Erasmus University Rotterdam

Erasmus School of Economics

Master Thesis Data Science and Marketing Analytics

Evaluating Word Embedding Methods for Sentiment Analysis

Kevin Kraayeveld (589908)

Supervisor:	Eran Raviv
Second assessor:	-
Date final version:	25th July 2024

Abstract

We evaluate the performance of various word embedding models for sentiment prediction. Specifically, we train Word2vec, GloVe, and FastText models on 3.6 million Amazon product reviews and assess their predictive performance compared to their pre-trained counterparts. We then compare these results with those of more advanced models: pre-trained BERT and ELMo models. Our evaluation test set consists of 50,000 labeled Amazon product reviews, the training set has 200,000 reviews. We use logistic- and ridge regression to classify to sentiment of the reviews. As a baseline, we also include the predictive performance of the VADER lexicon. For Word2vec, GloVe, and FastText, we test three methods of aggregating word embeddings into review embeddings: averaging, summing, and stacking. For ELMo and BERT, we utilize their built-in sentence embedding methods. Our findings indicate that the average aggregation method yields the best predictive performance. The Word2vec CBOW model trained on the Amazon reviews dataset achieved the highest accuracy of 84.13%. The pre-trained Word2vec, GloVe, and FastText models achieved around 80% accuracy. BERT and ELMo models, scored 75.54% and 81.64% accuracy respectively. The VADER lexicon performed the worst, with an accuracy of 69%. These results suggest that training word embedding models on context-specific data is more important to get an accurate sentiment prediction within a specific domain than employing advanced models trained on generic data.

Contents

1. Introduction

The increasing prevalence of Large Language Models (LLM) such as OpenAI's ChatGPT has made natural language processing (NLP) an undeniable aspect of our everyday lives. Central to NLP is the use of word embeddings, a technique that converts text into vectors. Word embeddings allow computers to mathematically manipulate and interpret words while keeping their semantic meaning. An example of the use of word embeddings is that you can measure the distances between two words. Similar words have a small distance, whereas opposite phrases have a large distance.

The aforementioned ChatGPT uses Transformer-based embeddings developed by OpenAI themselves (Brown et al., 2020), but many other word embedding methods exist. Notable examples include: Word2Vec (Mikolov et al., 2013), a widely used model, as well as Global Vectors for Word Representation (GloVe) (Pennington et al., 2014). More recently developed models which are also widely used are ELMo (Peters et al., 2018), BERT (Devlin et al., 2018), and Facebook's FastText (Bojanowski et al., 2017). However, the word embedding landscape is continually developing, with even newer models coming to the surface, such as RoBERTa, an optimized version of BERT (Liu et al., 2019). The continuing development and improvement of GPT models further state that word embedding methods are still in development and are continually improving.

Apart from the many methods to generate word embeddings, numerous applications exist, particularly in the field of marketing where NLP plays a significant role. Traditional applications of NLP in marketing include content and topic extraction, sentiment analysis, and writing style extraction (Hartmann and Netzer, 2023). Topic and content extraction can be used for brand name monitoring, by extracting brand information from social media and other similar sources (Klostermann et al., 2018). Sentiment analysis and writing style extraction are similar in that they both try to look at the underlying characteristics or emotions of the author of a piece of text while they are writing it. Sentiment analysis solely focuses on how negative or positive a text is, while writing style extraction can have numerous goals. Writing style extraction is the process of analyzing the characteristics of a text by looking at the vocabulary choice of the author. An example of an application in marketing is to study the vocabulary a consumer uses when they convey ideas about a brand and then examine the consumer's behavior that goes along with that (Hovy et al., 2021). Sentiment analysis is one of the most widely employed NLP tasks in marketing (Hartmann and Netzer, 2023). Sentiment analysis can be defined as "the identification of the positive or negative orientation of textual language" (Hirschberg and Manning, 2015). A widely used application is to analyze the sentiment of user-generated content such as product reviews. Companies can then examine its effect on multiple aspects of the firm, such as the brand's stock performance on the market (Tirunillai and Tellis, 2012) or their sales (Tang et al., 2014). The sentiment of a text can be expressed as a continuous range, usually between -1 (negative) and 1 (positive). It can also be expressed as a binary variable with 0 (negative) and 1 (positive).

More recent word embedding and training methods have allowed for newer applications such as summarizing text and generative question answering (Hartmann and Netzer, 2023). Both summarizing text and question-answering require natural text generation (NLG). NLG aims to produce a logical and grammatically correct text for a defined input type (Reiter and Dale, 1997). For text summarization, the input would generally be a large text. The goal would be to generate a smaller text with the same narrative that includes all crucial points of the original text. That could be useful in marketing by allowing advertisers to summarize product reviews into very informative summaries (Hartmann and Netzer, 2023). Chatbots are a practical marketing application for generative question answering. By having a well-trained chatbot on a sales website, a consumer's questions can be answered quickly, thereby boosting sales (Hildebrand and Bergner, 2019).

This thesis' objective is to assess the accuracy of the most prevalent word embedding models through a simple prediction task in a marketing context. The quality of the word embedding model is highly influential on the predictive ability of the downstream model. That is because the values generated by the word embedding model will be used as the predictor variables. A word embedding model of higher quality will generate values that capture the meaning of a word better than a model of lesser quality. With word embedding values that capture the meaning of the words well, downstream prediction models are able to predict instances more efficiently. We will use the performance metrics of the prediction task as an indication of the accuracy of the word embedding model, by keeping all things equal in the prediction process except for the word embedding model. We will then look at the differences in prediction metrics and computational load to evaluate each word embedding method relative to the other methods. After looking at the metrics and training time of the models, we can conclude the advantages and disadvantages of each word embedding model for this specific downstream marketing task.

In the context of downstream marketing tasks, opting for a straightforward prediction task is logical, as it allows for easy inference of the word embedding model's accuracy correlation. Since our aim is accuracy measurement, employing a supervised machine learning task with clearly defined labels is necessary. Among the discussed NLP applications for marketing, sentiment analysis aligns best with these criteria. That is primarily because sentiment analysis is the most quantifiable among the mentioned applications. The simplest form of sentiment analysis is to classify texts as positive or negative. Therefore, we will perform a binary prediction task on the sentiment of product reviews. We chose a binary prediction task instead of regression because it is easier to label. Additionally, we can use classification's performance metrics to give us insight into the predictive power beyond just accuracy. We can use precision and recall to evaluate differences in positive and negative predictions, respectively. These two metrics are interesting because they could indicate whether a word embedding model predicts positive or negative instances more accurately. These metrics allow us to create a nuanced conclusion about the accuracy, computational load, and the word embedding model's "temperament" so to say. Its temperament could be optimistic, pessimistic, or neutral, depending on the potential differences in precision and recall. A more scientific way to name this temperament metric would be the bias toward negative or positive predictions. The objectives of this research can be boiled down into the following research question:

> *How do different word embedding models perform in accuracy, bias, and computational load for sentiment analysis?*

Using word embeddings for sentiment analysis is a relatively recent development. Before the advent of word embeddings, lexicon-based methods were the standard approach for sentiment analysis (Taboada et al., 2011). Lexicons are essentially dictionaries with predefined sentiment values for words. Over time, these lexicon-based approaches have also seen significant improvements.

Given these advancements, it is interesting to compare the predictive performance of word embedding models with state-of-the-art lexicons. This comparison aims to determine which method yields better accuracy in sentiment prediction and whether the high computational complexity of word embeddings is justified. That leads us to our secondary research question:

> *How do word embeddings compare to lexicons in terms of accuracy for sentiment prediction, and is the high computational complexity of word embeddings justified?*

2. Theoretical Framework

2.1 Word Embeddings

Word embeddings are constructed through diverse methodologies, yet the foundational process of creating embeddings remains somewhat consistent across models. Each word embedding model needs extensive, diverse textual data to be trained. After the data gathering, the text needs to be preprocessed. Preprocessing is a crucial preliminary step, involving operations such as tokenization, lowercase conversion, and the removal of punctuation and common stop words (Webster and Kit, 1992). Tokenization facilitates the segmentation of text into manageable units. These units could range from individual words to multi-word phrases or even single characters. The goal of converting the text to lowercase and removing punctuation is to create a consistent string of words. Removing stop words (such as "a" and "the") increases the information density because stop words are not informational. Following preprocessing, feature extraction, and model training are executed, tailored to the specific characteristics of each word embedding model. After model training, the model's performance can be evaluated with any downstream task. Optionally, a model can be fine-tuned if the performance is not sufficient.

2.1.1 Word2vec

Google's Word2vec paper (Mikolov et al., 2013) proposed 2 new word embedding models called Continuous Bag-of-Words (CBOW) and Continuous Skip-gram. Both models use fairly simple approaches to minimize computational complexity (Mikolov et al., 2013).

Continuous Bag-of-Words

The Continuous Bag-of-Words model (CBOW) uses words from a fixed context window to predict the current word with a neural network. For a context window of size c, CBOW considers c words around the target word (before and after). The combination of those words is the context of the target word. CBOW collects many context-target pairs from the text corpus consisting of one target word with context words as specified earlier. To create meaningful word embeddings from the context-target pairs, CBOW uses a feed-forward neural network to train two matrices (Wang et al., 2019). These two matrices are initialized with random values before running the neural network for the first time (Wang et al., 2019).

The neural network consists of an input layer, a projection layer, and an output layer (Mikolov et al., 2013). The input layer's nodes are each context word as a one-hot encoded vector. The input layer gets mapped to the projection layer through a weight matrix that is the same for all words. This weight matrix is a two-dimensional matrix with dimensions $|V| \times D$ (Wang et al., 2019). Each one-hot encoded input vector gets multiplied by this matrix. After that, the intermediate embeddings of the context words get combined, typically through averaging or summation. This combined context vector gets applied by another weight matrix. This weight matrix is also a two-dimensional matrix, however, this matrix has dimensions $D \times |V|$. After applying the second weight matrix, CBOW applies a softmax operation to create a probability vector of size $|V|$ as the output layer. Each word's probability of being the target word is represented in this vector. Because we know the actual target word, we can calculate the loss through back-propagation with stochastic gradient descent. The weight matrices get updated to minimize the cross-entropy loss. This process gets repeated for a specified number of iterations. After all iterations are completed, the word embeddings are trained.

After minimizing the loss of the target word prediction, the word embeddings of the context words are in the first weight matrix. The word embeddings of the target words are in the second matrix. Because all words in V are used as context and target words, two embeddings exist for each word. Both word embeddings can be used individually or averaged to get the final vector embedding for a word.

Continuous Skip-gram

Skip-gram (Mikolov et al., 2013) essentially works the same way as CBOW. Only the order of the middle word and context words are reversed. The middle word gets used as the input, and the probabilities for context words get predicted to create word embeddings in a similar neural network as the CBOW method.

2.1.2 Global Vectors

Global Vectors (GloVe) (Pennington et al., 2014) uses a word-word co-occurrence matrix to train word embeddings. A word-word co-occurrence matrix captures the amount of times each word in the vocabulary co-occurs with each other word. For vocabulary V of size $|V|$, the matrix is of dimensions $|V| \times |V|$. Cell X_{ij} of this matrix represents how often word w_i co-occurs with word w_j. The window that defines co-occurrence is a given value that has to be determined before creating the co-occurrence matrix. From the word-word co-occurrence matrix the probability of co-occurrence ($P_{ij} = P(j|i)$) can be calculated, which is X_{ij}/X_i, i.e. the probability that w_j co-occurs in the text along with x_i. Now that we know the probabilities of words occurring in the same context, we can start to train the GloVe model. GloVe initializes random word vectors for each word in V, which then get trained to minimize GloVe's loss function, which is the following function:

$$J = \sum_{i,j=1}^{|V|} f(X_{ij})(\mathbf{w}_i^T \tilde{\mathbf{w}}_j + b_i + \tilde{b}_j - \log X_{ij})^2 \tag{2.1}$$

Where $f(X_{ij})$ is the weighting function of co-occurrences. w_i and $\tilde{w}_j$ are the word vectors for words i and j, respectively. b_i and $\tilde{b}_j$ are the bias terms for words i and j, respectively. X_{ij} is the co-occurrence count of words i and j. In other words, the loss function minimizes the difference between the dot product of word vectors of words i and j and the logarithm of the word's co-occurrence count. The model does this for each word combination in the vocabulary. The function is weighted by a weighting function and optimized using bias terms. The bias terms allow Glove to capture additional information about the words beyond their vector representations. After running a set number of iterations of this training process, the loss function should converge, after which the embeddings have been to the best of the model's ability.

2.1.3 FastText

FastText (Bojanowski et al., 2017) is an extension of the Word2vec model. The model is enriched by considering the subwords of a word w. In the model, w gets represented as a bag of character n-gram. To take the example as given in the original paper: if we were to take $n = 3$ for the n-gram, the word "where" could be represented as $< wh, whe, her, ere, re >$ and $< where >$ (Bojanowski et al., 2017). While FastText and Word2Vec share similarities in architecture, FastText diverges by considering words as bags of character n-grams rather than treating them as indivisible units. However, like Word2Vec, FastText encompasses skip-gram and CBOW implementations and follows the same training procedure. The incorporation of sub-words in FastText aims to enhance out-of-vocabulary accuracy by enabling the generation of embeddings based on sub-word components present in the vocabulary. This feature enhances the model's capability to handle previously unseen words effectively.

2.1.4 ELMo

One of the newer generations of word embeddings is ELMo (Peters et al., 2018). ELMo is a contextualized embedding model, meaning that the vector representation of a word varies based on the word's context. To achieve this, ELMo uses entire sentences as input to create word embeddings for each word in the sentence. That allows the vector representation of a word like "bank" to differ between sentences such as "he deposits money at the bank" and "he swims across the river to reach the bank".

ELMo employs a two-layer bidirectional language model (biLM), which is a long short-term memory (LSTM) model to predict each word using both preceding and following words in the sentence. Additionally, ELMo utilizes character convolutions to enhance the model's ability to capture character-level syntactic features of words. Character convolutions generate dense vector representations for each character, enabling the model to handle out-of-vocabulary words and misspellings better than models that rely solely on complete word embeddings, which may not cover all possible variations of a word.

To create word embeddings, ELMo utilizes the intermediate results (hidden states) of the biLM. For each token, $2L + 1$ representations are computed by the L-layer biLM. ELMo then combines these representations into a single vector representation by selecting the top layer or using task-specific weighting on the representations.

This architecture allows ELMo to produce embeddings that capture the nuanced meanings of words in context, leading to improved performance on multiple natural language processing tasks.

2.1.5 BERT

BERT is a multi-layer bidirectional Transformer encoder (Devlin et al., 2018). That means the model consists of transformer blocks that can communicate back and forth. A transformer consists of an encoder and a decoder, with N identical layers (Vaswani et al., 2017). The encoder transforms every input to the same format with the same vector size for the algorithm to process every input identically. Each layer in the encoder has two sub-layers: a multi-head attention mechanism and a position-wise fully connected Feed-Forward network, also called a multi-layer perceptron (Devlin et al., 2018).

Attention

To understand what a multi-head attention mechanism is, we first need to understand the concept of single-head attention, also called self-attention. Attention functions are used to transfer the meaning from one word to another and thereby add context to the meaning of a word. This creates richer word embeddings compared to embeddings without attention functions because, without it, a word can only have one embedding and therefore only one meaning. In the original transformer paper (Vaswani et al., 2017), the formula for single-head attention is given as this:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{2.2}$$

where Q is the query matrix, K is the key matrix, V the value matrix and d_k is the dimensionality of the word embeddings. As the d_k variable suggests, you start with training word embeddings without an attention function beforehand. The attention function will only modify existing word embeddings based on their context, not create new ones. It uses the Q, K, and V matrices to become context-aware. You can think of the query matrix as words "querying" other words to give it context. For example, you can have a noun such as "pen" querying other words to acquire knowledge on whether it has a certain color. If the word "red" appears just before "pen", intuitively humans know the word red is about the pen, which means that the pen is red. A computer can understand this relationship through the attention mechanism. It is convenient to conceptualize the key matrix as the answer to the query. In reality, the queries and keys are usually not as straightforward as this example, however it is a convenient example to understand the concept.

For each word, the original word embedding gets multiplied by the query matrix to create a smaller query vector. The change in dimensionality is caused by the size of Q. Q has d_k columns and i rows, where $i << d_k$, so when multiplying the two you get a much smaller vector because the amount of rows of Q is low compared to d_k. The same thing is done for K. K is of equal size to Q and gets multiplied by the original embeddings to create key vectors. For all words, the dot product of the query and key vectors gets taken (the QK^T part of the formula). This creates a grid of dot products between key and query vector pairs. We call this the attention pattern. In the attention pattern, large values are query and key pairs that match, so the query and key vectors are similar. Similar vectors have a high dot product. In the case of a large dot product, you would say the key attends to the query. In the example given earlier, that means "red" attends to "pen". When the query and key vectors do not match, the dot product is small or negative. To make the values of the dot products less extreme, divide each element in the attention pattern by $\sqrt{d_k}$. But after that, the values in the attention pattern are still somewhat trivial. Because we want to conceptualize these values as probabilities, a softmax function gets run over it. This function ensures all values are between 0 and 1, and add up to one. So for each query, the keys add up to one. Large dot products get a high probability, while small and negative dot products get pushed to 0.

While training, it is important to ensure that words cannot give context to words that come before them. That means the value in the attention pattern must be 0 when the position of the key in the text comes after the query. This is crucial because while training, not just the last word gets predicted, but each word gets predicted to increase training data volume. For example in the sentence "He went to the store" not just the word store gets predicted but also the word that comes after "He", "He went", "He went to" and so on. You do this by setting all values that meet this condition to minus infinity before running the softmax function. That way they are always 0 afterwards and the values still add up to 1. This process is called masking.

Now finally to change the original embedding of a word to the embedding with added context, we multiply the probabilities in the attention pattern with the value matrix V. By multiplying the probabilities in the attention pattern by the value matrix, new vectors are created. So this the part where we multiply softmax($\frac{QK^T}{\sqrt{d_k}}$) by V. These vectors now have the updated word embeddings for the context. So now the word embedding of "pen" also has information about the fact that it is red. V is not of equal size to Q and K. It even consists of two separate matrices to save on the number of parameters needed. The size of the first matrix V_1 is usually i rows and d_k columns. The second matrix V_2 has d_k rows and i columns. That way the inputs and outputs are both of size d_k while using much fewer parameters than when using a single square matrix for V.

Multi-head attention uses multiple attention heads in parallel. In addition to that, the V_2 matrix of each head is concatenated to create one bigger matrix, called the output matrix. All distinct value vectors created by each attention head get summed up to make the final value vector, which is then added to the original word embedding to change the meaning of that word according to the context.

The parameters for Q, K, and V are all trained using back-propagation and gradient descent-based algorithms during the training phase.

Position-wise Feed-Forward Networks

After each layer of attention heads, the Transformer model includes a Position-wise Feed-Forward Network (FFN) layer. The primary role of the FFN is to enhance the model's learning capacity by transforming vector representations and introducing non-linearity. This transformation of vector representations, acquired through self-attention, refines and enriches the features encoded in the tokens. The introduction of non-linearity is crucial as it enables the model to capture intricate patterns in the data that linear transformations alone cannot represent effectively.

When processing a sentence, each token undergoes independent processing through the FFN. This ensures that each token's representation is transformed individually without considering information from other tokens at this stage. These token representations, each of dimensionality 512, are initially acquired through prior layers in the Transformer model (Vaswani et al., 2017).

The FFN applies the following sequence of operations to each token's vector representation individually:

$$\mathbf{x}_1 = \mathbf{x}W_1 + \mathbf{b}_1 \tag{2.3}$$

$$\mathbf{x}_2 = \max(0, \mathbf{x}_1) \tag{2.4}$$

$$\text{output} = \mathbf{x}_2 W_2 + \mathbf{b}_2 \tag{2.5}$$

Here, W_1 and W_2 are weight matrices, and $\mathbf{b}_1$ and $\mathbf{b}_2$ are bias vectors. The ReLU activation function is applied element-wise and is the step that introduces the non-linearity in the data. The weights and biases are different for each layer in the model. These three operations can be boiled down into one formula (Vaswani et al., 2017):

$$\text{FFN}(\mathbf{x}) = \max(0, \mathbf{x}W_1 + \mathbf{b}_1)W_2 + \mathbf{b}_2 \tag{2.6}$$

This results in a transformed vector which still has size 512.

2.1.6 Embedding methods overview

Embedding model	**Year**	**Published by**	**Method**	**Key features**	**Type of embedding**
Word2Vec	2013	Google	Sliding context window	Continuous Bag of Words (CBOW) and Skip-gram	Word embedding
GloVe	2014	Stanford University	Word co-occurrence matrix	Global Vectors	Word embedding
FastText	2016	Facebook	Sliding context window	Subword information, handling of out-of-vocabulary (OOV) words	Word embedding
ELMo	2018	Allen Institute for AI	Bidirectional Language model	Contextualized word embeddings using deep bidirectional LSTMs	Contextual embedding
BERT	2018	Google	Transformers	Bidirectional encoder representations from transformers by using attention	Contextual embedding

Table 2.1: Overview of Word Embedding Models

2.2 Lexicons

Another approach to performing sentiment analysis, distinct from using word embeddings and a machine learning prediction model, is a lexicon-based method. Lexicons are specialized dictionaries that focus on the meaning and usage of words within specific contexts or domains. In sentiment analysis, lexicons define the semantic orientation of words in terms of positive or negative sentiment, and often quantify the strength of that sentiment (Taboada et al., 2011). This method works by using a structured database of words and their attributes, enabling sentiment analysis based on predefined sentiment scores.

There are many different lexicons available, and significant development has occurred in this field over the past decades. Currently, the state-of-the-art lexicon for analyzing short web texts is VADER (Valence Aware Dictionary and sEntiment Reasoner) (Hutto and Gilbert, 2014; Bonta et al., 2019).

The VADER lexicon was created through four steps. First, a list of 9,000 lexical feature candidates was rated by 10 humans, using the wisdom of the crowd approach to acquire sentiment ratings between -4 and 4 for each feature (Hutto and Gilbert, 2014). Second, two experts rated the sentiment of 400 negative and 400 positive tweets between -4 and 4 (Hutto and Gilbert, 2014). These texts and ratings were analyzed by an algorithm to identify properties and characteristics of the text that affect sentiment intensity (Hutto and Gilbert, 2014). This analysis resulted in five heuristics: punctuation, capitalization, degree modifiers, contrastive conjunctions, and negation. The third step involved controlling for grammatical and syntactical features by having 30 people rate 6 to 10 different variations of 30 baseline tweets with slight grammar or syntax changes (Hutto and Gilbert, 2014). The fourth and final step was rating 4,000 tweets, 10,605 sentence-level movie reviews, 3,708 sentence-level snippets of technical product reviews, and 5,190 sentence-level snippets from 500 New York opinion editorials (Hutto and Gilbert, 2014). These steps resulted in a comprehensive list of sentiments for words, phrases, and idioms, which can be used to classify the sentiment scores of new texts.

3. Methodology

To answer our research question, we will perform sentiment analysis with the five popular word embedding models explained in section 2.1. We will use a large generic dataset with pre-defined labels to quantify the performance of each word embedding model for sentiment analysis. We implement the models on our chosen dataset in R and Python with the help of packages. The code is available on GitHub [1].

3.1 Data

For our data, we use a dataset from Kaggle [2] constructed by (Zhang et al., 2015). This dataset consists of 4 million English Amazon Reviews about all sorts of products offered on Amazon. 3.6 million of which are training data and 400,000 thousand are test data. All reviews are labeled as either negative or positive. Both classes have exactly 2 million observations so it has a perfect 50/50 split to prevent bias in the model. The labels were constructed by labeling reviews with a 1 or 2-star rating as negative and reviews with a 4 or 5-star rating as positive. The data contains 3 columns: the sentiment, the review title, and the review content. All these 3 columns have no missing values, which makes the dataset complete. We will not be using the title of the review in our analysis. The unprocessed training reviews consist of 267 million tokens, of which 4.67 million are unique (including punctuation, capitalization and numbers). We read the data into our R environment as a data table using the data.table package (Dowle and Srinivasan, 2023).

3.2 Data Preprocessing

We employ three distinct approaches for data preprocessing to accommodate different model requirements.

The first approach is the "complete cleaning" preprocessing approach. In this approach, we remove all punctuation, numbers, and leading/trailing whitespaces from the review text all with the text2vec package (Selivanov et al., 2023). After that, we remove accents from characters with the stringi (Gagolewski, 2022) package and convert all text to lowercase to standardize the text format. We tokenize the cleaned review text into individual words by splitting between words. Each token is therefore one word. Next, common stop words such as 'the', 'is', and 'and' are removed to filter out commonly occurring but less informative words. The removal of stop words tokens is done by the quanteda package (Benoit et al., 2018), which uses optimized C++ code to speed up the process. We used the stop word vocabulary of the tidytext package (Silge and Robinson, 2016). However, negation words are kept in the reviews, although they are stop words. These words are particularly important for sentiment analysis because they reverse the sentiment of a text. The words we did not remove, but that are in the stop words vocabulary are:

[1]https://github.com/KevinKraayeveld/WordEmbedding

[2]https://www.kaggle.com/datasets/kritanjalijain/amazon-reviews, accessed February 5th, 2024.

”not”, ”no”, ”never”, ”don’t”, ”shouldn’t”, ”isn’t”, ”aren’t”, ”hadn’t”, ”haven’t”. Stemming is then applied to reduce words to their root form, aiding in simplifying the vocabulary and improving text analysis accuracy. To stem the words we used the SnowballC package (Bouchet-Valat, 2023).

After tokenization and cleaning, we proceed to create a vocabulary of unique terms from the tokenized text with the help of the text2vec package (Selivanov et al., 2023). The vocabulary consists of 1.8 million words and 97.6 million tokens at this point. The vocabulary undergoes pruning to eliminate less frequent terms, specifically those appearing fewer than five times, reducing the feature space’s dimensions and improving computational efficiency. Post-pruning, the vocabulary comprises 207 thousand words. Given the dataset’s size, it’s reasonable to consider words occurring less than five times as misspellings or other uninformative words. Consequently, these infrequent words are removed from the reviews. The training dataset now consists of 97.6 million tokens, which will be used for training the word embedding models.

The no-stemming cleaning procedure does all the same steps as the complete cleaning procedure, but without word stemming. We leave this out because some pre-trained models are trained on unstemmed data and will therefore perform better if we do not stem the data. The vocabulary of this cleaning method is 258,000 words and consists of 97.3 million tokens.

The minimal cleaning method only removes excess whitespace, turns the text to lowercase, and removes unwanted punctuation. For punctuation, we only keep single dots, question marks, commas, and exclamation marks. This results in full lowercase sentences with sentence structure still in place. This pre-processing approach has a vocabulary size of 2.97 million and has 266.3 million tokens in total.

An overview of the three pre-processing approaches can be found in table 3.1.

	Complete cleaning	**No stemming**	**Minimal cleaning**
Remove excess whitespace	Yes	Yes	Yes
Remove punctuation	Yes	Yes	Yes, except for single commas, periods, question marks, and exclamation marks
Turn to lowercase	Yes	Yes	Yes
Remove numbers	Yes	Yes	No
Remove uncommon words	Yes	Yes	No
Remove stop words	Yes, except negation words	Yes, except negation words	No
Remove accents	Yes	Yes	No
Word stemming	Yes	No	No
Amount of tokens	97.6 million	97.3 million	266.3 million
Vocabulary size	207,437	258,165	2,972,305
Average amount of words per review	27.1	27	74
Longest review	207	207	254

Table 3.1: Text Cleaning Procedures Overview

3.3 Training Embeddings

While training each model, we utilized only the training dataset comprising 3.6 million reviews, ensuring that embeddings are not trained on the test data to maintain fair and unbiased results. Consequently, the test dataset contains out-of-vocabulary (OOV) words. The approach to handling these OOV words varies depending on the word embedding model. To ensure a fair comparison with the pre-trained Word2vec, GloVe, and FastText models, which utilize 300 dimensions, we will evaluate the custom-trained models with 300 dimensions. Additionally, we will also evaluate the custom-trained models with 50 dimensions to assess the necessity of utilizing 300 dimensions.

3.3.1 Word2vec

For the Word2vec model, we implemented both CBOW and Skip-gram. Both models were implemented using the Word2vec function of the Word2vec package (Wijffels and Watanabe, 2023). The model has a few hyperparameters, apart from the dimensionality of the word vectors: the context window as described in section 2.1.1, the number of learning iterations, and a learning rate. We used a context window of 5, 200 learning iterations, and a learning rate of 0.05. The Word2vec model is unable to handle OOV words, therefore we removed them from the test dataset before creating the review vectors.

3.3.2 GloVe

To train the GloVe model, we begin by constructing a word co-occurrence matrix from the training data. This matrix is generated using the text2vec package (Selivanov et al., 2023) in R, using a context window size of 5. After creating the co-occurrence matrix, the GloVe model is trained using the rsparse package (Selivanov, 2022) in R. During training, we set 100 as the maximum number of co-occurrences considered in the weighting function. We chose a learning rate of 0.05, along with default values of alpha (0.75) and lambda (0) as our hyperparameters. The training process is conducted over 200 iterations to ensure model convergence. For the GloVe model, we also removed OOV words from the test dataset.

3.3.3 FastText

We implemented the FastText model using the FastText (Facebook, 2016) and FastTextR (Schwendinger and Hvitfeldt, 2023) packages in R. The Continuous Bag of Words (CBOW) method was employed with a learning rate of 0.05, matching the learning rates used for GloVe and Word2vec models. This model is saved as a binary file, enabling the retrieval of word embeddings for both in-vocabulary and out-of-vocabulary words.

3.4 Pre-trained Embeddings

3.4.1 Word2vec

The pre-trained Word2vec model [3] is trained on a dataset of 100 billion words sourced from Google News articles. This model encompasses 3 million words and phrases, each represented in a 300-dimensional vector space.

In our implementation, we utilized R and Python in tandem, facilitated by the reticulate package (Ushey et al., 2024). Using the gensim package in Python (Rehurek and Sojka, 2011), we imported the model and extracted embeddings for words in both our training and test datasets. Notably, out-of-vocabulary (OOV) words were removed from both datasets before embedding, as Word2vec does not support handling such words.

3.4.2 GloVe

For the pre-trained GloVe model, we use a model published by Standford University [4]. It is trained on 42 billion tokens. Its vocabulary size is 1.9 million. The data was gathered by Common Crawl [5]. This word embedding model also has word vectors of length 300.

We read the model as a data table from a CSV file using (Dowle and Srinivasan, 2023). We then remove all the OOV words from the train and test datasets and only keep the words in the model that are in either the training or test dataset to save on memory space.

3.4.3 FastText

We utilized the pre-trained FastText model provided by Facebook AI Research, accessible via their FastText website [6]. Specifically, we employed the crawl-300d-2M-subword model (Mikolov et al., 2018), trained on the extensive Common Crawl dataset, consisting of 600 billion tokens. This model has a vocabulary of 3 million words, incorporating sub-word information, and operates with vectors of size 300. Our implementation made use of the FastText package in R (Schwendinger and Hvitfeldt, 2023).

3.4.4 ELMo

For ELMo we used a model published by Google, trained on the 1 billion word benchmark [7]. The 1 billion word benchmark is a dataset designed to evaluate different language models with each other. It consists of roughly 1 billion words gathered by the 2011 WMT News Crawl, which consists of news articles from around the world. The pooler embeddings in this model have a dimensionality of 1024.

First, we download the model using the Kagglehub package in Python. We then embed the

[3]https://github.com/mmihaltz/Word2vec-GoogleNews-vectors, accessed May 29th, 2024.
[4]https://nlp.stanford.edu/projects/glove/, accessed May 29th, 2024.
[5]https://commoncrawl.org/, accessed June 5th, 2024.
[6]https://fasttext.cc/docs/en/english-vectors.html, accessed June 26th, 2024
[7]https://www.kaggle.com/models/google/elmo/tensorFlow1/elmo/3, accessed May 29th, 2024.

reviews in Python using the tensorflow package (Abadi et al., 2015). We embed the reviews in batches of a few hundred reviews at a time to avoid memory errors.

3.4.5 BERT

For the pre-trained BERT model, we use the BERT base uncased model published by Google and available on Hugging Face [8]. This model is trained on the BooksCorpus and English Wikipedia, encompassing 3.3 billion words. It has 12 layers, 768 hidden units, and 12 attention heads, with 110 million parameters.

We implemented the model in R, but using Python code through the reticulate package (Ushey et al., 2024). We import the BERT-base-uncased model with the tensorflow Python package (Abadi et al., 2015). Through the transformers Python package (Wolf et al., 2020) we encode each review as one and take the pooler output as the review embedding. These embeddings have a vector size of 768.

3.5 Review Vectorization

After obtaining individual word embeddings, the next step is to embed the review itself. It's important to note that this applies only to non-contextual embedding models, as contextual embedding models have built-in functions to embed entire texts into a single embedding. Singular word embedding models do not offer this functionality by definition.

3.5.1 Non-contextual models

We will explore three different methods to embed the review from individual word embeddings:

1. **Averaging** all the word embeddings, as performed by (Dilawar et al., 2018). For each word in the review, we have its corresponding word embedding. To obtain a single embedding for the entire review, we compute the average of these word embeddings. Specifically, if a review consists of n words and each word has a d-dimensional embedding, then the average embedding $\mathbf{v}_{\text{avg}}$ for the review is computed as:

$$\mathbf{v}_{\text{avg}} = \frac{1}{n}\sum_{i=1}^{n}\mathbf{v}_i$$

   where $\mathbf{v}_i = [v_{i,1}, v_{i,2}, \ldots, v_{i,d}]$ is the embedding of the i-th word in the review, with $v_{i,j}$ being the j-th dimension of the embedding.

2. **Summing** the word embeddings. Another approach is to sum the word embeddings dimension-wise instead of averaging them. This method also preserves the original dimensionality of the word embeddings for the review embeddings. Unlike averaging, summing can produce more extreme values across each dimension of the sentence embedding. However, this could introduce a bias in the prediction model. Reviews with more words might generate more extreme values, potentially being interpreted as more positive by the model,

[8] https://huggingface.co/google-bert/bert-base-uncased, accessed May 29th, 2024.

even if the review's sentiment is not more positive, but merely contains more words. Despite this potential bias, we are interested in the prediction power when using this method. The sum embedding $\mathbf{v}_{\text{sum}}$ for the review is computed as:

$$\mathbf{v}_{\text{sum}} = \sum_{i=1}^{n} \mathbf{v}_i$$

where $\mathbf{v}_i = [v_{i,1}, v_{i,2}, \ldots, v_{i,d}]$ is the embedding of the i-th word in the review.

3. **Stacking** the word embeddings. In this method, we concatenate all the word embeddings together to form a long vector. This approach is straightforward but results in a very long vector, which can be computationally expensive. To handle varying review lengths, we pad the concatenated vector with zeros to match the size of the longest review. For instance, if the longest review has m words, and each word has a d-dimensional embedding, then the stacked embedding $\mathbf{v}_{\text{stacked}}$ is:

$$\mathbf{v}_{\text{stacked}} = [v_{1,1}, v_{1,2}, \ldots, v_{1,d}, v_{2,1}, v_{2,2}, \ldots, v_{2,d}, \ldots, v_{n,1}, v_{n,2}, \ldots, v_{n,d}, \underbrace{0, 0, \ldots, 0}_{(m-n)\times d \text{ zeros}}]$$

Where $v_{i,j}$ is the j-th dimension of the embedding of the i-th word in the review, and the $(m - n) \times d$ zeros are added to pad the vector to the length of the longest review.

In the averaging and summing methods, the information of the individual embeddings gets lost. This method is supposed to keep as much information as possible at the cost of having many predictor variables. For instance, if you choose a vector length of 50 for your word embeddings and your longest review is 100 words, you would end up with 5000 predictor variables.

We will try these three vectorization methods on a few different models with different dimensionalities to see which approach yields the best accuracy and whether it holds up across distinct models. After selecting the best approach, we will use it from then forward on all the non-contextualized models.

3.5.2 Contextual models

In the BERT model, the [CLS] tag is meant to represent sentence-level classification. We use the embedding vector of this token as the predictor variable of the review's sentiment. This embedding vector is of size 768.

The ELMo model offers a "pooler" output. The pooler output of ELMo word embeddings provides a distilled representation of a sentence by aggregating information from the contextualized word embeddings across different layers of the bidirectional language model. After ELMo processes text through multiple LSTM layers, capturing various levels of linguistic context, the pooler output combines these embeddings to form a single, comprehensive vector. This aggregated vector encapsulates the diverse semantic and syntactic information encoded in the individual word embeddings, offering a robust and contextually nuanced representation of the entire sentence. The vector has a size of 1024.

3.6 Sentiment Prediction

We use logistic regression for most of our models, opting for ridge regression only when handling embeddings from the BERT, ELMo, or stacked embeddings created by any embedding model. The choice to use ridge regression in those cases is driven by the fact that ELMo, BERT, and stacked review vectors have a very high number of predictor variables, which logistic regression cannot efficiently manage due to potential issues with multicollinearity and overfitting. However, because this is not a problem for most models, logistic regression is selected as the primary model. Its simplicity makes it not only straightforward to implement but also easy to interpret, allowing us to clearly understand the relationship between the predictor variables and the outcome variable. This transparency is crucial for our research as it enables us to draw meaningful conclusions and insights from the word embeddings.

The primary focus of our research is on the relative performance of different models rather than their absolute performance. Given this objective, a simple model like logistic regression is advantageous. It allows us to make fair comparisons without the added complexity and computational demands that more sophisticated models would entail. This increased complexity could obscure the interpretability of the results and detract from the primary aim of our study, which is to assess the relative performance of word embedding models.

To ensure that our models are both robust and generalizable, we use a substantial subset of our dataset for training and evaluation. Specifically, we use 200,000 reviews from the training dataset to fit the logistic regression model. Note that this is only for the sentiment prediction. We use 3.6 million reviews of the training dataset to train the models. For evaluation, we use 50,000 reviews from our test dataset.

3.7 Lexicon

We utilized the VADER lexicon through the VADER package in R (Roehrick, 2020). Since the VADER lexicon does not require training data (Hutto and Gilbert, 2014), we applied it to a subset of 50,000 reviews from our test data set. The VADER function generates sentiment scores in four categories: positive, neutral, negative, and compound. The positive, neutral, and negative scores each range from 0 to 1 and together sum up to 1. The compound score, which represents the overall sentiment of the text, is calculated from the positive, neutral, and negative scores and ranges from -1 to 1.

For our analysis, we use the compound score to predict the sentiment of each review. Reviews with a negative compound score are classified as negative, while those with a positive compound score are classified as positive. Reviews with a compound score of exactly 0 are randomly assigned to either the positive or negative category with equal probability.

3.8 Hardware

Two machines were used to generate the results. The first is a laptop with 16GB of RAM, an AMD Ryzen 5700U processor with 16 threads, a clock speed of 1.8GHz, and an embedded AMD Radeon graphics card. The second is a PC with 32GB of RAM, AMD Ryzen 5 5600X 6-Core Processor with 12 threads, and a clock speed of 4.28GHz. The PC has an AMD Radeon RX 5700 XT GPU. In the results, we will indicate computation time with a ”^{1}” next to model training done with the laptop and a ”^{2}” next to model training done with the PC.

4. Results

4.1 Review vectorization method

Model	**Dim**	**Accuracy**	**Precision**	**Recall**
GloVe	50	80.14%	78.98%	82.08%
GloVe	300	80.2%	79.31%	81.67%
Word2vec CBOW	50	82.07%	81.38%	83.12%
Word2vec CBOW	300	84.07%	83.33%	83%
Pre-trained FastText	300	80.52%	80.16%	81.39%
Average	-	81.4%	80.6%	82.25%

Table 4.1: Results of **summing** review vectorization method

Model	**Dim**	**Accuracy**	**Precision**	**Recall**
GloVe	50	80.22%	79.98%	80.56%
GloVe	300	83.27%	83.24%	83.27%
Word2vec CBOW	50	82.06%	82.05%	82.02%
Word2vec CBOW	300	84.13%	83.99%	84.29%
Pre-trained FastText	300	80.53%	80.97%	80.1%
Average	-	82.04%	82.05%	82.03%

Table 4.2: Results of **averaging** review vectorization method

Model	**Dim**	**Accuracy**	**Precision**	**Recall**
GloVe	50	64.58%	63.56%	68.44%
GloVe	300	Too computationally expensive		
Word2vec CBOW	50	57.72%	57.53%	58.98%
Word2vec CBOW	300	Too computationally expensive		
Pre-trained FastText	300	Too computationally expensive		
Average	-	61.15%	60.55%	63.71%

Table 4.3: Results of **stacking** review vectorization method

The results from the three tables above indicate that the averaging review (table 4.2) vectorization method achieves the highest accuracy across all the tested models. The sum method's accuracy, as can be seen in table 4.1, is only 0.64% worse than the averaging method on average. Using a z-test with a sample size of 50,000 we assess that this difference is statistically significant ($p < 0.01$), which allows us to conclude that the averaging method is significantly better. Apart from the superior accuracy, the averaging model also yields slightly more balanced results. This is evident as the precision and recall percentages diverge more from the accuracy percentage in the sum method compared to the average method. Specifically, the recall in the sum method is, on average, 1.6% higher than the precision, whereas in the average method, the recall is pretty much the same. This indicates that the sum method is slightly more inclined towards predicting positives when it should not. That results in a higher number of false positives.

In contrast, the stacking method (table 4.3) not only performs relatively poorly but also struggles with higher-dimensional word embeddings. This method demands a large amount of RAM to store all review vectors in working memory, often exceeding the capacity of a typical machine. While there are techniques to mitigate this memory issue, we deemed them unnecessary for this research, because the performance of stacked review vectors at lower dimensionality was significantly inferior to that of the summed and averaged methods. Additionally, there's no indication that increasing the dimensionality would yield a substantial improvement. The summed and averaged methods only saw a 2-3 percentage point increase in accuracy with higher dimensional models, so it's unlikely that the stacking method would achieve the roughly 20 percentage point improvement needed to match their performance.

Because the averaging vectorization method got the best accuracy, we will be using that approach to evaluate the rest of the non-contextualized models.

4.2 Main results

To create an uncluttered display of all the results, we divided them into four parts. The first part is the models which we trained on the Amazon reviews dataset. The second part is the pre-trained models, which were trained on other very large datasets by their creators. Next, we report the results of the contextualized models, and finally the results from the VADER lexicon. To see all results in one table, we refer the reader to Appendix A, in which table A.1 contains all results.

Model	**Dimensions**	**Preprocessing method**	**Accuracy**	**Precision**	**Recall**	**Time**
Word2vec CBOW	50	Complete cleaning	82.06%	82.05%	82.02%	2749 seconds[1]
Word2vec CBOW	300	Complete cleaning	84.13%	83.99%	84.29%	10186 seconds[1]
Word2vec skip-gram	50	Complete cleaning	82.01%	82.10%	81.81%	9576 seconds[1]
Word2vec skip-gram	300	Complete cleaning	84.04%	83.84%	83.79%	36792 seconds[1]
GloVe	50	Complete cleaning	80.22%	79.98%	80.56%	2398 seconds[1]
GloVe	300	Complete cleaning	83.27%	83.24%	83.27%	9991 seconds[1]
FastText	50	Complete cleaning	80.93%	80.59%	81.44%	483 seconds[2]
Average	-	-	**82.72%**	**82.57%**	**82.93%**	-

Table 4.4: Trained Models

The performance of the word embedding models that we trained on the Amazon reviews dataset can be seen in table 4.4. For these models, logistic regression was used to predict sentiment, and the complete cleaning approach was used to pre-process the reviews.

The Word2vec CBOW model with 300 dimensions showed the highest accuracy of 84.13%, precision of 83.99%, and recall of 84.29%, with a processing time of 10186 seconds, which is about 2 hours and 50 minutes. The skip-gram variant of Word2vec also performed well, particularly at 300 dimensions, with an accuracy of 84.04%, but required the longest processing time with 36792 seconds (which is 10 hours and 13 minutes). From a z-test, it follows that the difference of the accuracy between the two Word2vec models is not statistically significant ($p \approx 0.7$). Which means, based on accuracy we cannot definitively say one performed better than the other. However, the time required to train the Skip-gram model was about 3.5 times more than that of the CBOW model. Therefore, when also taking training time into account, the Word2vec CBOW model was much better.

GloVe models displayed moderate performance, with the 300-dimensional version achieving 83.27% accuracy. GloVe's 83.27% accuracy is statiscally different to Word2vec CBOW's 84.13% accuracy with $p < 0.001$. FastText, while having the shortest processing time of 483 seconds (8 minutes), had slightly lower performance metrics.

Overall, the average performance across models we trained was 82.72% for accuracy, 82.57% for precision, and 82.93% for recall, highlighting a trade-off between dimensionality, performance, and processing time.

Model	**Dimensions**	**Preprocessing method**	**Accuracy**	**Precision**	**Recall**
Pre-trained FastText	300	Complete cleaning	77.96%	77.59%	78.58%
Pre-trained FastText	300	No stemming	79.96%	80.03%	80.07%
Pre-trained Word2vec	300	Complete cleaning	76.88%	76.76%	77.03%
Pre-trained Word2vec	300	No stemming	79.64%	79.63 %	79.42%
Pre-trained GloVe	300	Complete cleaning	76.9%	76.01%	78.56%
Pre-trained GloVe	300	No stemming	80.54%	80.64%	80.33%
Average	-	-	**78.23%**	**78.11%**	**78.82%**

Table 4.5: Pre-trained Models

The performance comparison of FastText, GloVe, and Word2vec models with 300 dimensions, focusing on results without stemming, is summarized in table 4.5. Logistic regression was utilized for sentiment prediction across all models. The time column was left out of this table because pre-trained models do not require training- or other processing time.

The pre-trained GloVe model showed the best accuracy performance of 80.54%, alongside a precision of 80.64% and recall of 80.33% without stemming. FastText achieved an accuracy of 79.96%, precision of 80.03%, and recall of 80.07% without stemming. Word2vec demonstrated the lowest accuracy of the three with an accuracy of 79.64%, precision of 79.63%, and recall of 79.42% without stemming. Complete cleaning achieved slightly lower accuracy scores. These results underscore the significant impact of pre-processing methods on model performance, highlighting the importance of choosing appropriate pre-processing techniques to optimize model outcomes.

The average results of the pre-trained models are; an accuracy of 78.23%, precision of 78.11%, and recall of 78.82%. That is about 4% to 4.5% lower than the average of the models trained on the Amazon reviews across the three performance metrics. All custom trained models are significantly better than their pre-trained counter-parts (all with $p < 0.001$).

Table 4.6 presents performance metrics for the two contextualized models we used. The predictions for these models were done using ridge regression. BERT, utilizing a 768-dimensional representation and using the pre-processing approach without stemming, achieves an accuracy of 75.54%, precision of 75.42%, and recall of 75.69%. Getting the sentence embeddings from the BERT model took 22,633 seconds for 250,000 reviews. ELMo, operating with a 1024-dimensional

Model	Dimensions	Preprocessing method	Accuracy	Precision	Recall	Time
BERT	768	No stemming	75.54%	75.42%	75.69%	23230 seconds[1]
ELMo	1024	No stemming	81.64%	82.36%	80.45%	8147 seconds[1]
Average	-	-	**78.59%**	**78.89%**	**78.07%**	-

Table 4.6: Contextualized Models

representation and also utilizing the no stemming pre-processing approach, demonstrates higher performance with an accuracy of 81.64%, precision of 82.36%, and recall of 80.45%. Acquiring the sentence embeddings from the ELMo model took significantly less time, 8,147 seconds to be exact. Note that these times are without training the model, however, because the models create contextualized embeddings for entire reviews, significant processing time is still required.

The average performance across these contextualized models is an accuracy of 78.59%, precision of 78.89%, and recall of 78.07%.

Model	Preprocessing method	Accuracy	Precision	Recall	Time
Lexicon	Complete cleaning	65.87%	63.48%	74.58%	4608 seconds[2]
Lexicon	No stemming	68.62%	65.45%	80.3%	4284 seconds[2]
Lexicon	Minimal cleaning	69.01%	63.1%	91.13%	9576 seconds[2]
Average	-	**67.17%**	**64.68%**	**81.67%**	-

Table 4.7: Lexicons

Table 4.7 summarizes performance metrics for lexicon-based models with different pre-processing methods. Each model's accuracy, precision, recall, and computational time are evaluated.

The VADER lexicon achieves an accuracy of 65.87%, precision of 63.48%, and recall of 74.58% when using the complete cleaning pre-processing approach. The time it took to get the sentiments from the lexicon was 4,608 seconds for this approach. While using the no stemming approach, the lexicon models improve to an accuracy of 68.62%, precision of 65.45%, and recall of 80.3%, requiring 4,284 seconds to get the sentiments. Minimal cleaning of lexicon models yields the highest performance, with an accuracy of 69.01%, precision of 63.1%, and recall of 91.13%, though at the cost of increased training time of 9,576 seconds. The difference in accuracy between the no stemming and minimal cleaning approach is not statistically significant however, since a z-test gives $p \approx 0.18$. The difference between the complete cleaning and other methods are very significantly significant. The p-value is pretty much 0. The relatively high training time of the minimal cleaning approach caused by the fact that the minimal cleaning approach resulted in having almost three times as many words compared to the complete cleaning and no stemming approaches. That is why it takes the minimal cleaning lexicon longer to get the sentiments.

5. Discussion

5.1 Review vectorization

When aggregating word embeddings into a review embedding, using an averaging or summing approach is generally more advantageous than stacking individual word embeddings. This is because a stacking approach increases the number of predictor variables significantly, making it challenging for prediction models to effectively handle the resulting high-dimensional input. By averaging or summing the word embeddings, the information from multiple dimensions (word embeddings) is condensed into a single vector. This aggregation reduces the complexity of the input data, transforming multiple dimensions into a unified representation that captures the essence of the entire review. Consequently, prediction models can more efficiently interpret and utilize this lower-dimensional representation to make predictions, thereby enhancing the model's performance and interpretability. Thus, employing averaging or summing techniques simplifies the input space and facilitates more effective utilization of the aggregated embeddings in predictive tasks.

While the summing and averaging vectorization methods yielded similar accuracy, they exhibited differences in precision and recall. The averaging approach demonstrated balanced precision and recall, whereas the summing approach showed higher recall than precision, indicating a tendency to predict more positive instances. This bias could stem from the summing method potentially generating more extreme values in the resulting review embeddings. These non-normalized extremes might mislead the logistic regression model into favoring positive predictions over negative ones, even if the actual distribution is evenly split between positive and negative cases. In contrast, the averaging vectorization method avoids this issue by aggregating word embeddings without creating extreme values. Due to the averaging process, the range of values within each dimension remains consistent between the original word embeddings and the resulting review embeddings.

5.2 Word embedding models accuracy comparison

The most remarkable finding from our results is that the Word2vec and GloVe models that we trained on the Amazon reviews dataset perform significantly better than BERT, slightly better than ELMo, and quite a lot better than the pre-trained Word2vec, GloVe, and FastText models. The best performing trained model was Word2vec CBOW with an accuracy of 84.13%, while BERT had an accuracy of 75.42%, ELMo had 81.64%, and the pre-trained models with no stemming pre-processing had 80.05% accuracy on average. These results signify the importance of training word embedding models in a relevant context as the data that you want to predict the sentiment in. Pre-trained embeddings, though often derived from vast and diverse datasets, may not adequately capture the intricacies of sentiment-related language in specialized domains such as product reviews. This limitation arises because pre-trained models are typically trained on general corpora with a broad scope, which may not encompass the domain-specific vocabulary, idiomatic expressions, or sentiment markers unique to a particular dataset. In contrast, embeddings trained directly on domain-specific data can more effectively capture these features from the text, thereby enhancing their performance in sentiment analysis tasks in that specific domain.

Looking at the results of the FastText models suggests that handling out-of-vocabulary (OOV) words does not enhance model performance in sentiment analysis. Although the pre-trained FastText model showed comparable results to pre-trained Word2vec and GloVe models, this was not the case for the FastText model trained specifically on the Amazon reviews. In that scenario, both Word2vec CBOW and skip-gram models outperformed the FastText model by approximately 2% in terms of accuracy at a dimensionality of 50. Given that FastText and Word2vec CBOW models share the same methodology, the lower accuracy of FastText suggests that its approach to OOV words may detrimentally affect predictive performance in sentiment analysis. This implies that FastText may not effectively embed sentiment information from OOV words, contributing to its relatively poorer performance in this context. FastText offers a significant advantage in its remarkably short training time. For instance, it requires only 8 minutes to train compared to Word2vec, which demands 46 minutes. The shortened training time makes FastText perhaps a viable option in situations where you want to continually train a model on new data because the model takes very little time to train.

5.3 Contextualized models

BERT represents the pinnacle of word embedding models among those assessed in our research. BERT having the lowest accuracy in this study is therefore unexpected and noteworthy. Given the recent prominence of transformer models, one would anticipate a higher accuracy from BERT. BERT's lower performance of 75.54% accuracy contrasts with ELMo, which achieved 81.64%. This discrepancy prompts a deeper exploration into why BERT might not have excelled in this sentiment analysis prediction task.

BERT's underperformance could stem from several factors. First, despite its sophisticated architecture and contextual understanding capabilities, BERT might not have been fine-tuned specifically for sentiment analysis on the dataset used in this study. Transformer models like BERT require substantial computational resources and extensive fine-tuning of domain-specific data to optimize performance. If BERT was not adequately fine-tuned for sentiment analysis or if the dataset characteristics did not align well with BERT's pre-training, its performance could have suffered.

Moreover, the tokenization strategy employed by BERT may not have been optimal for the sentiment analysis task at hand. BERT tokenizes text into subword units, which might affect how sentiment-related nuances are captured and processed compared to models like ELMo, which uses character-based word representations.

5.4 Lexicons

Lexicons generally exhibit lower performance in sentiment analysis compared to word embeddings. Most word embedding models achieve an accuracy of around 80%, while the lexicon with a minimal cleaning approach had an accuracy of 69%. That suggests that lexicons may not adequately capture the complexity and diversity of language, especially in context-specific or evolving domains like social media or product reviews. Lexicons are static and lack adaptability to new or evolving language patterns. They do not inherently learn from data or update their sentiment associations based on new information.

None of the models exhibit a significant bias towards positive or negative sentiment, as indicated by their similar precision scores compared to recall. However, the lexicon demonstrates a notable discrepancy between recall and precision. On average, this difference amounts to 17%. In the lexicon with the highest accuracy, which employs the minimal cleaning approach, this difference increases to 28%. This disparity suggests that the VADER lexicon tends to bias towards predicting positive sentiment, leading to a higher rate of false positives. This could be attributed to the dataset containing a higher proportion of words considered positive by the lexicon compared to negative words, thereby influencing the overall positive prediction. The substantial increase in recall with the minimal cleaning approach, in contrast to no stemming or complete cleaning, implies that stop words and/or infrequent words may frequently lead the lexicon to classify a review as positive, even when it might not be the case.

5.5 Recommendations

For the review vectorization we recommend using the averaging method when the word embedding model does not offer built-in sentence embedding techniques. The averaging model had significantly better accuracy than the stacking method. The summing method performed similarly, but was still the difference in accuracy was still statistically significant. Also, the averaging method and less bias than the summing method, which is favorable.

When doing sentiment alaysis, we recommend using recommend using a custom-trained word embedding model, since these models had the highest accuracy scores. The Word2vec model had the best accuracy score, in particular, therefore we recommend using that model. Because the CBOW method had required the lowest training time we recommend using the CBOW method.

Because the VADER lexicon had significantly lower accuracy than all word embedding models and also did not take much less time to get the sentiments, we do not recommend using it for sentiment analysis, rather we recommend always using a word embedding model. Preferably, the model is custom-trained, but only if you have enough data to train the model.

6. Limitations

A limitation of this study is that the word embedding models used did not have consistent embedding vector sizes. Each pre-trained model comes with fixed dimensionality, determined during its training phase, which cannot be altered without retraining the model from scratch. This inconsistency in vector sizes means that comparisons between models may be influenced by the differing dimensional capacities, potentially impacting their performance in sentiment analysis tasks. As a result, variations in model accuracy could partly stem from the inherent differences in their embedding dimensionality, complicating a direct comparison of their effectiveness.

Additionally, the difference in vector sizes necessitated the use of different regression models. For larger embeddings, we employed ridge regression to handle the increased dimensionality. Apart from the models with higher dimensionality (BERT and ELMo), we applied ridge regression to word2vec, GloVe, and fastText embeddings to see whether there was a significant difference. For all three models, the accuracy of ridge regression predictions for these models was within a 1% range of the accuracy achieved using logistic regression.

Another limitation of this study is that we were unable to gather results for the FastText model with a dimensionality of 300 using the same implementation as for the 50-dimensional model. This issue appears to be a limitation of the R package we used. Consequently, we could not directly compare the performance of FastText at different dimensionalities, which might have provided valuable insights into how dimensionality affects model performance. This gap in our data limits the comprehensiveness of our analysis and the robustness of our conclusions regarding the effectiveness of different embedding sizes.

Another limitation of this study is that the results are confined to short reviews. The findings may not apply to other data formats or longer reviews. Specifically, the longest review in our dataset, considering both the complete cleaning and no stemming pre-processing approaches, was 207 words. The reviews were 27 words on average. This limitation suggests that the performance and conclusions drawn from our models might not hold up when applied to longer or differently structured reviews or other sorts of texts, potentially affecting the generalizability of our results to broader text analysis tasks.

7. Conclusion

This study evaluated various word embedding models in terms of accuracy, precision, recall, and computation time to answer the main research question: *How do different word embedding models perform in accuracy, bias, and computational load for sentiment analysis?* To answer this question, we focus on the key findings of our results.

The best-performing model in terms of accuracy is the custom-trained Word2vec CBOW model. This model achieved an 84.13% accuracy, which is about 4% higher than the non-contextual pre-trained models, 3% higher than ELMo, and 8.5% higher than BERT, which are all statistically significant differences, assessed by z-tests. These results underscore the importance of training word embeddings within the context of the specific data domain to achieve high predictive accuracy. None of the models exhibited an inherent bias towards positive or negative predictions, as evidenced by the similar precision and recall values across all models. However, the review vectorization method did introduce slight bias in the prediction models. Specifically, the summing vectorization method had similar accuracy to the averaging method but demonstrated higher recall than precision, indicating a bias toward positive predictions.

In terms of computational load to train a model, fastText performed the best, requiring only 8 minutes to train a 50-dimensional model on 3.6 million reviews. In contrast, the worst performing model with 50 dimensions was Word2vec skip-gram, needing 2 hours and 40 minutes of training time for the same dataset. The Word2vec CBOW model, while sharing a similar methodology to the Word2vec skip-gram model, required only 46 minutes to train a 50-dimensional model. The contextualized models, although pre-trained, still demanded substantial computation time to create review embeddings, with ELMo taking 2 hours and 16 minutes and BERT taking 6 hours and 27 minutes. In conclusion, in terms of computational load, using a pre-trained model or training a FastText model is the least demanding. However, in terms of accuracy, the custom-trained Word2vec CBOW model stands out. The observed bias, influenced by vectorization methods and possibly prediction models, suggests that further research is needed to thoroughly investigate the bias of embedding models. This research could measure precision and recall across different prediction models and vectorization approaches.

Moving on to the second research question: *How do word embeddings compare to lexicons in terms of accuracy for sentiment prediction, and is the high computational complexity of word embeddings justified?* Embedding models outperform the state-of-the-art VADER lexicon in both computation time (for most models) and accuracy. This indicates that the high computational complexity of word embeddings is justified, especially since lexicons, to achieve better predictive power, have also become more computationally complex.

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Benoit, K., Watanabe, K., Wang, H., Nulty, P., Obeng, A., Müller, S., and Matsuo, A. (2018). quanteda: An r package for the quantitative analysis of textual data. *Journal of Open Source Software*, 3(30):774.

Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2017). Enriching word vectors with subword information. *Transactions of the association for computational linguistics*, 5:135–146.

Bonta, V., Kumaresh, N., and Janardhan, N. (2019). A comprehensive study on lexicon based approaches for sentiment analysis. *Asian Journal of Computer Science and Technology*, 8(S2):1–6.

Bouchet-Valat, M. (2023). *SnowballC: Snowball Stemmers Based on the C 'libstemmer' UTF-8 Library.* R package version 0.7.1.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. (2018). BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805.

Dilawar, N., Majeed, H., Beg, M. O., Ejaz, N., Muhammad, K., Mehmood, I., and Nam, Y. (2018). Understanding citizen issues through reviews: A step towards data informed planning in smart cities. *Applied Sciences*, 8(9):1589.

Dowle, M. and Srinivasan, A. (2023). *data.table: Extension of 'data.frame'.* R package version 1.14.8.

Facebook, I. (2016). *fastText: Library for fast text representation and classification.*

Gagolewski, M. (2022). stringi: Fast and portable character string processing in R. *Journal of Statistical Software*, 103(2):1–59.

Hartmann, J. and Netzer, O. (2023). Natural language processing in marketing. In *Artificial Intelligence in Marketing*, volume 20, pages 191–215. Emerald Publishing Limited.

Hildebrand, C. and Bergner, A. (2019). Ai-driven sales automation: Using chatbots to boost sales. *NIM Marketing Intelligence Review*, 11(2):36–41.

Hirschberg, J. and Manning, C. D. (2015). Advances in natural language processing. *Science*, 349(6245):261–266.

Hovy, D., Melumad, S., and Inman, J. J. (2021). Wordify: A tool for discovering and differentiating consumer vocabularies. *Journal of Consumer Research*, 48(3):394–414.

Hutto, C. and Gilbert, E. (2014). Vader: A parsimonious rule-based model for sentiment analysis of social media text. In *Proceedings of the international AAAI conference on web and social media*, volume 8, pages 216–225.

Klostermann, J., Plumeyer, A., Böger, D., and Decker, R. (2018). Extracting brand information from social networks: Integrating image, text, and social tagging data. *International Journal of Research in Marketing*, 35(4):538–556.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized BERT pretraining approach. *CoRR*, abs/1907.11692.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781.*

Mikolov, T., Grave, E., Bojanowski, P., Puhrsch, C., and Joulin, A. (2018). Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018).*

Pennington, J., Socher, R., and Manning, C. D. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., and Zettlemoyer, L. (2018). Deep contextualized word representations. *CoRR*, abs/1802.05365.

Rehurek, R. and Sojka, P. (2011). Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic*, 3(2).

Reiter, E. and Dale, R. (1997). Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87.

Roehrick, K. (2020). *vader: Valence Aware Dictionary and sEntiment Reasoner (VADER).* R package version 0.2.1.

Schwendinger, F. and Hvitfeldt, E. (2023). *fastTextR: An Interface to the 'fastText' Library.* R package version 2.1.0.

Selivanov, D. (2022). *rsparse: Statistical Learning on Sparse Matrices.* R package version 0.5.1.

Selivanov, D., Bickel, M., and Wang, Q. (2023). *text2vec: Modern Text Mining Framework for R*. R package version 0.6.4.

Silge, J. and Robinson, D. (2016). tidytext: Text mining and analysis using tidy data principles in r. *JOSS*, 1(3).

Taboada, M., Brooke, J., Tofiloski, M., Voll, K., and Stede, M. (2011). Lexicon-based methods for sentiment analysis. *Computational linguistics*, 37(2):267–307.

Tang, T., Fang, E., and Wang, F. (2014). Is neutral really neutral? the effects of neutral user-generated content on product sales. *Journal of Marketing*, 78(4):41–58.

Tirunillai, S. and Tellis, G. J. (2012). Does chatter really matter? dynamics of user-generated content and stock performance. *Marketing Science*, 31(2):198–215.

Ushey, K., Allaire, J., and Tang, Y. (2024). *reticulate: Interface to 'Python'*. R package version 1.36.1.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. *CoRR*, abs/1706.03762.

Wang, B., Wang, A., Chen, F., Wang, Y., and Kuo, C.-C. J. (2019). Evaluating word embedding models: Methods and experimental results. *APSIPA transactions on signal and information processing*, 8:e19.

Webster, J. J. and Kit, C. (1992). Tokenization as the initial phase in nlp. In *International Conference on Computational Linguistics*.

Wijffels, J. and Watanabe, K. (2023). *word2vec: Distributed Representations of Words*. R package version 0.4.0.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. (2020). Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Zhang, X., Zhao, J. J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. *CoRR*, abs/1509.01626.

A. Appendix

Model	**Dimensions**	**Preprocessing method**	**Accuracy**	**Precision**	**Recall**	**Time**
Word2vec CBOW	50	Complete cleaning	82.06%	82.05%	82.02%	2749 seconds[1]
Word2vec CBOW	300	Complete cleaning	**84.13**%	83.99%	84.29%	10186 seconds[1]
Word2vec skip-gram	50	Complete cleaning	82.01%	82.1%	81.81%	9576 seconds[1]
Word2vec skip-gram	300	Complete cleaning	84.04%	83.84%	83.79%	36792 seconds[1]
GloVe	50	Complete cleaning	80.22%	79.98%	80.56%	2398 seconds[1]
GloVe	300	Complete cleaning	83.27%	83.24%	83.27%	9991 seconds[1]
FastText	50	Complete cleaning	80.93%	80.59%	81.44%	483 seconds[2]
Pre-trained FastText	300	Complete cleaning	77.96%	77.59%	78.58%	-
Pre-trained FastText	300	No stemming	79.96%	80.03%	80.07%	-
Pre-trained Word2vec	300	Complete cleaning	76.88%	76.76%	77.03%	-
Pre-trained Word2vec	300	No stemming	79.64%	79.63 %	79.42%	-
Pre-trained GloVe	300	Complete cleaning	76.9%	76.01%	78.56%	-
Pre-trained GloVe	300	No stemming	80.54%	80.64%	80.33%	-
BERT	768	No stemming	75.54%	75.42%	75.69%	23230 seconds[1]
ELMo	1024	No stemming	81.64%	82.36%	80.45%	8147 seconds[1]
Lexicon	-	Complete cleaning	65.87%	63.48%	74.58%	4608 seconds[2]
Lexicon	-	No stemming	68.62%	65.45%	80.3%	4284 seconds[2]
Lexicon	-	Minimal cleaning	69.01%	63.1%	91.13%	9576 seconds[2]

Table A.1: Table of all results