200	2	500	0.2853	0.2164	...	0.306	...	0.3085
IWCM	200	1	1000	0.3022	0.1965	...	0.3109	...	0.3333
IWCM	200	1	750	0.2834	0.204	...	0.3035	...	0.301
IWCM	200	1	500	0.2668	0.2214	...	0.2736	...	0.3109
IWCM	100	3	1000	0.3194	0.2264	...	0.3284	...	0.3308
IWCM	100	3	750	0.3181	0.2189	...	0.3234	...	0.3209
IWCM	100	3	500	0.2972	0.2239	...	0.2886	...	0.3134
IWCM	100	2	1000	0.3176	0.2214	...	0.3358	...	0.3483
IWCM	100	2	750	0.3189	0.1915	...	0.3358	...	0.3507
IWCM	100	2	500	0.2935	0.2189	...	0.2886	...	0.3109
IWCM	100	1	1000	0.3076	0.209	...	0.3184	...	0.3308
IWCM	100	1	750	0.2898	0.209	...	0.2836	...	0.3234
IWCM	100	1	500	0.2848	0.2239	...	0.296	...	0.3209

Annex B

Time serie plots

In this section, we present the results of different hyperparameter settings, considering the number of instances trained per period, in the three test datasets that were studied. For each set of results, the period p was set at 3,200,000 instances.

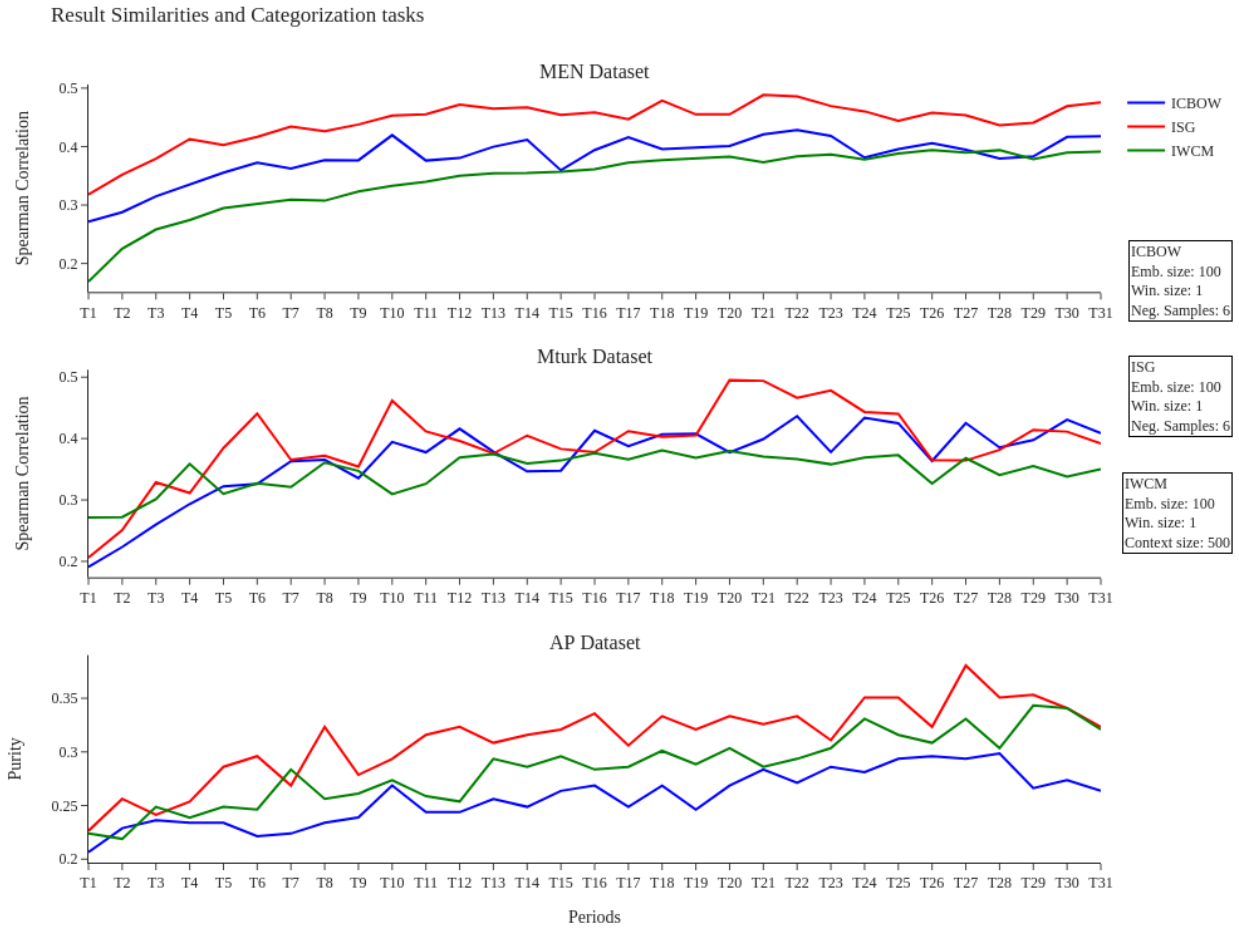

Figure B.1: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 1$, $ns\_samples = 1$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

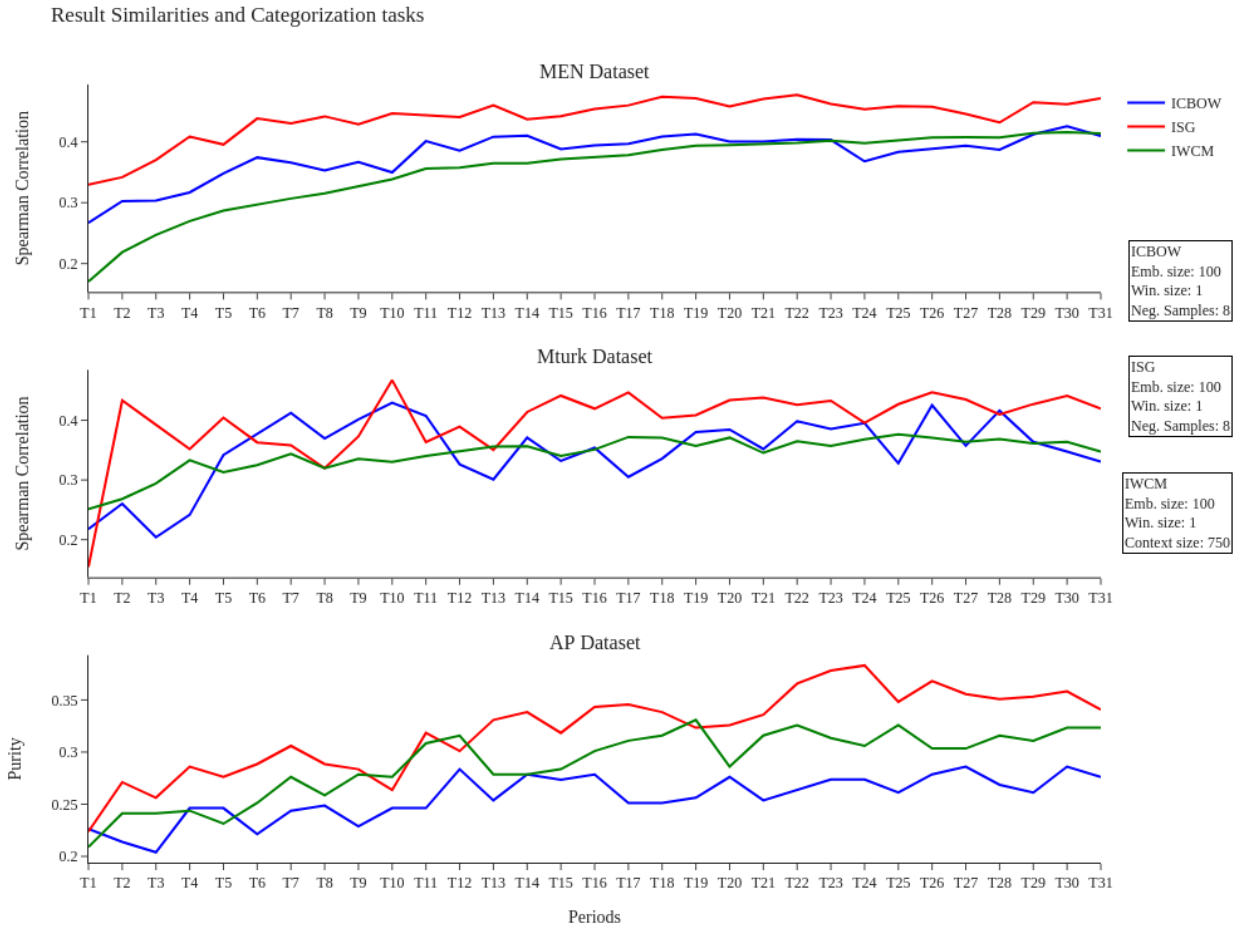

Figure B.2: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb_size = 100$, $window_size = 1$, $ns_samples = 8$, and $context_size = 750$across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

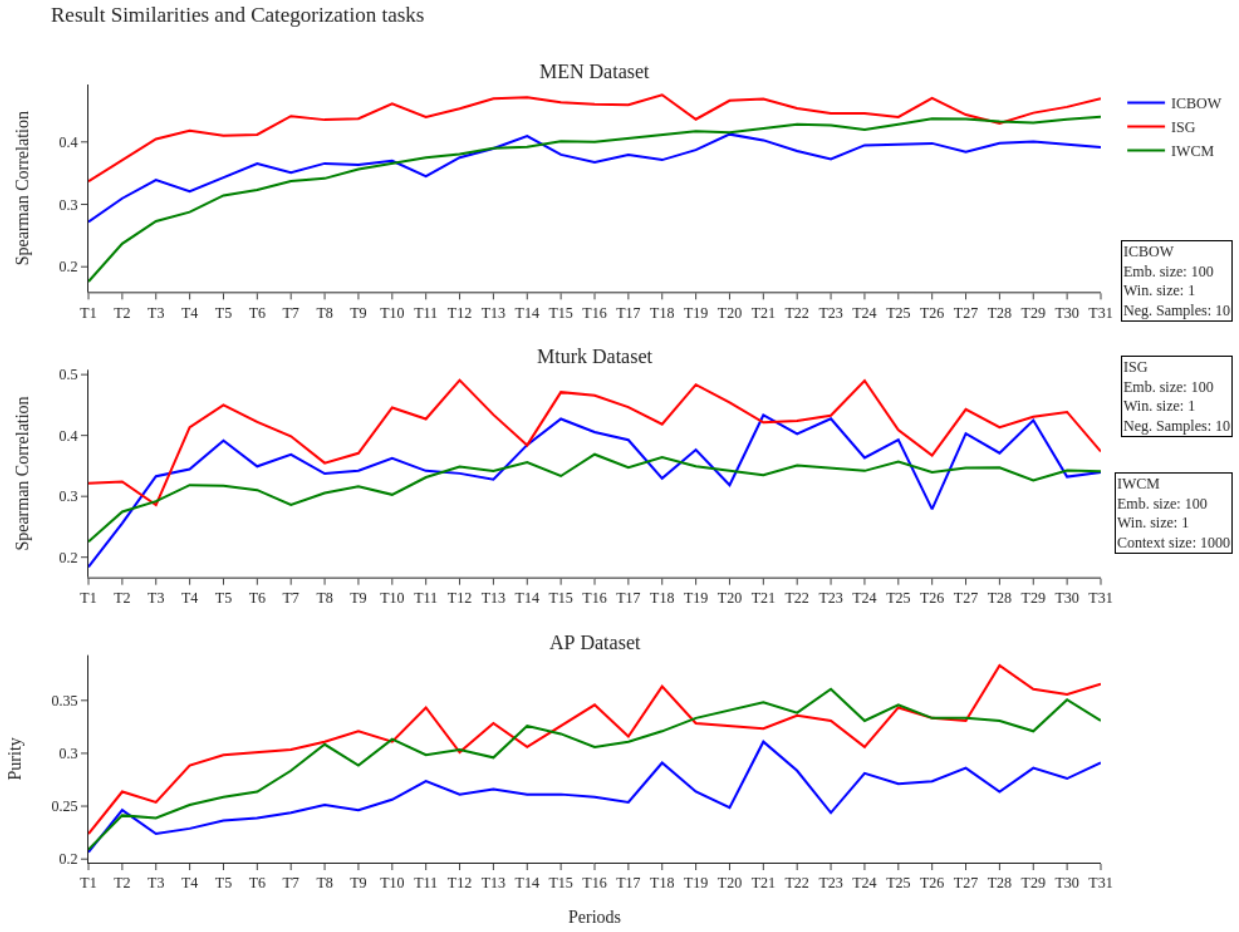

Figure B.3: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb_size = 100$, $window_size = 1$, $ns_samples = 10$, and $context_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

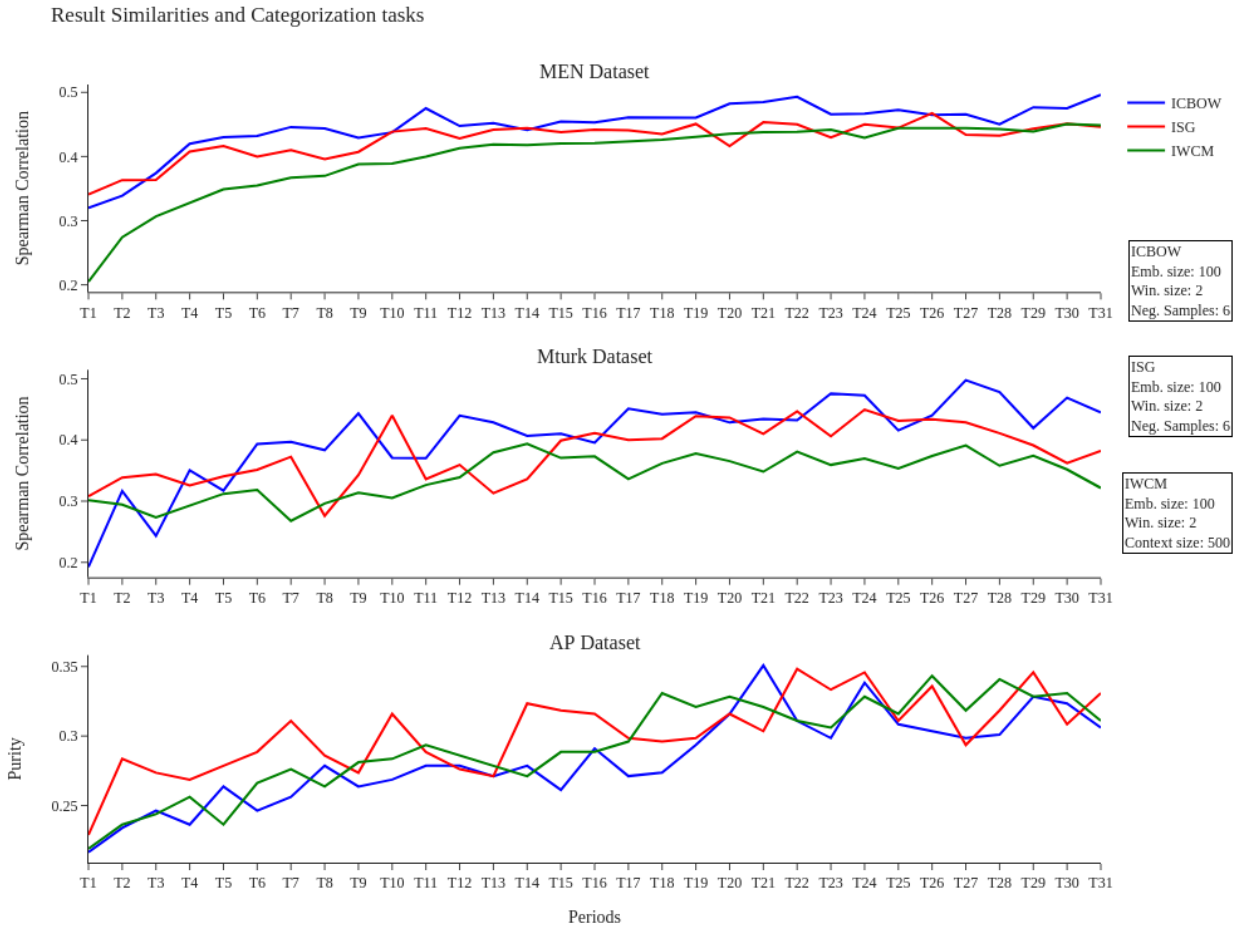

Figure B.4: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 2$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

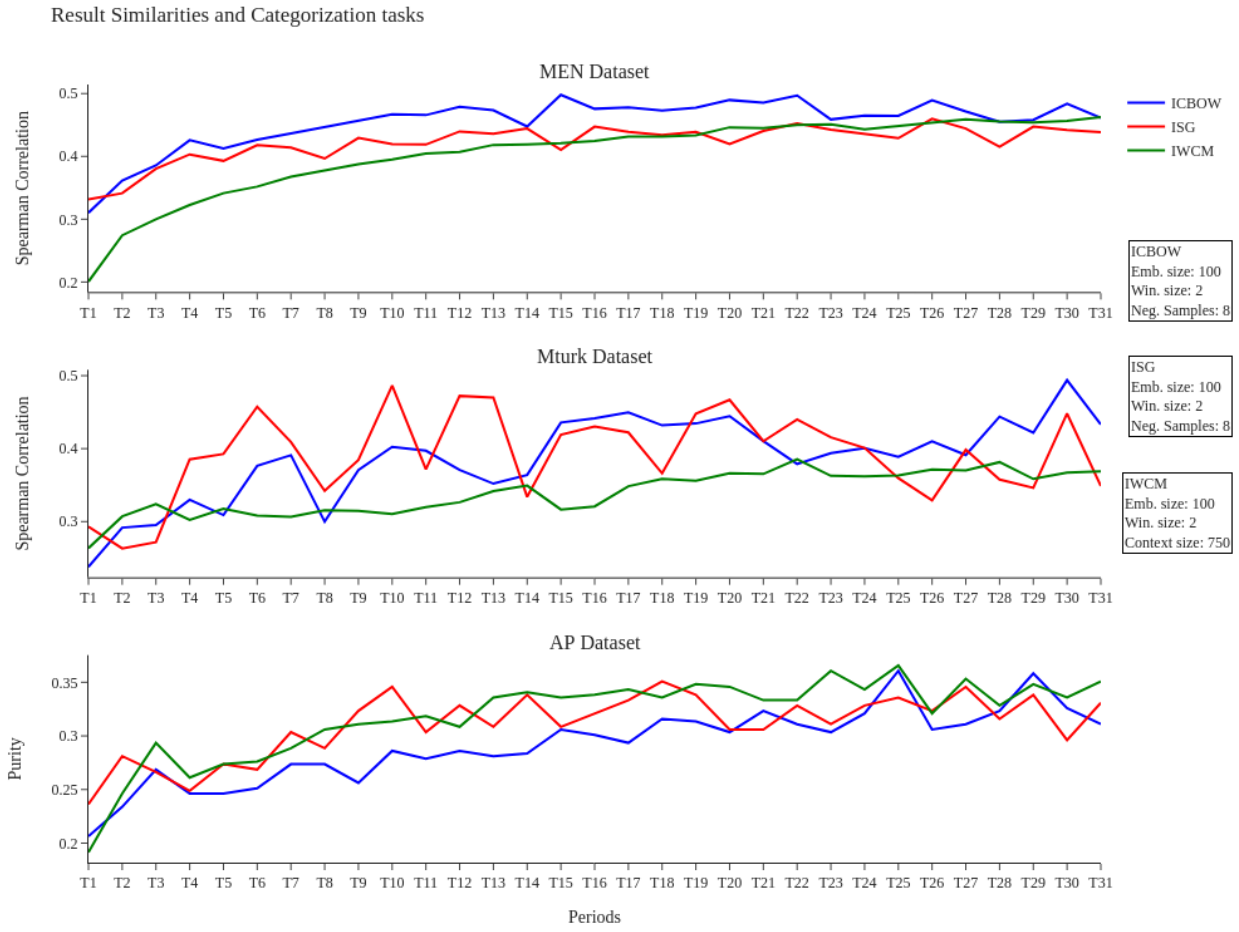

Figure B.5: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 2$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

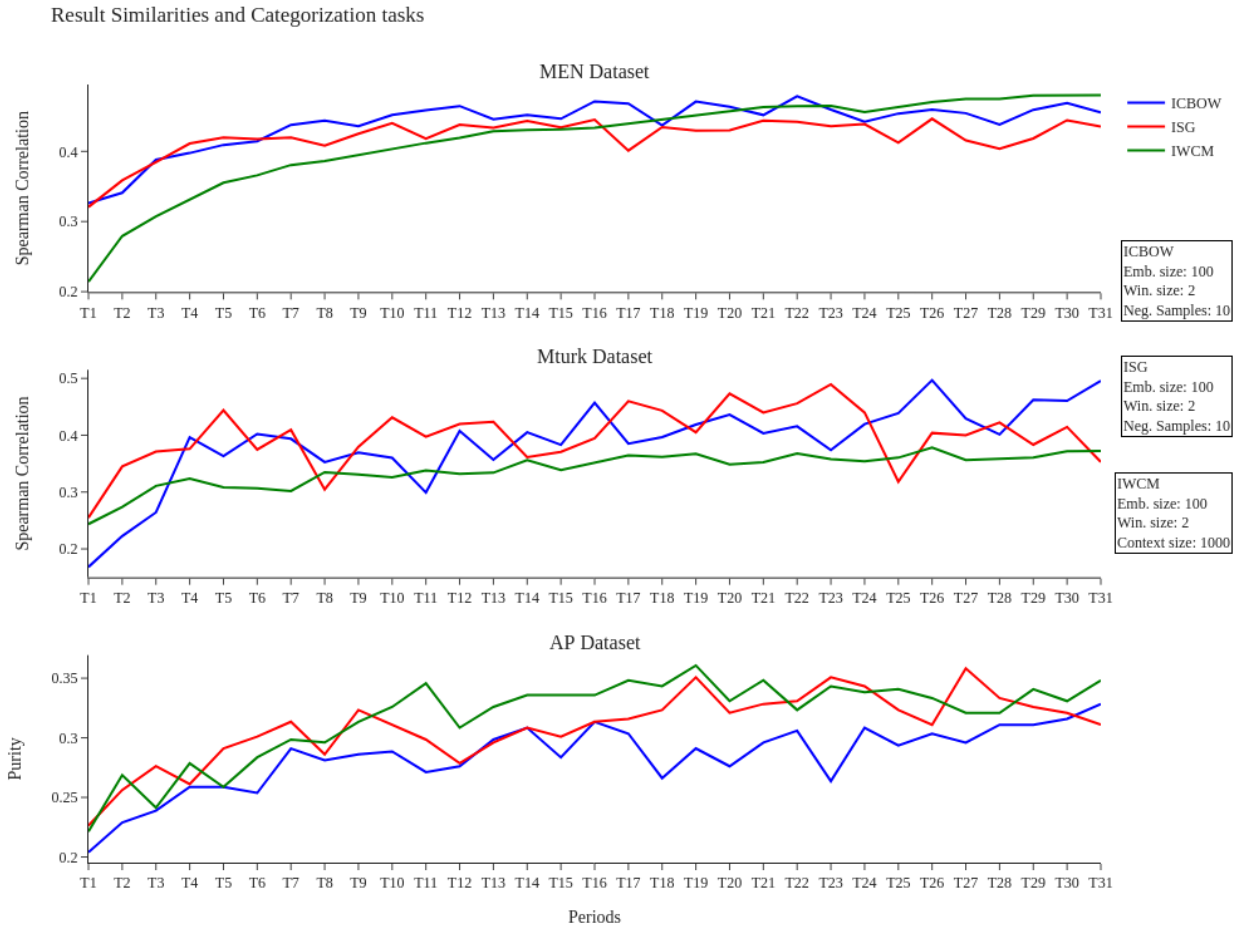

Figure B.6: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 2$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

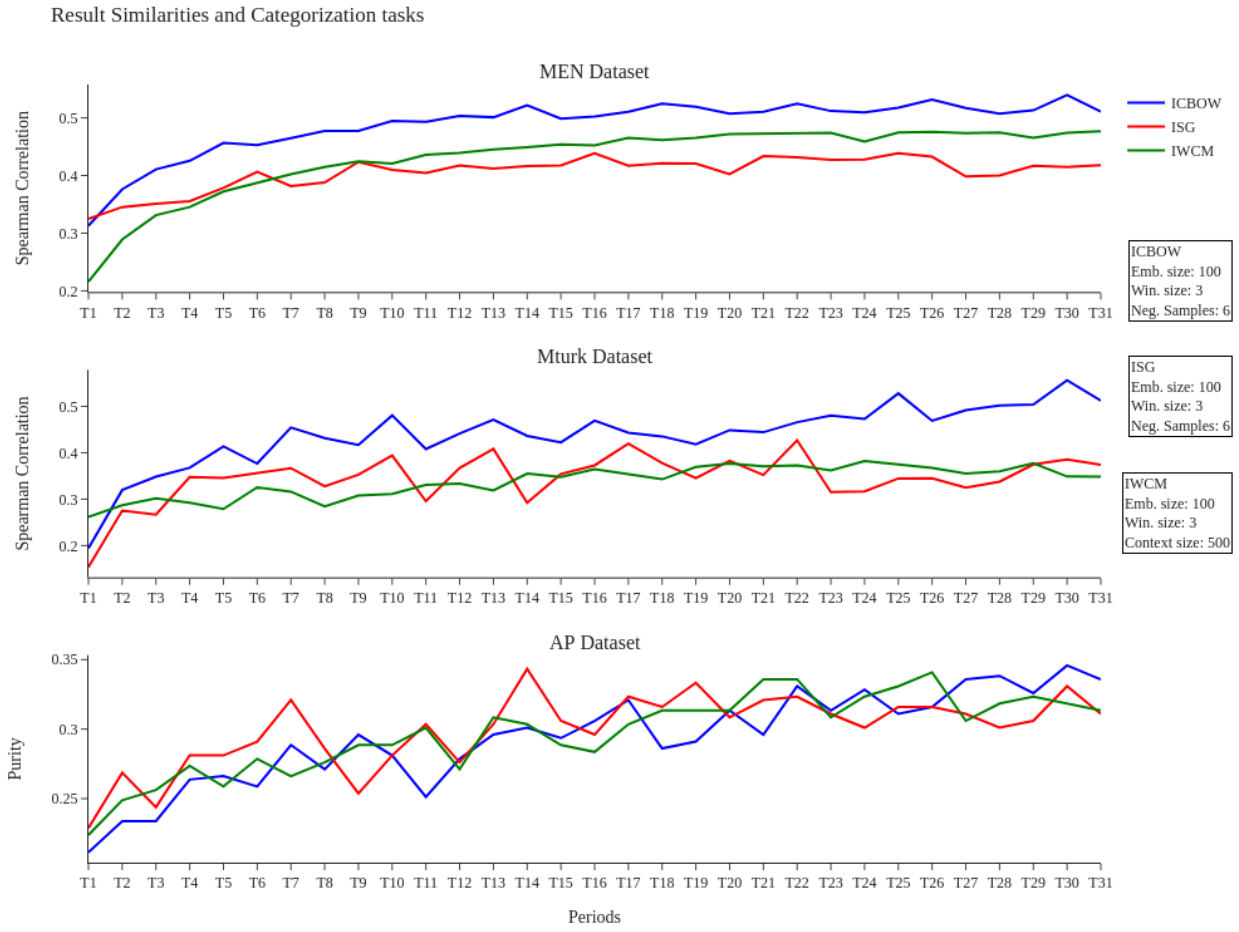

Figure B.7: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 3$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

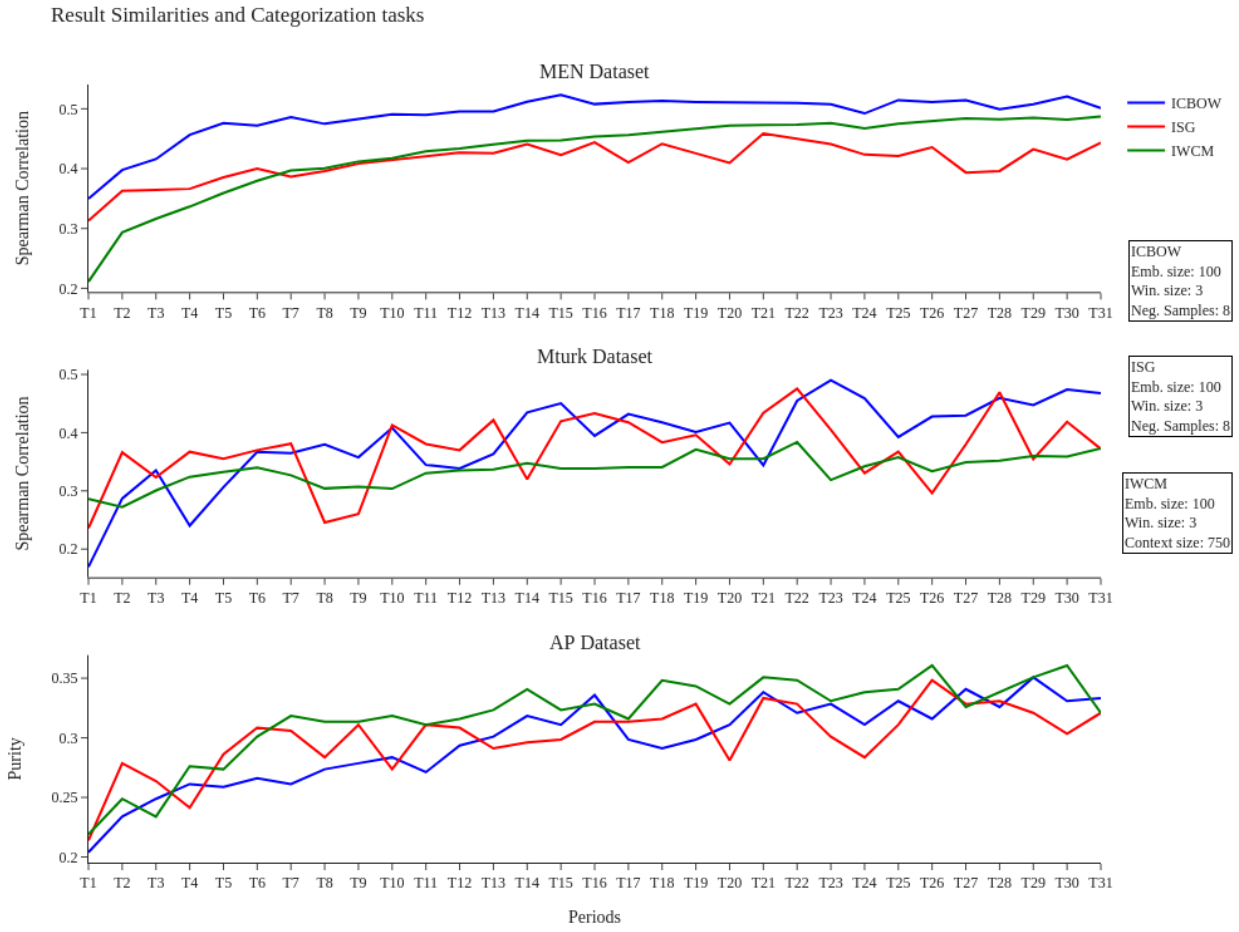

Figure B.8: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 3$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

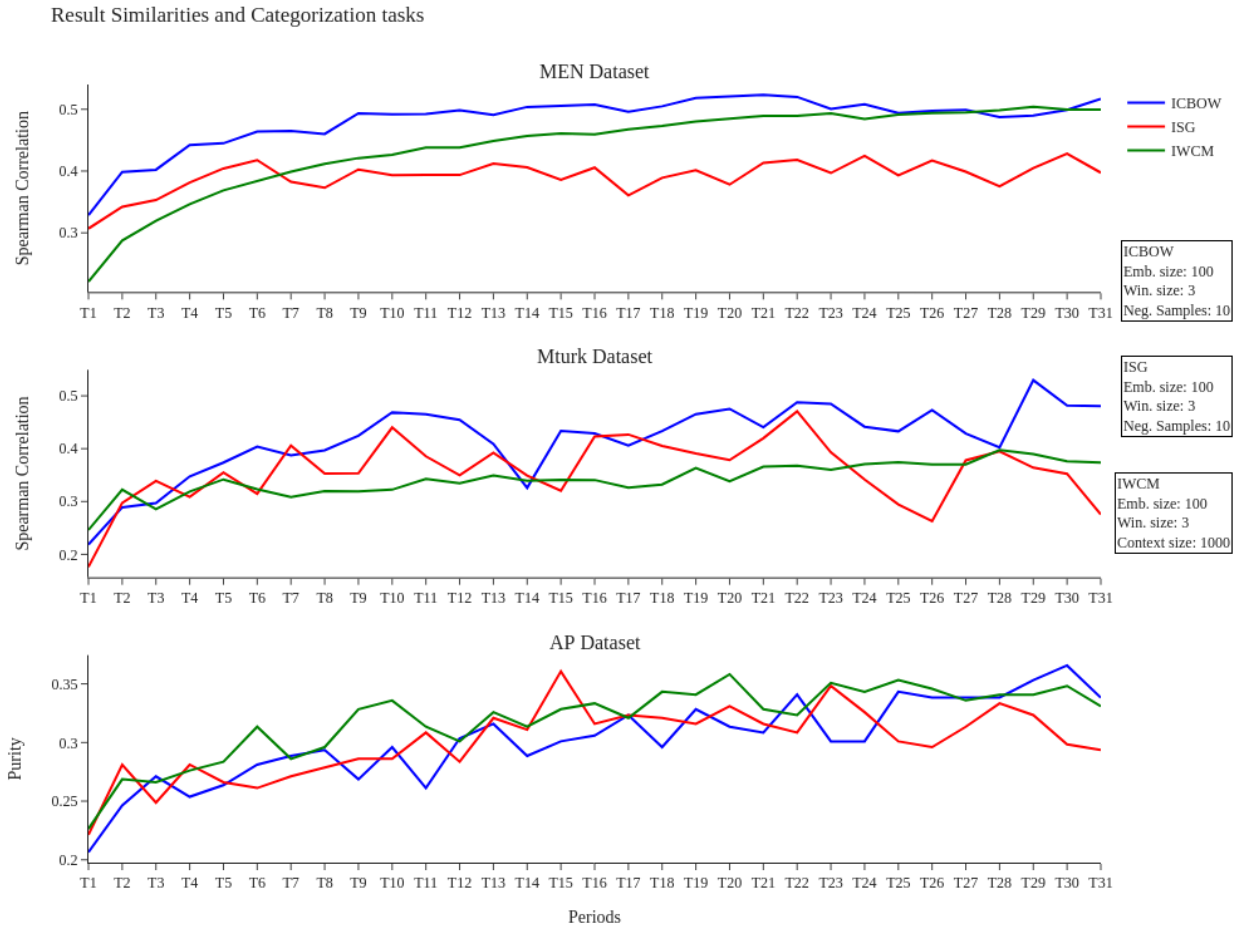

Figure B.9: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 100$, $window\_size = 3$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

Figure B.10: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 1$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

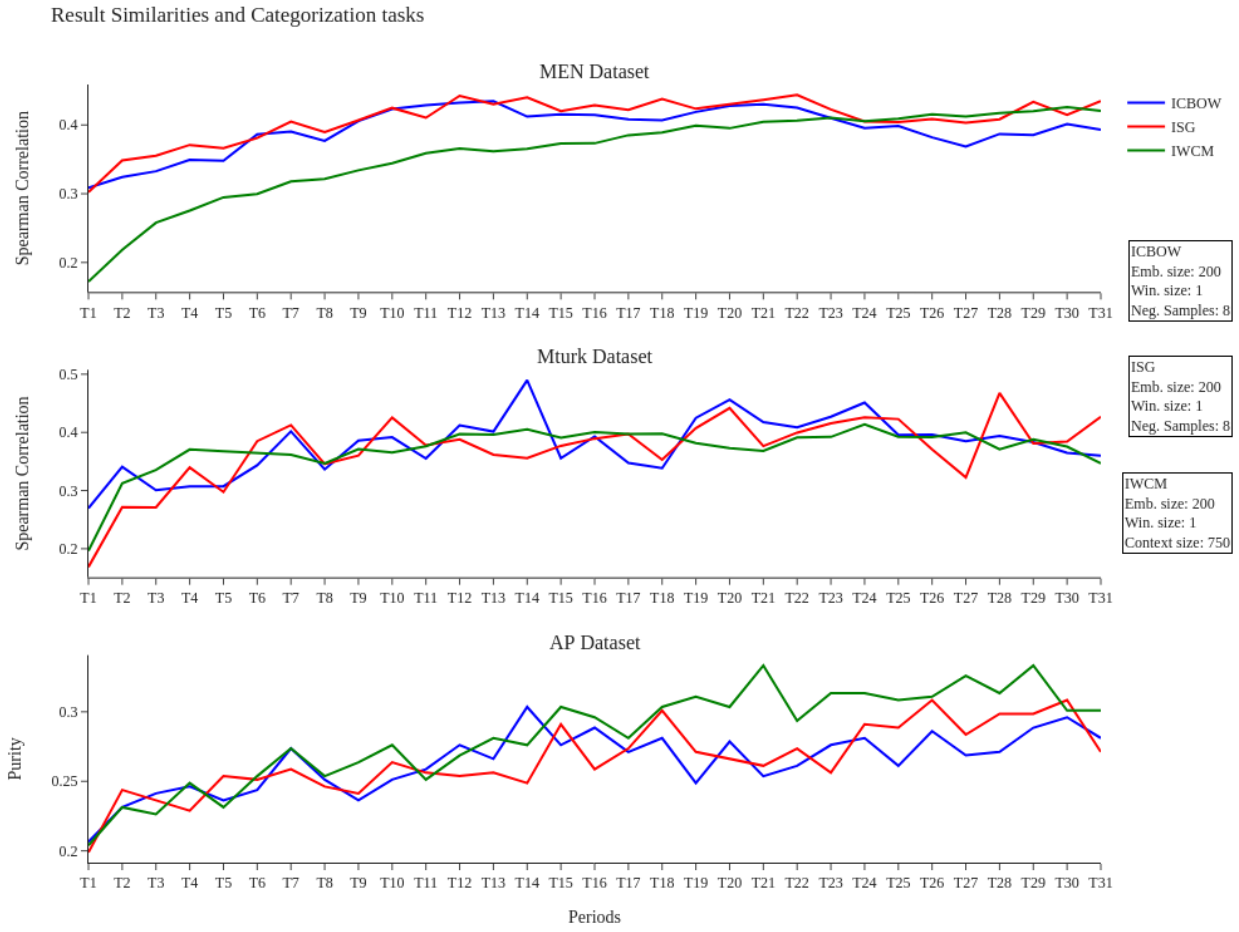

Figure B.11: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 1$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

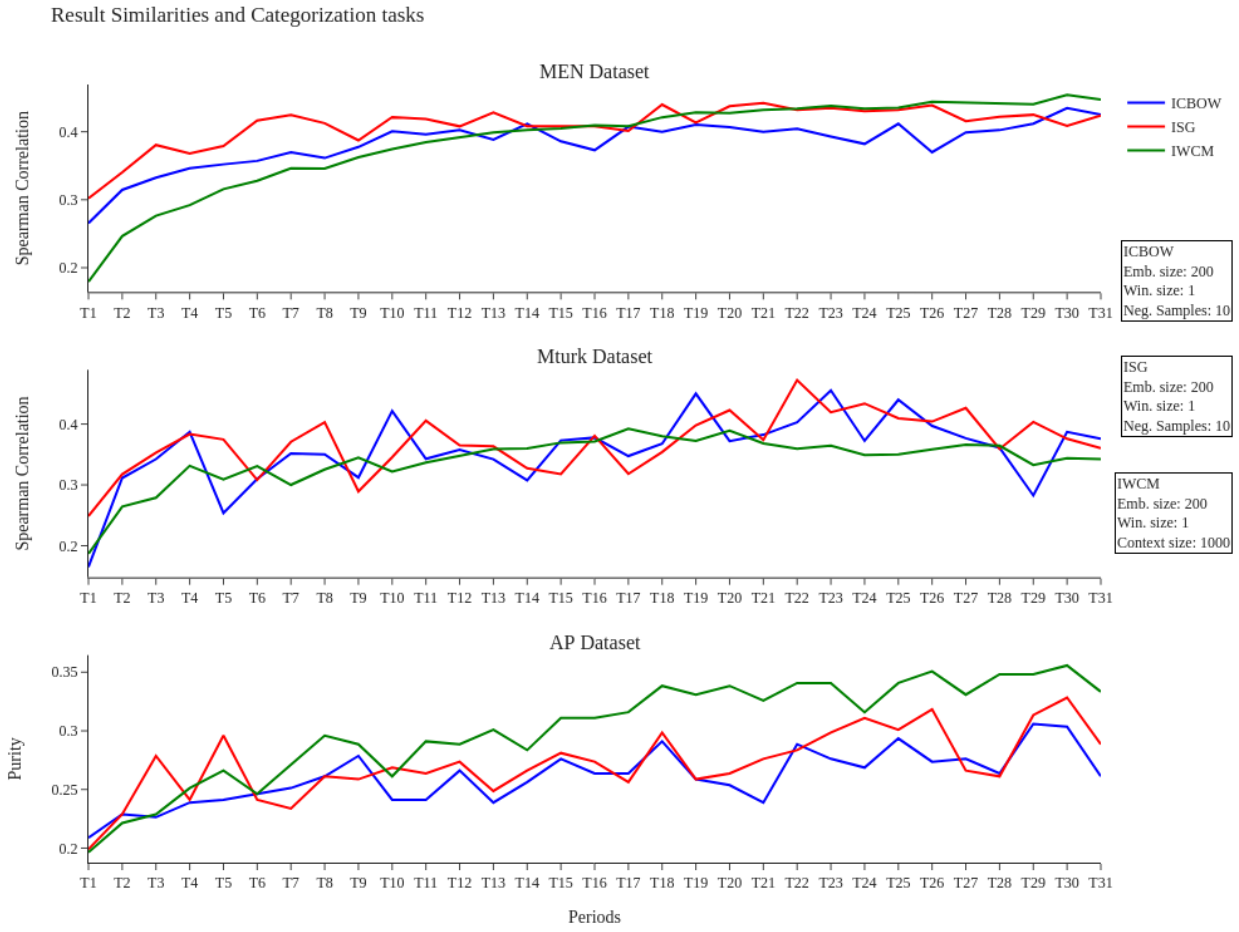

Figure B.12: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 1$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

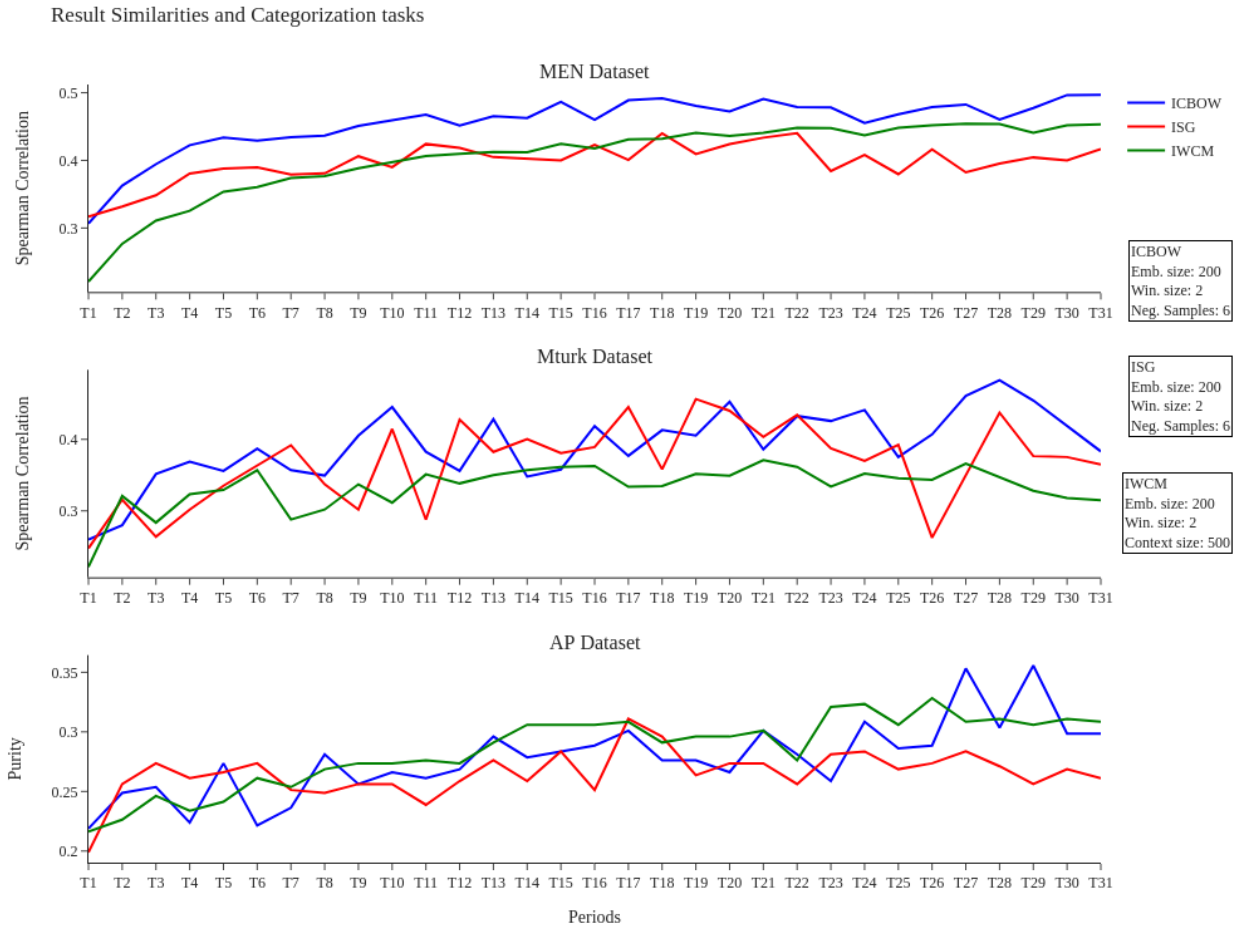

Figure B.13: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 2$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

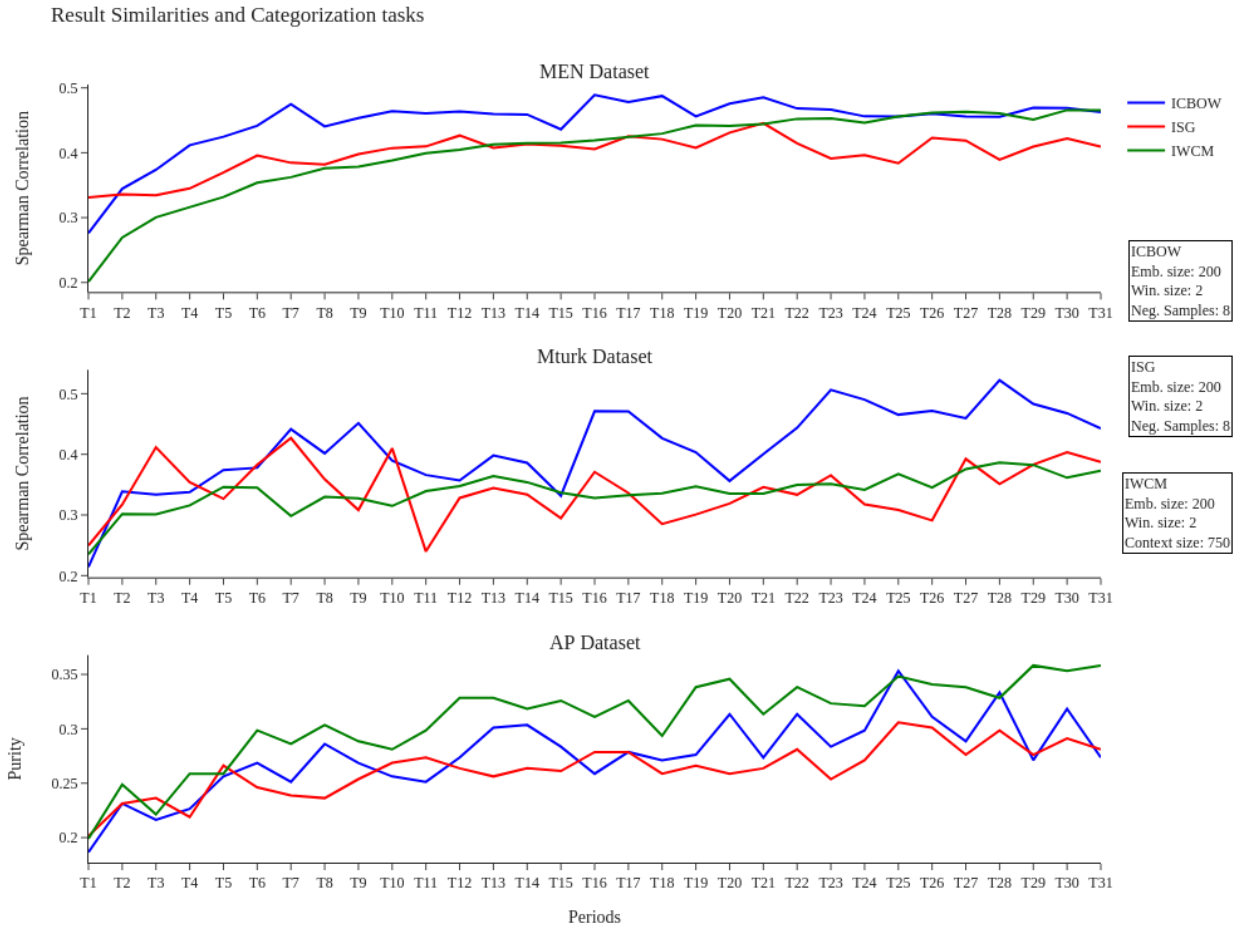

Figure B.14: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 2$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

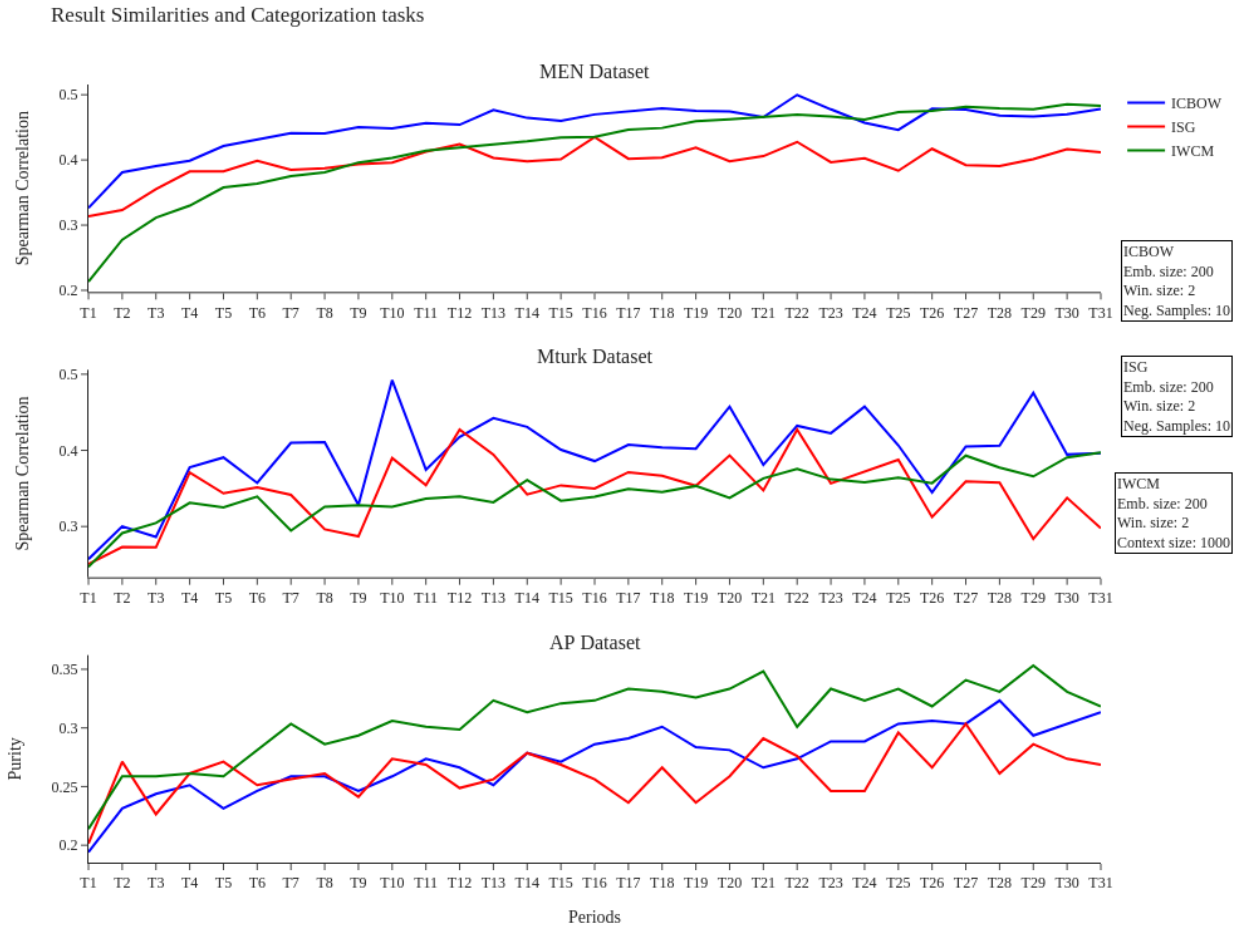

Figure B.15: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 2$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

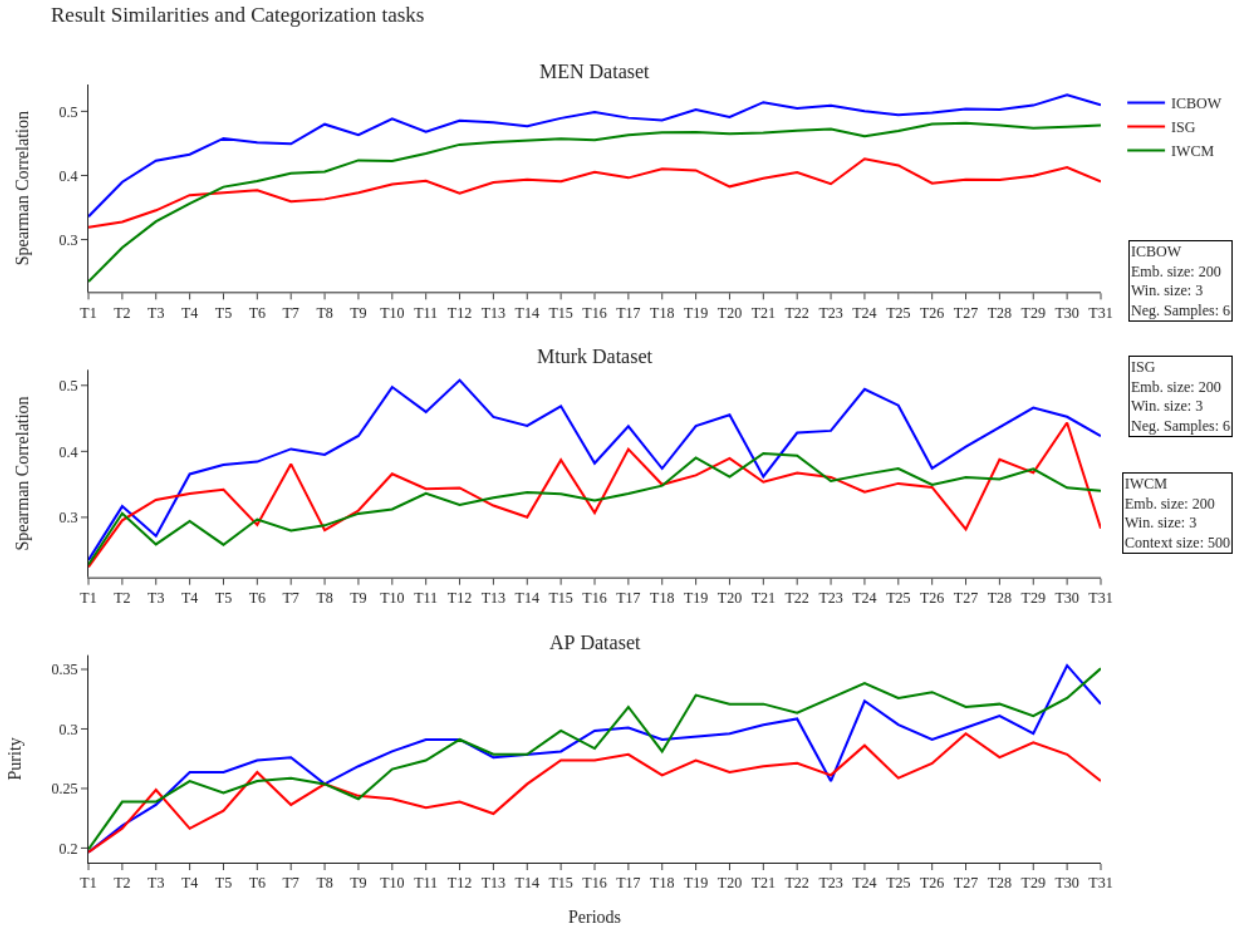

Figure B.16: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 3$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

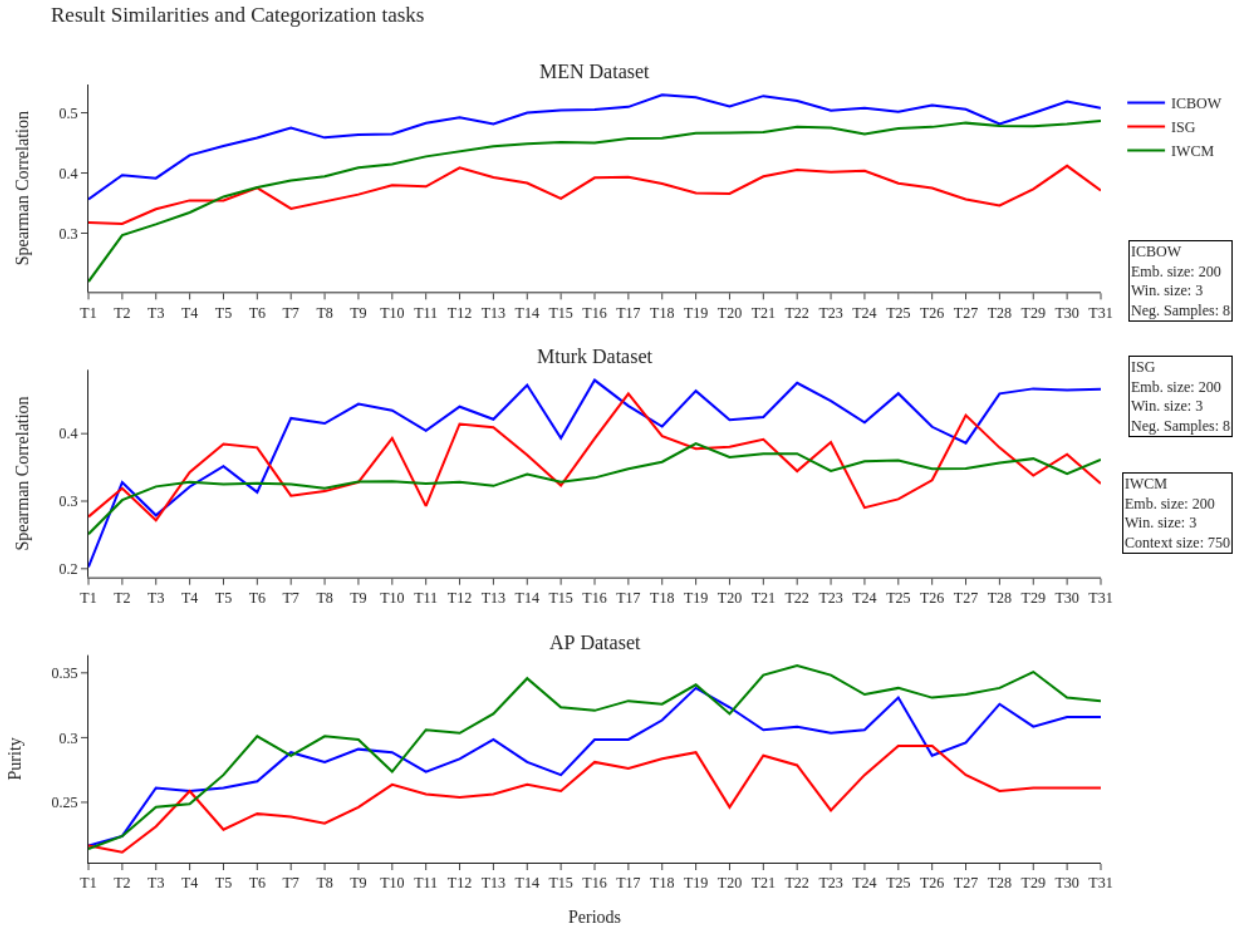

Figure B.17: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 3$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

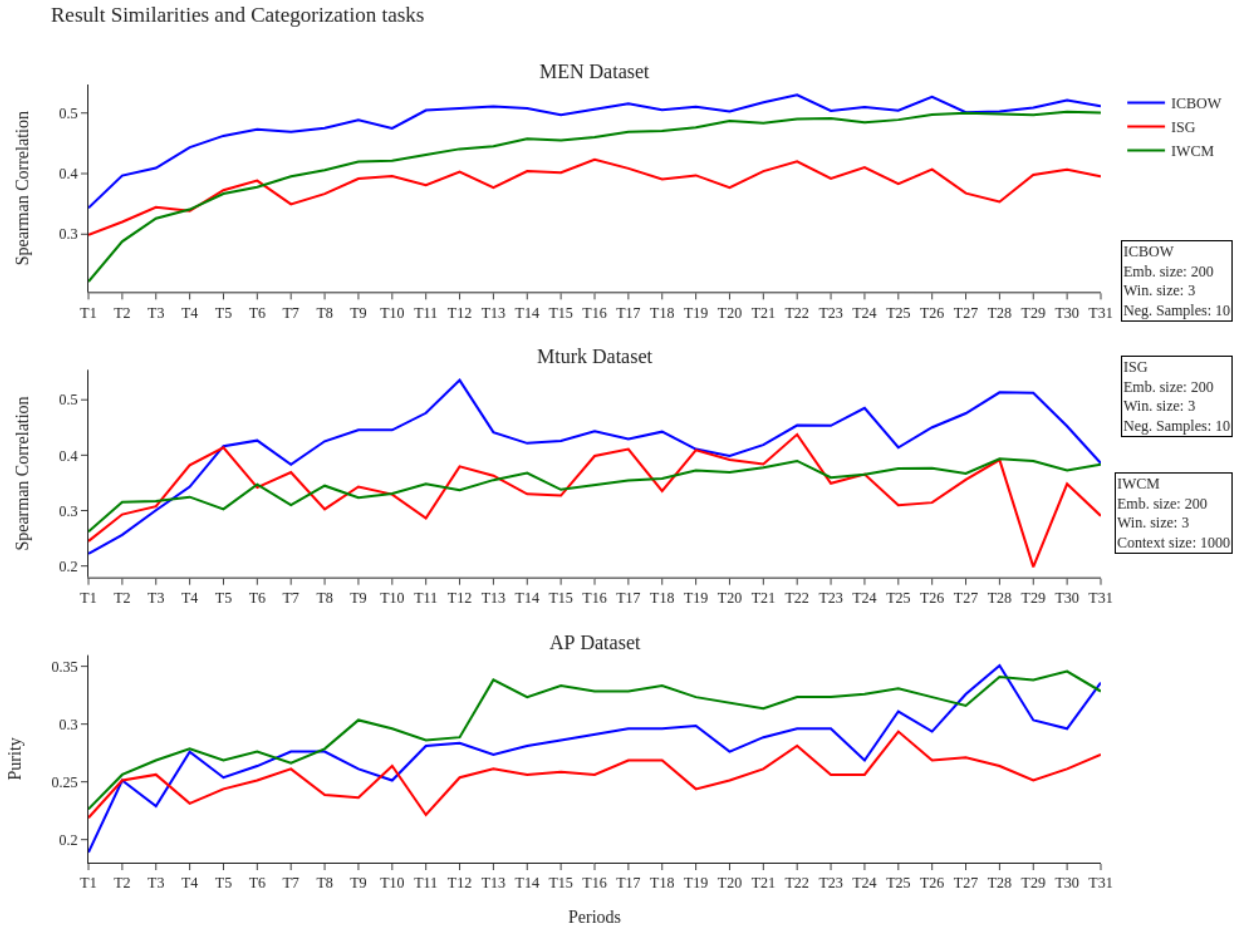

Figure B.18: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 200$, $window\_size = 3$, $ns\_samples = 10$, and $context_size = 1000$across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

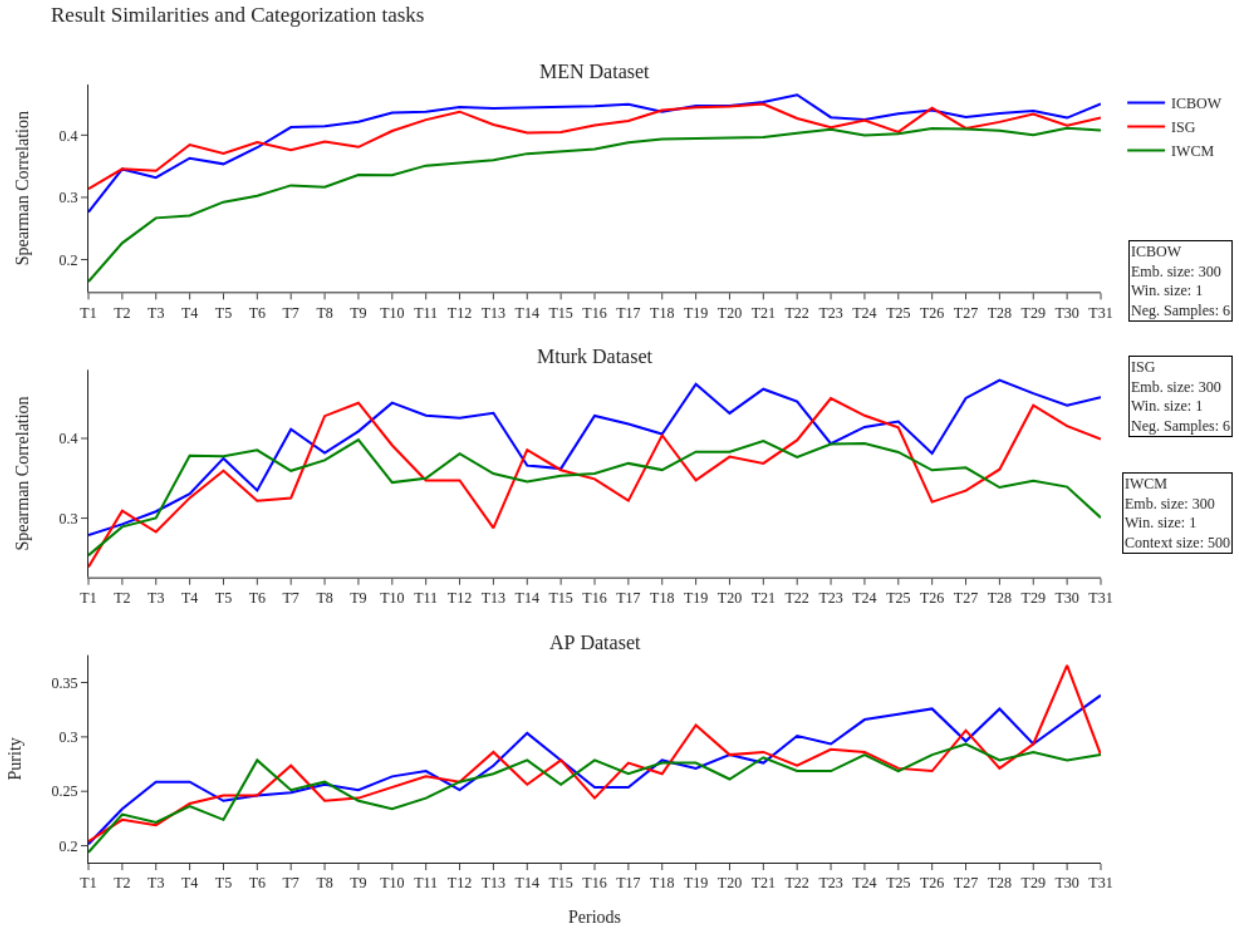

Figure B.19: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 1$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

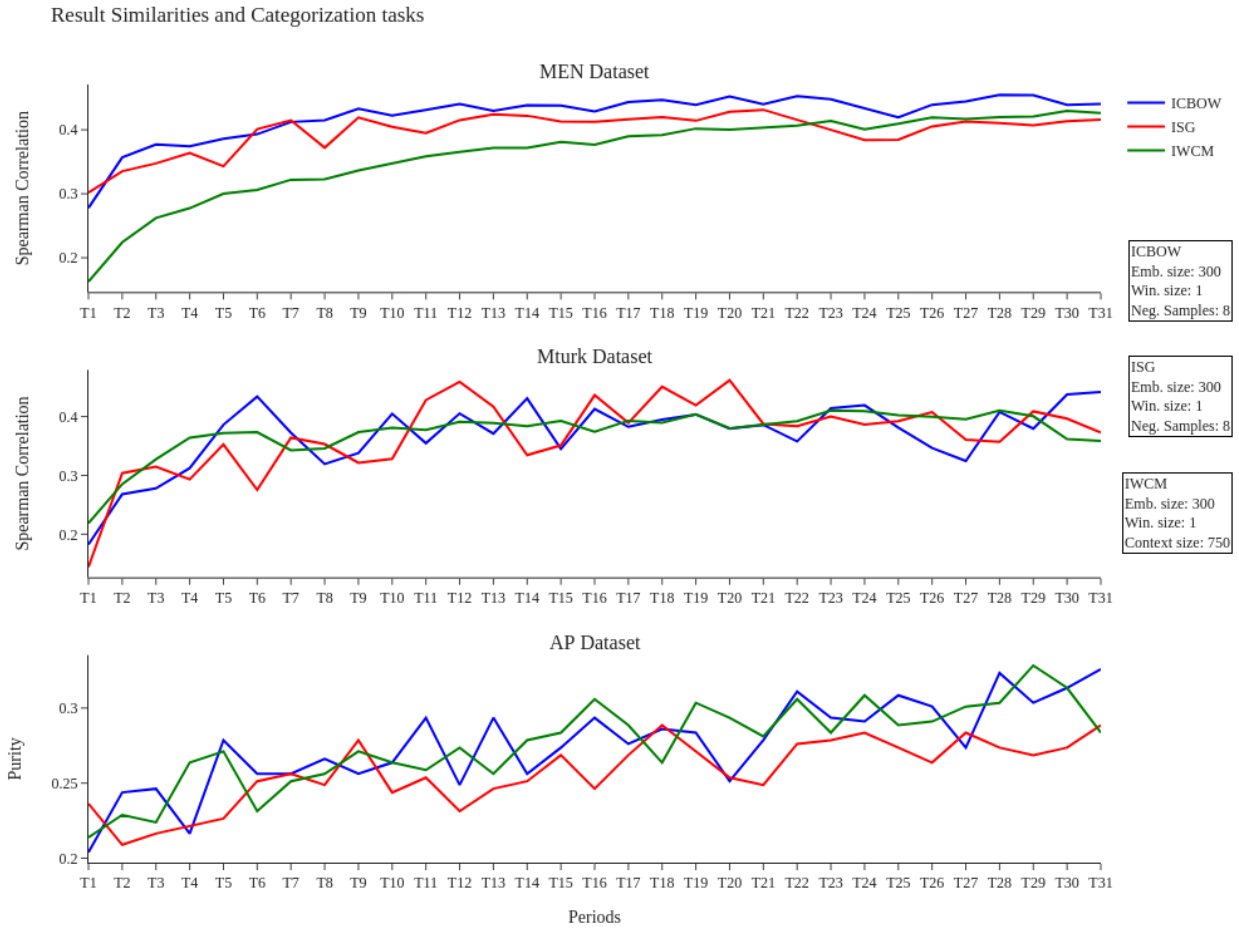

Figure B.20: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 1$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

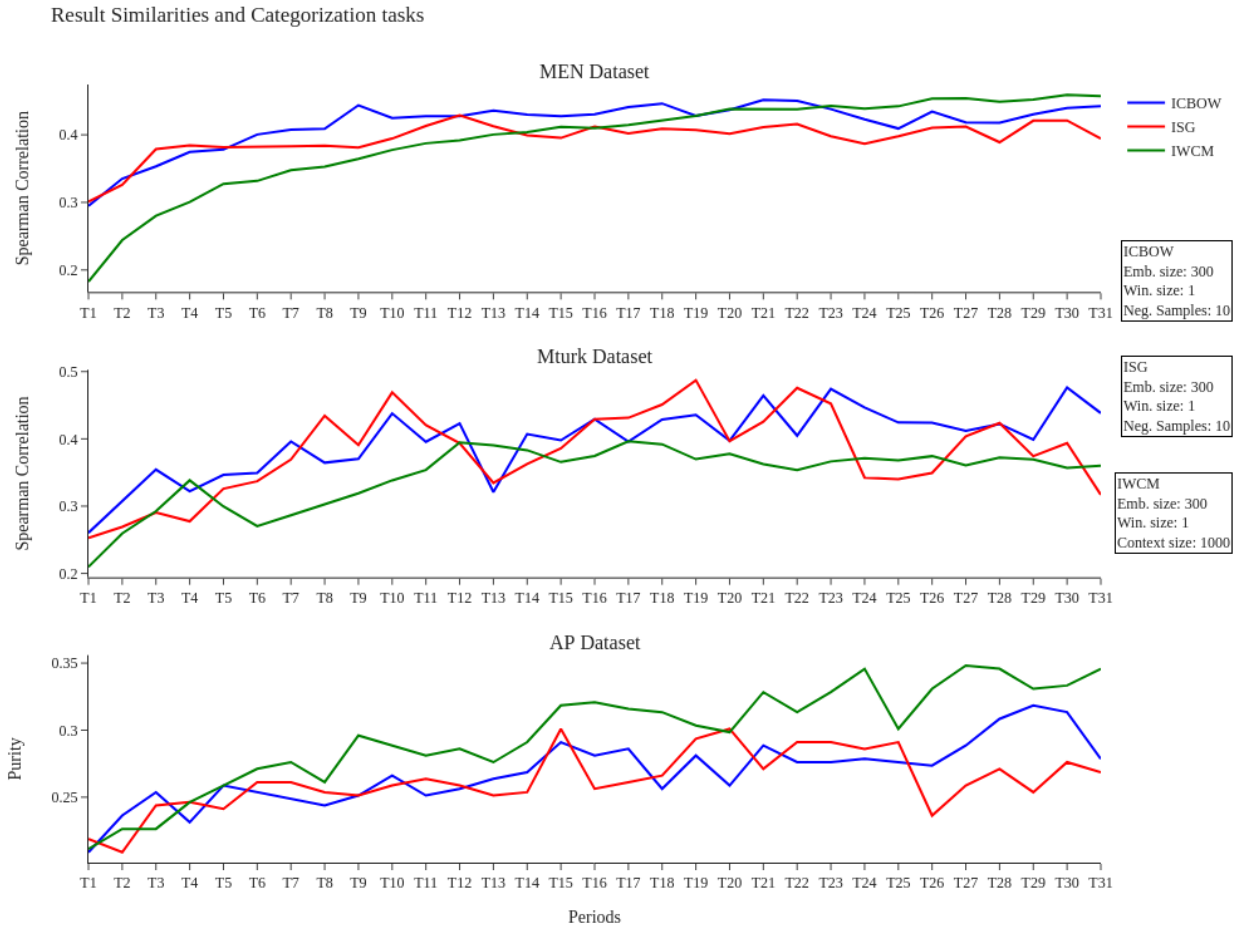

Figure B.21: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 1$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

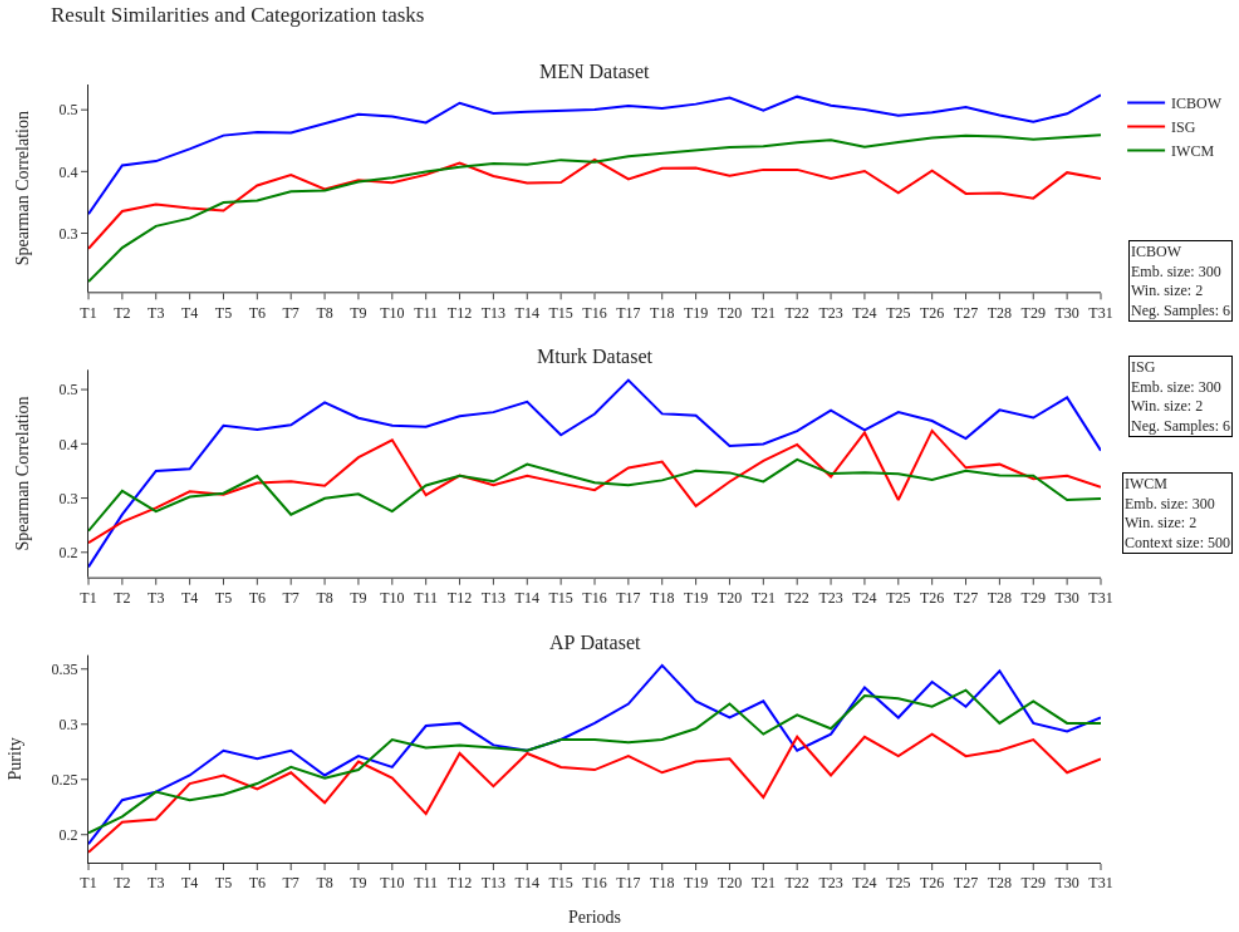

Figure B.22: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 2$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

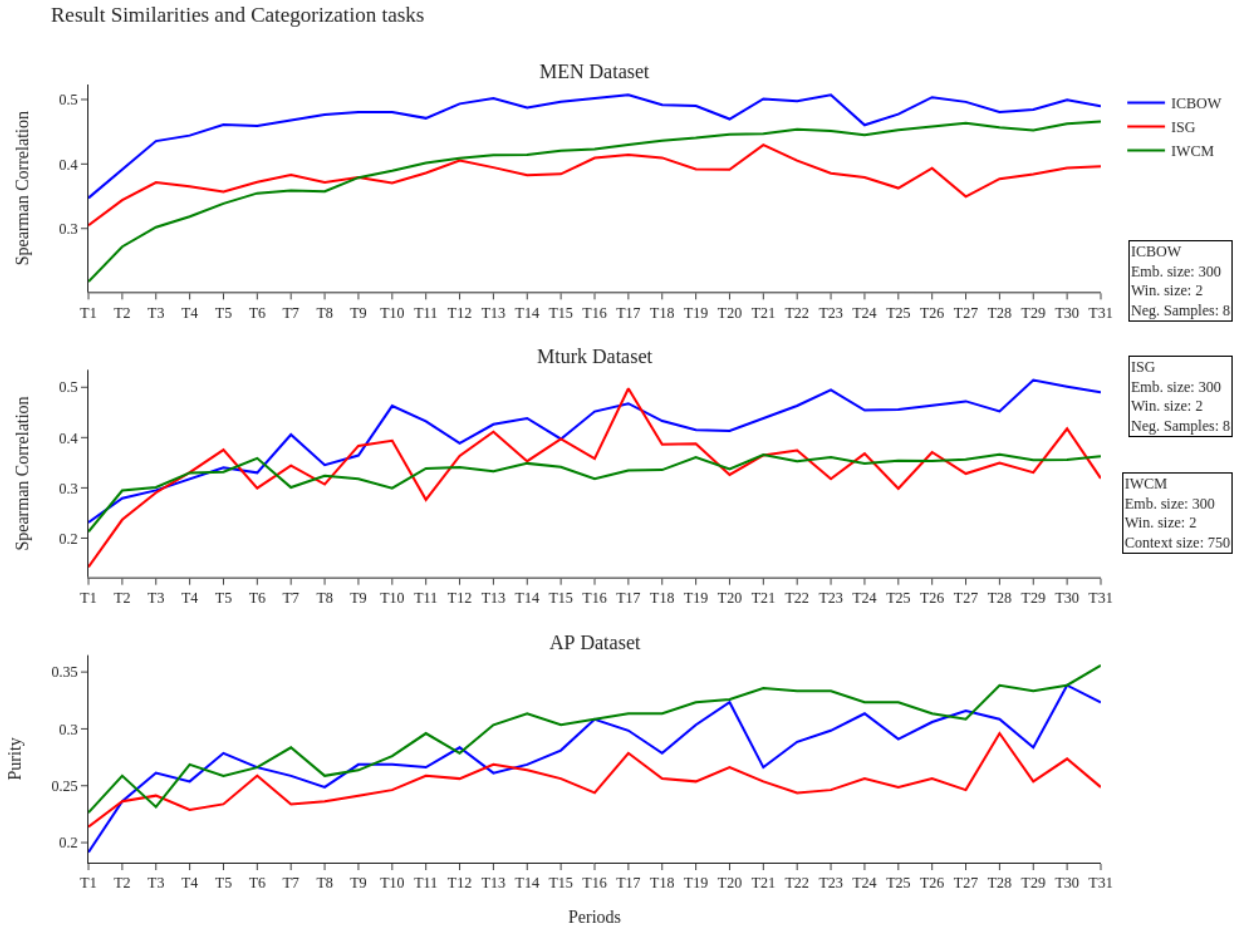

Figure B.23: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 2$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

Figure B.24: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 2$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

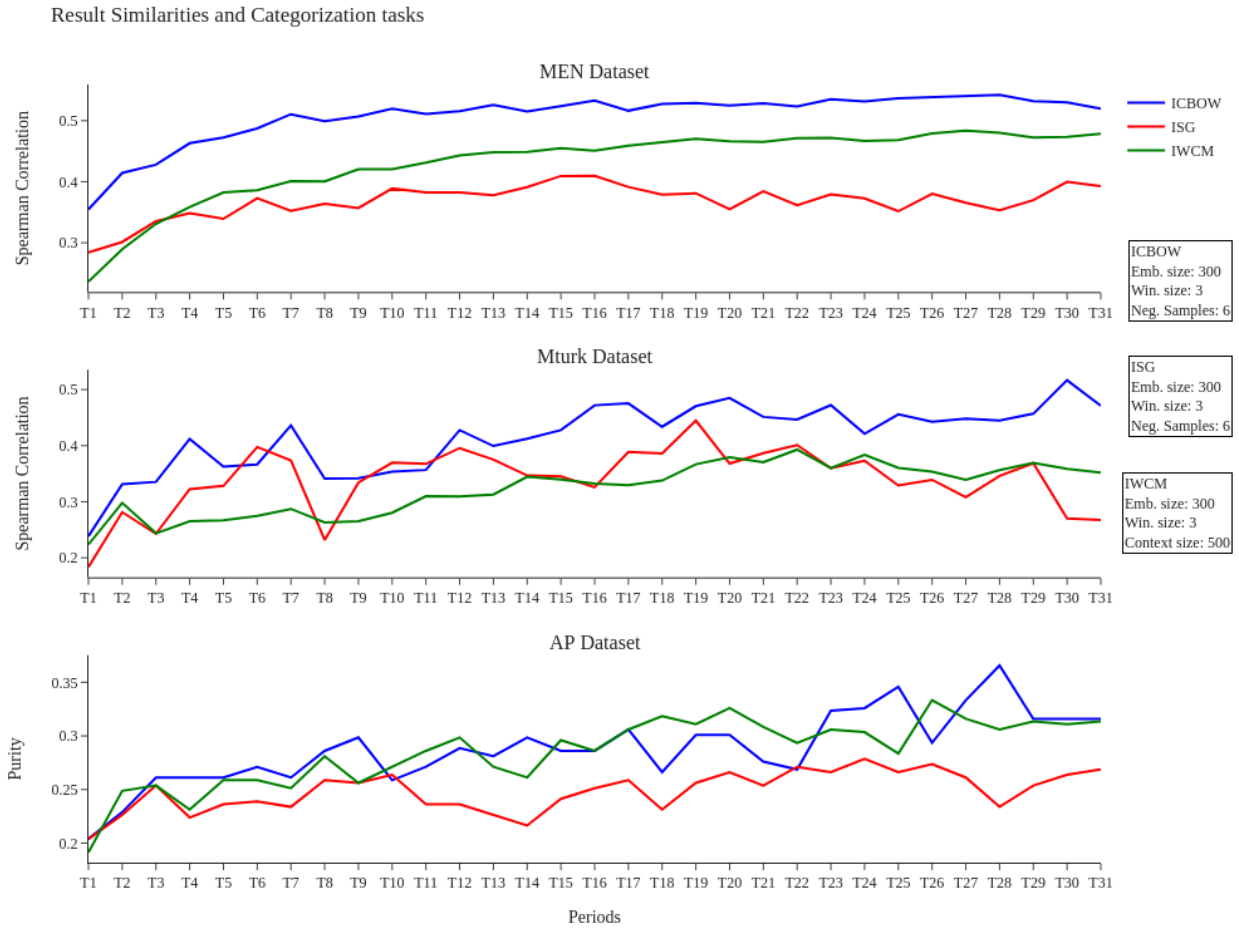

Figure B.25: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 3$, $ns\_samples = 6$, and $context\_size = 500$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

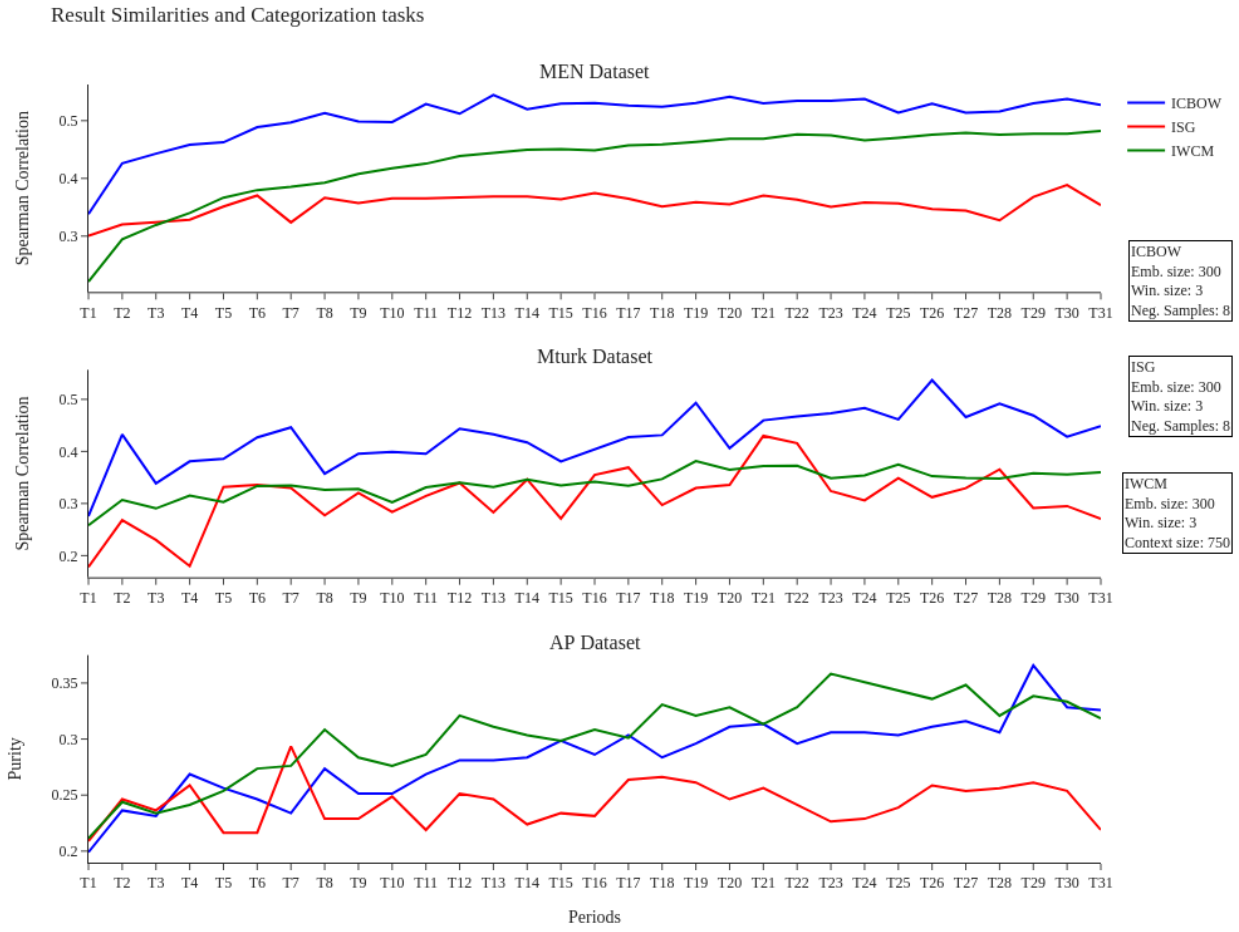

Figure B.26: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 3$, $ns\_samples = 8$, and $context\_size = 750$ across the training phase. The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.

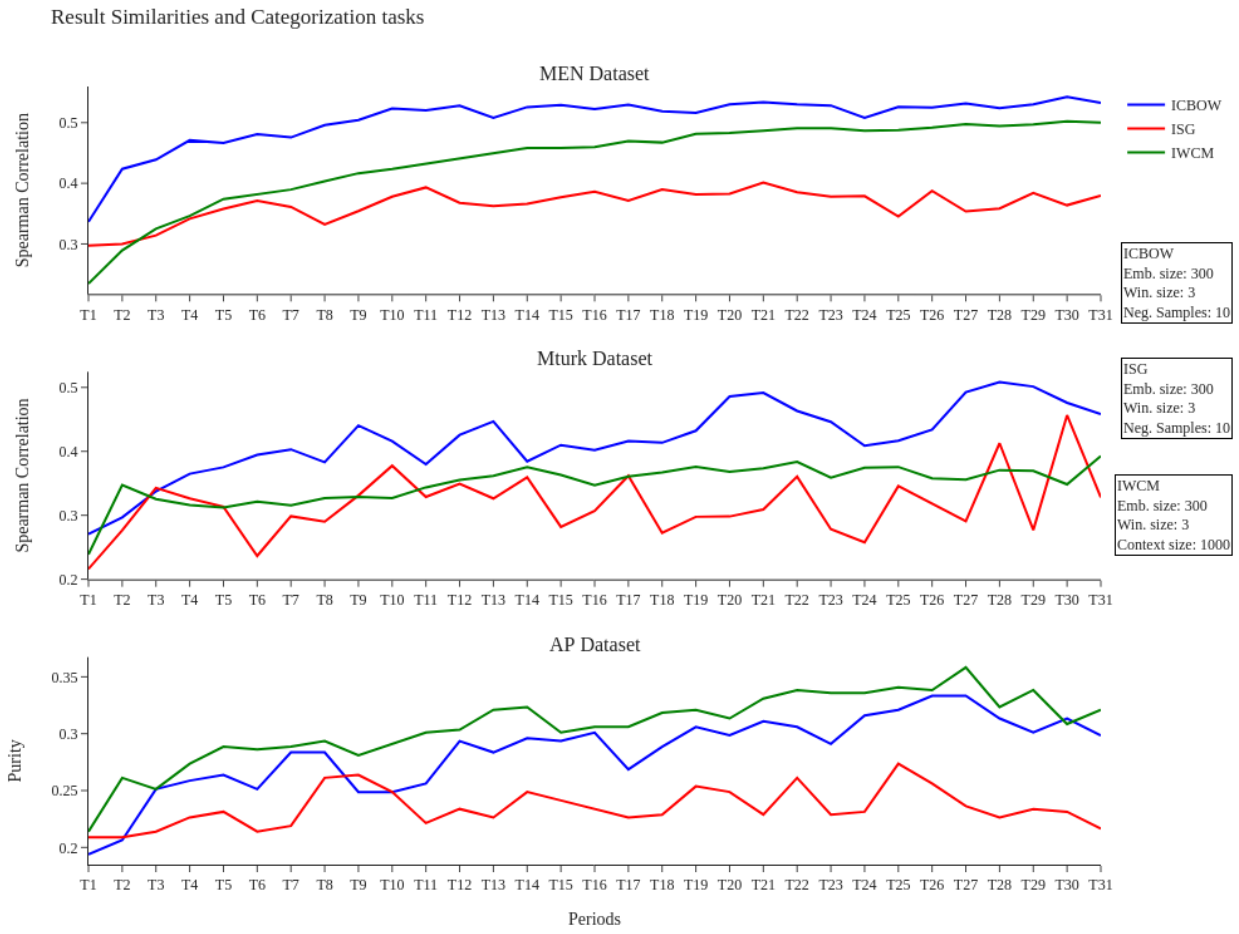

Figure B.27: In this experiment, we evaluated the performance of three incremental word embedding models, IWCM, ISG, and ICBOW, using the Periodic Evaluation technique for the similarity and categorization task using the hyperparameter settings of $emb\_size = 300$, $window\_size = 3$, $ns\_samples = 10$, and $context\_size = 1000$ across the training phase.The period p was set as 3,200,000 instances, which means the evaluator of the period evaluation was applied every 3,200,000 training instances.