BINUS UNIVERSITY

BINUS INTERNATIONAL

Final Project Cover Letter

(Group Work)

Student Information:	Surname:	Given Name:	Student ID Number:
1.	Kusnadi	Clarissa Audrey Fabiola	2602118490
2.	-	Jeffrey	2602118484
3.	Munthe	Priscilla Abigail	2602109883

Course Code : COMP6049001 **Course Name :** Algorithm Design and Analysis

Class : L3AC **Lecturer:** Dr. Maria Seraphina Astriani, S. Kom., M. T. I.

Type of Assignment : Final Project Report

Submission Pattern

Due Date : 8 January 2024 **Submission Date :** 8 January 2024

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offences which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to BiNus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

Clarissa Audrey Fabiola Kusnadi

Jeffrey

Priscilla Abigail Munthe

Table of Contents

Introduction/Background

Sentiment analysis has become a prominent area of study in recent years, particularly in the fields of computer science and linguistics. Sentiment refers to the subjective perception of an individual, which can be conveyed as positive, neutral, or negative. Sentiment analysis is a process that recognizes the words used and evaluates how language is employed to communicate human emotions (Guerreiro & Rita, 2020). It is a critical aspect in assessing viewpoints on subjects like politics, sports, finances, and product reviews. This will enable individuals and businesses to track the feedback regarding how people perceive them. Thus, aiding them in developing and enhancing marketing, branding, and product strategies.

The emergence of e-commerce platforms in recent years has significantly transformed business practices for companies. These platforms are widely used by businesses because they provide various online buying and selling options. They allow consumers to make purchases without the need to go to a physical store (Chatrath et al., 2022). The difference in information between the actual quality of products and the descriptions provided by sellers has led to a growing number of buyers relying on e-commerce reviews to gather information about various aspects, including pricing, service, logistics, and more. The variety of perspectives in the reviews makes it important to determine whether statements are positive or negative. This is crucial for customers when deciding what to purchase (Gandhimani, 2023). Hence, sentiment analysis serves as a method of content analysis for discerning the positive or negative perspectives of consumers.

Problems

The growing popularity of E-commerce websites has resulted in an abundance of distinct thoughts and viewpoints pertaining to different items and services. These platforms have emerged as a central space for consumers to articulate their opinions, encompassing a wide range of sentiments, including both favorable and unfavorable, thereby giving rise to an extensive and complex realm of content generated by users.

Consequently, there has been a notable surge in the quantity of data produced by users in the form of product reviews, ratings, comments, and debates, exhibiting an exponential growth pattern. The dataset under consideration is characterized by its extensive nature and intricate nature, which is further compounded by its dynamic nature as it continually evolves with the introduction of fresh perspectives. The management and extraction of relevant insights from this data provide a significant challenge.

In light of this challenge, it is crucial to utilize efficient techniques for data analysis. Sentiment analysis has emerged as a significant and practical methodology for comprehending the sentiments, attitudes, and emotions conveyed by users (Mm et al., 2020). Valuable insights into the general sentiment of customers can be gained through the analysis of this data, facilitating individuals and corporations to make informed decisions (Sanzgiri, 2023).

Sentiment analysis, alternatively referred to as opinion mining, utilizes methodologies from natural language processing and machine learning to categorize and measure the sentiment conveyed within textual data. Through the process of discerning favorable and unfavorable attitudes, businesses are empowered to assess the level of satisfaction among their customers, unveil possible concerns, and monitor patterns in consumer inclinations.

Sentiment analysis is of great importance in comprehending and addressing client feedback for both individuals and corporations. This phenomenon provides valuable insights into the cognitive processes and decision-making patterns of consumers, hence facilitating more effective and informed strategic planning. Sentiment analysis plays a crucial role in adjusting to the dynamic nature of the E-commerce industry and efficiently addressing user demands and concerns. It is utilized to enhance product offerings, improve customer service, and refine

marketing methods (Liu et al., 2020). Essentially, it serves as the intermediary that links the realm of electronic commerce with human sentiments and impulses that drive it.

Based on the problem analysis provided, we draw the conclusion to conduct a study analysis titled “Comparative Analysis of Text Processing Algorithms for Sentiment Analysis”.

Proposed Algorithms

Utilizing effective data analysis tools is of the utmost importance in the e-commerce sector in order to glean vital insights from the copious and ever-changing user-generated content. Opinion mining, which is another name for sentiment analysis, is a crucial technique that has gained considerable importance in understanding the varied sentiments, attitudes, and opinions expressed by consumers.

To address the problem of data management and extraction, we suggest conducting a comparative analysis of text processing algorithms. Our focus will be on widely recognized methods like TF-IDF and Word2Vec. TF-IDF is a traditional algorithm used for text processing, which allows us to identify and assign importance to important words in textual data (Addiga & Bagui, 2022). This feature is especially valuable for extracting characteristics from user-generated content in the E-commerce industry, such as product reviews and comments.

In contrast, Word2Vec is a relatively new and robust method that relies on neural networks. It captures the semantic relationships between words by converting them into continuous vector representations (Karani, 2022). This aids in comprehending sentiment as well as interpreting the contextual significance of words. Word2Vec possesses the benefit of capturing intricate nuances and can offer a more comprehensive sentiment analysis that takes into account the surrounding context. This dynamic approach is suitable for addressing the ever-changing user perspectives and language in E-commerce platforms.

By conducting a comparative analysis of these algorithms, we aim to identify which one is the most suitable to meet the specific needs of sentiment analysis in the e-commerce sector. This comparative methodology will enable us to evaluate factors such as precision, effectiveness, and flexibility in relation to the dynamic nature of the data. Furthermore, this will allow us to refine our methods for analyzing sentiment, which will provide businesses and individuals with valuable insight into customer attitudes and preferences.

Measurements

There are 5 standard metrics for comparing text processing algorithms, which are as follows:

- Creating Word Embeddings: Representing words as numerical vectors, capturing their semantic relationships and contextual meaning.

  Methods:

  TF-IDF: Calculates term frequencies (TF) and inverse document frequencies (IDF) to create numerical vectors based on word occurrences and rarity.

  Word2Vec: Uses neural networks to learn word embeddings from large text corpora, capturing semantic and contextual relationships.

  Measurement: Time taken to generate word embeddings for a given dataset.
- Training Time: The time required to train a sentiment analysis model using the generated word embeddings.

  Measurement: Time taken to train a model, measured in minutes and seconds.
- Memory Usage: The amount of memory required to store and process word embeddings and train the model.

  Measurement: Memory consumed during model training and prediction, measured in megabytes (MB).
- Virtual Memory: The combination of physical RAM and disk space used to extend available memory.

  Measurement: Virtual memory used during model training and prediction, measured in megabytes (MB).
- Classification Report: A performance evaluation of the sentiment analysis model using various metrics as follows:
  - Accuracy: The proportion of correctly classified sentiment labels.
  - Precision: The proportion of predicted positive sentiment labels that are positive.
  - Recall: The proportion of actual positive sentiment labels that are correctly predicted as positive.
  - F1 Score: The harmonic mean of precision and recall, balancing both metrics.

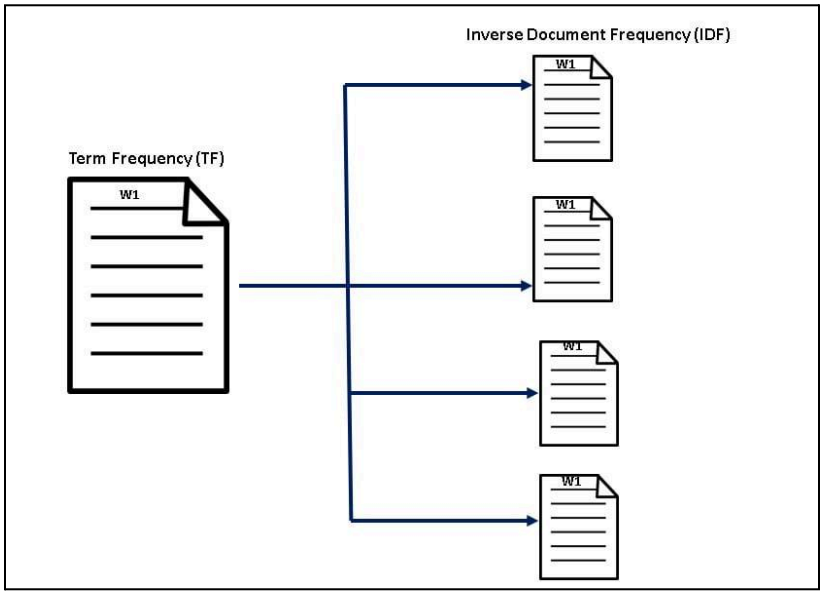
These 5 metrics were chosen to comprehensively assess text processing algorithms for sentiment analysis. They cover embedding creation time for efficiency, training time for development speed, memory usage for resource management, virtual memory for full resource picture, and a classification report for model performance, considering both accuracy and sensitivity with metrics like precision, recall, and F1-score. This holistic approach allows us to compare algorithms, understand trade-offs, and recommend the best one for specific project goals and resource constraints.

Concept of Algorithms

TF-IDF

The Term Frequency-Inverse Document Frequency (TF-IDF) is the product of two statistics, term frequency, and inverse document frequency, developed by two people, namely Hans Peter Luhn, recognized for his research on term frequency in 1957 and Karen Spärck Jones, who contributed to inverse document frequency in 1972 (Jeske, 2023). TF-IDF is a method of weighting a word or term in a text by considering both its frequency within the documents and its frequency across all documents (Rajaraman & Ullman, 2011).

The following illustration shows how the implementation of TF-IDF works (Wisdomml, 2022).



According to Cahyani and Patasik (2021), there are four steps to train the TF-IDF model as listed below:

1. Calculate the Term Frequency (TF)

   The first step is the calculation of the frequency of occurrence of each word in each document by using the formula shown below.

$$tf_t = 1 + \log(tf_t)$$

Where: tf_t is the number of occurrences of term t.

2. Calculate the Document Frequency (DF)

   The second step is the calculation of the number of documents containing a specific word.

3. Calculate the Inverse DF (IDF)

   The third step is the calculation of inverse document frequency of all the words in the sentences by using the formula shown below.

$$idf_t = log(\frac{D}{df_t})$$

Where: idf_t is the inverse document frequency, D is the number of documents, and df_t is the number of documents that contain the term t.

4. Calculate the product of Term Frequency and Inverse Document Frequency (TF-IDF)

   The last step is to take the product of the term frequency of each of the words with their inverse document frequency by using the formula shown below

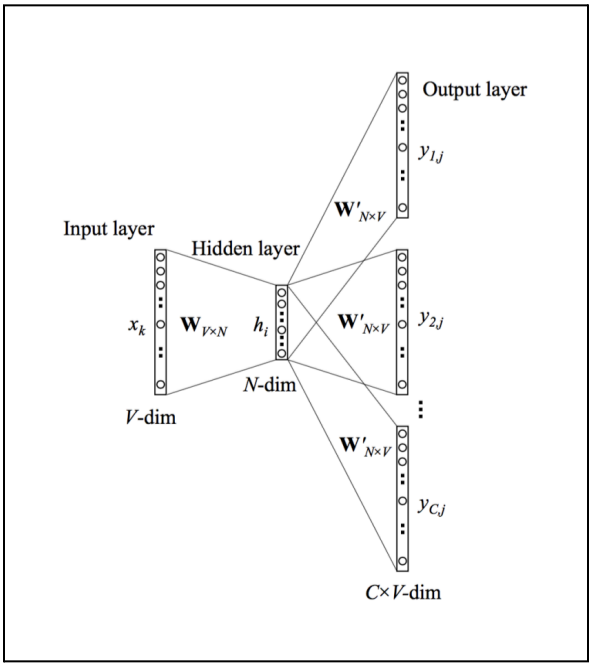
$$W_{t,d} = tf_t \times idf_t$$

Where: W is the weight of *term* (t) in document (d), tf_t is the number of occurrences of term t, and idf_t is the inverse document frequency that contains term t.

Word2Vec (Skip-Gram Model)

The Word2Vec model is a prediction-based algorithm developed by Tomas Mikolov and his colleagues at Google in 2013. The Word2Vec model uses a shallow neural network to embed high-quality word vectors (Mikolov et al., 2013). The underlying principle of the word2vec model is that words occurring in similar contexts are related (Rong, 2014). The Word2Vec algorithm comes in two flavors: the continuous bag-of-words (CBOW) model and the skip-gram (SG) model. In this project, the Skip-Gram model is employed. In the skip-gram model, the word vector is built by predicting the context or neighboring words of a given word. It enables the capture of semantic relationships and similarities between words based on each word's co-occurrence in text.

The following illustration shows the original model architecture/diagram presented in the paper by Mikolov et al., 2013.



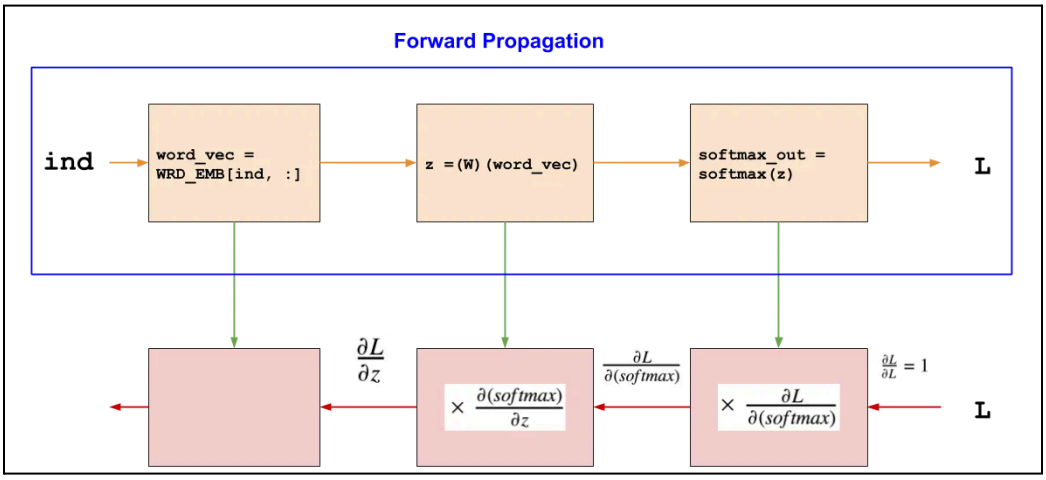
There are a few steps to train the word2vec model listed below:

1. Initialization of parameters to be trained

   Similar to most neural network models, the first step is to initialize the parameters which are the word embedding layer and the dense layer. There are some parameter initialization techniques used widely in deep learning, but in this project, we focused on random initialization. Random initialization enhances accuracy significantly by

initializing the weights very close to zero, but randomly (Ananthram, 2018). This ensures that each neuron performs different computations, which leads to better neural network training.

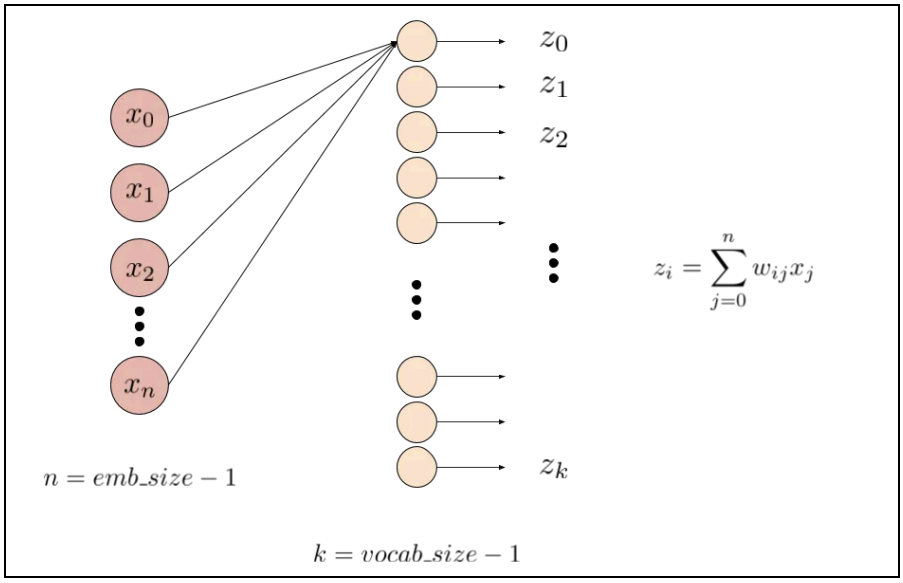
2. Forward pass (Forward propagation)



Forward propagation consists of three crucial steps: extracting the vector representation of the input word from word embeddings, transmitting the vector to the dense layer, and then applying the softmax function to the output of the dense layer (Chen, 2021).

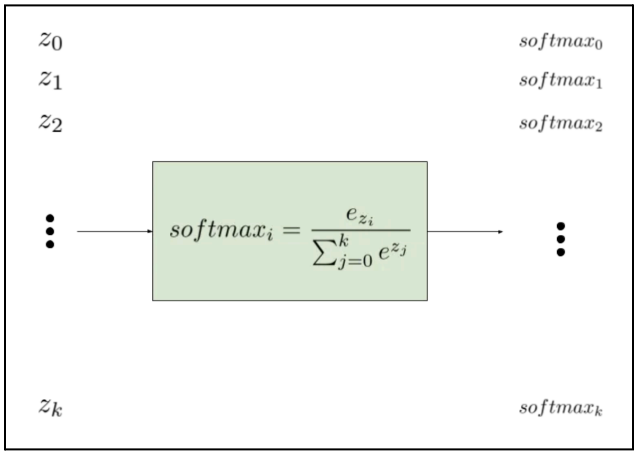
In many literary works, the input is represented as a one-hot vector, where each member of the vector is assigned a value of 1, indicating its presence. The vector representing the input word can be obtained by performing matrix multiplication between the word embedding matrix and the one-hot vector. However, the matrix multiplication result is essentially equivalent to choosing the ith row of the word embedding matrix. To optimize computational efficiency, we can expedite the process by exclusively choosing the row that corresponds to the input word.

The following graph shows the fundamental process of the dense layer.



where z is the output of the dense layer, w is the weight (word embedding) matrix and x is the one-hot vector.

The softmax function outputs a vector of probabilities for each word appearing near the given input word (Brownlee, 2020). The following equation shows the formula of what the softmax function is.



where z is the input vector, and k is the number of classes. The term inside the exponent is the logit, which is the raw score for each class. The softmax function normalizes these logits into a probability distribution over the classes.

The output from the forward pass is then compared with the desired outputs to be compared to the errors (Cilimkovic, 2015).

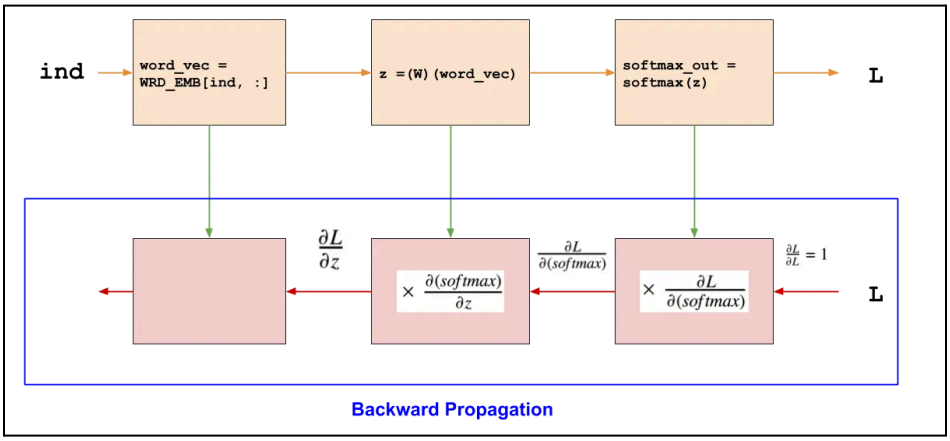
3. Computation of Cost (L)

The objective of this function is to evaluate how well the predicted probabilities correspond to the actual labels. This function serves as a metric for evaluating the performance of the model during training.

$$L = -\frac{1}{m}\sum_{j=0}^{m}\sum_{i=0}^{n} y_{ij}\log \hat{y}_{ij}$$

where L is cross-entropy loss, m is the number of training samples, y_{ij} is the true label and $\hat{y}_{ij}$ is the predicted probability.

4. Backward pass (Backpropagation)



In the backward pass, the gradient of the loss function is calculated in relation to the weights of a neural network (Kumawat, 2023). This enables the efficient training of deep neural networks. It includes the chain rule in Calculus. We can calculate the gradients of the weights in the dense layer and word embedding layer with the following formula.

$$\frac{\partial L}{\partial Z} = softmax\_out - Y$$

where Y is the one-hot encoded matrix, softmax_out is the output from the softmax function, and dL_dZ represents the gradient of the loss with respect to the output of the softmax function.

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} \times \frac{\partial Z}{\partial W} = \frac{\partial L}{\partial Z} \times word\_vec^T$$

$$\frac{\partial L}{\partial word\_vec} = \frac{\partial L}{\partial Z} \times \frac{\partial Z}{\partial word\_vec} = W^T \times \frac{\partial L}{\partial Z}$$

where dL_dZ is the gradient of the loss for the output of the softmax, dL_dW represents the gradient of the loss concerning the weights of the dense layer, and dL_dword_vec represents the gradient to the word vectors.

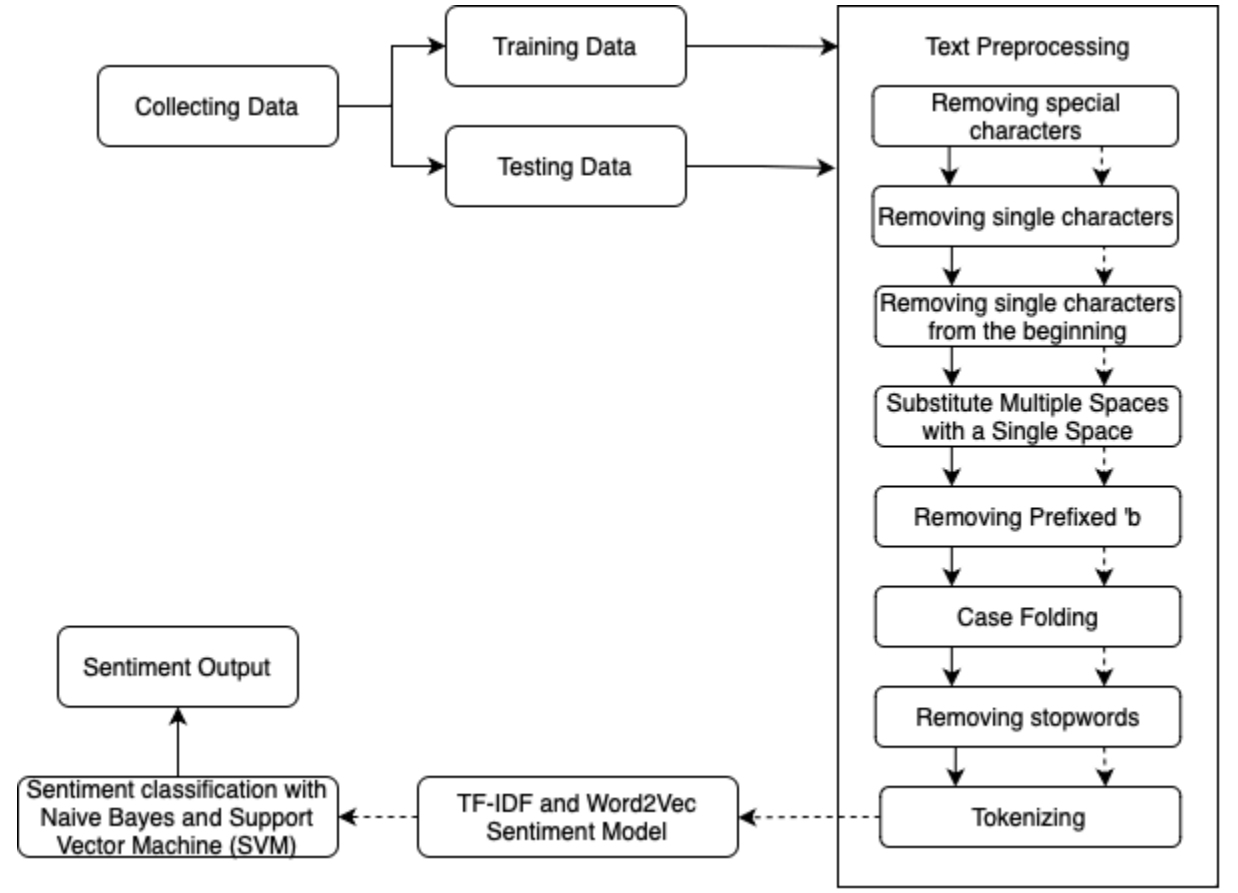
After that, we need to update the weights with the following formula:

$$W_{cur} = W_{pre} - \alpha \frac{\partial L}{\partial W_{pre}}$$

This function updates the weights of the dense layer by subtracting the gradient of the loss with respect to these weights (dL_dW) multiplied by the learning rate. This is a gradient descent step specifically for the dense layer weights. This process helps in optimizing the network's performance by gradually adjusting the parameters to minimize the loss function over the training data.

The forward and backward passes are iteratively performed until the error reaches a sufficiently low level (Kumawat, 2023).

The Flow



Research Framework

What was done

Libraries/Modules Used

- Pandas. This module is used for data manipulation and analysis, particularly for importing datasets and organizing data into data frames.
- Numpy. This module is essential for handling arrays, matrices, and mathematical computations in this project.
- Re. This module is useful for text processing tasks, especially in cleaning and manipulating raw text data by matching patterns, removing special characters, and many more.
- Psutil. This module helps analyze memory usage or resource utilization during the model-building process.
- Time. This module enables tracking time-related metrics during the sentiment analysis process.
- Matplotlib and Seaborn. These modules are mainly used for visualizing plots (count plot and pie plot), and graphs for sentiment trends, distributions, or analysis results.
- WordCloud. This module is used
- NLTK. This module is used for natural language processing and text preprocessing tasks, specifically for text tokenization, stemming, and removing stopwords.
- Sklearn. This module provides various machine learning algorithms and tools, including data splitting, model evaluation, and classification.
- VaderSentiment. This module/library imported from an external source is designed to predict sentiment (positive, neutral, and negative) from text data.

Data Collection

The data used comes from reviews on products on Amazon as many as 3k, 5k, and 35k collected from the Kaggle dataset. After collecting data, then the data is stored in DataFrames to be processed further for text preprocessing and many more.

	reviewerName	rating	review	date
0	0mie	5.0	Purchased this for my device, it worked as adv...	2013-10-25
1	1K3	4.0	it works as expected. I should have sprung for...	2012-12-23
2	1m2	5.0	This think has worked out great.Had a diff. br...	2013-11-21
3	2&1/2Men	5.0	Bought it with Retail Packaging, arrived legit...	2013-07-13
4	2Cents!	5.0	It's mini storage. It doesn't do anything els...	2013-04-29
...	...	...	...	...
4908	ZM "J"	1.0	I bought this Sandisk 16GB Class 10 to use wit...	2013-07-23
4909	Zo	5.0	Used this for extending the capabilities of my...	2013-08-22
4910	Z S Liske	5.0	Great card that is very fast and reliable. It ...	2014-03-31
4911	Z Taylor	5.0	Good amount of space for the stuff I want to d...	2013-09-16
4912	Zza	5.0	I've heard bad things about this 64gb Micro SD...	2014-02-01

Text Processing

At this stage, preprocessing is performed to clean the data from noise and inconsistent data.

```
processed_reviews = []

# Iterate through each review
for review in range(0, len(X)):
    # Remove all special characters
    processed_review = re.sub(r'\W', ' ', str(df['review'][review]))

    # Remove all single characters
    processed_review = re.sub(r'\s+[a-zA-Z]\s+', ' ', processed_review)

    # Remove single characters from the beginning
    processed_review = re.sub(r'\^[a-zA-Z]\s+', ' ', processed_review)

    # Substitute multiple spaces with a single space
    processed_review = re.sub(r'\s+', ' ', processed_review, flags=re.I)

    # Remove prefixed 'b' (if applicable, assuming X is a list of strings)
    processed_review = re.sub(r'^b\s+', '', processed_review)

    # Convert to lowercase
    processed_review = processed_review.lower()

    # Append to the empty list created earlier
    processed_reviews.append(processed_review)
```

This code iterates through each review in the DataFrame's 'Review' column, applies a series of text cleaning steps using regular expressions, and stores the processed text in a new column 'processed_reviews'.

```
stopWords = set(stopwords.words('english') + ['the', 'a', 'an', 'i', 'he', 'she',
'they', 'to', 'of', 'it', 'from'])

def removeStopWords(stopWords, rvw_txt):
    newtxt = ' '.join([word for word in rvw_txt.split() if word.lower() not in
stopWords])
  return newtxt
```

This code defines a function that takes a piece of text as input and removes common English stopwords (from the NLTK library) with additional specified stopwords, generating the modified text without the stopwords.

```
#splitting text into words
tokenList=[]

for indx in range(len(df)):
    token= word_tokenize(df['clean_review_text'][indx])
    tokenList.append(token)
```

This code tokenizes the cleaned review text in each row of the DataFrame into single words.

Sentiment Classification for Label

```
sentiment_model = SentimentIntensityAnalyzer()
sentiment_scores=[]
sentiment_score_flag = []
for text in df['clean_review_text']:
      sentimentResults = sentiment_model.polarity_scores(text)
      sentiment_score = sentimentResults["compound"]
      #print(sentimentResults)
       #The compound value reflects the overall sentiment ranging from -1 being very
negative and +1 being very positive.
      sentiment_scores.append(sentiment_score)
      # marking the sentiments as positive, negative and neutral
      if sentimentResults['compound'] >= 0.05 :
          sentiment_score_flag.append('positive')
       elif sentimentResults['compound'] <= - 0.05 :
          sentiment_score_flag.append('negative')
       else :
          sentiment_score_flag.append('neutral')
```

```
df['scores'] = sentiment_scores
df['scoreStatus'] = sentiment_score_flag
```

This code uses VADER sentiment analysis to compute sentiment scores for each review text in the 'clean_review_text' column of the DataFrame. It generates two new columns: 'scores', storing the sentiment scores and 'scoreStatus', representing the categorized sentiment labels for each review. This will be used as the label for our model training.

TF-IDF Vector Representation (Word Embedding Process)

After preprocessing the review and defining our label, the tf idf model is built based on the formula mentioned in the concept of algorithms.

```
def calculate_tf(documents):
    tf = defaultdict(lambda: defaultdict(int))

    for idx, doc in enumerate(documents):
        terms = doc.lower().split()  # Tokenization and lowercasing
        total_terms = len(terms)
        for term in set(terms):  # Using set to count each term only once
            tf[term][idx] = terms.count(term) / total_terms

    return tf
```

This function basically calculates the term frequency (TF) for each term in each document, and store these TF values in a nested dictionary indexed by terms and document indices.

```
def calculate_idf(tf, total_docs):
    idf = {}
    for term, doc_freq in tf.items():
        idf[term] = math.log10(total_docs / (1 + len(doc_freq)))
    return idf
```

This function takes the Term Frequency (TF) information and calculates the Inverse Document Frequency (IDF) for each term based on the total number of documents in the collection. The output is a dictionary where each term is associated with its calculated IDF value.

```
def compute_tfidf(tf, idf):
    tfidf = defaultdict(lambda: defaultdict(float))

    for term, doc_freq in tf.items():
        for doc_idx, tf_score in doc_freq.items():
            tfidf[term][doc_idx] = tf_score * idf[term]
```

```
    return tfidf
```

This function takes the tf and idf information, calculates the TF-IDF scores for each term in each document, and stores the TF-IDF values in a nested dictionary.

```
def create_tfidf_vectors(tfidf, num_documents):
    tfidf_vectors = []
    for doc_idx in range(num_documents):
        vector = [tfidf[term][doc_idx] for term in tfidf]
        tfidf_vectors.append(vector)
    return tfidf_vectors
```

This function creates the TF-IDF vectors for each document. Each vector represents a document and consists of TF-IDF scores for terms appeared in the document.

```
# Apply TF-IDF to the data
tf_dict = calculate_tf(df["clean_review_text"])
idf_dict = calculate_idf(tf_dict, len(df["clean_review_text"]))
tf_idf = compute_tfidf(tf_dict, idf_dict)
tf_idf_vectors = create_tfidf_vectors(tf_idf, len(df["clean_review_text"]))
```

After creating these functions, this code applies the TF-IDF technique to the data in the 'clean_review_text' column. Final output is the TF-IDF scores and vectors that represent the importance of each word within each document correspond with the entire document collection. These vectors are then used as features for classification model training.

Word2Vec Vector Representation (Word Embedding Process)

After preprocessing the review and defining our label, the word2vec model is built based on the steps mentioned in the concept of algorithms.

1. ***Parameter Initialization***

```
# initialize the word embedding matrix
def initialize_wrd_emb(vocab_size, emb_size):
    """
    vocab_size: int. vocabulary size of your corpus or training data
    emb_size: int. word embedding size. How many dimensions to represent each
vocabulary
    """
    WRD_EMB = np.random.randn(vocab_size, emb_size) * 0.01

    assert(WRD_EMB.shape == (vocab_size, emb_size))
    return WRD_EMB
```

This function generates the word embeddings for words in the entire vocabulary using random initialization technique.

```
# initialize the weight matrix for a dense layer
def initialize_dense(input_size, output_size):
    """
    input_size: int. size of the input to the dense layer
    output_szie: int. size of the output out of the dense layer
    """
    W = np.random.randn(output_size, input_size) * 0.01

    assert(W.shape == (output_size, input_size))
    return W
```

This function initializes the weight for a dense layer in the neural network using random initialization technique.

```
# initialize parameters for a neural network model
def initialize_parameters(vocab_size, emb_size):
    WRD_EMB = initialize_wrd_emb(vocab_size, emb_size)
    W = initialize_dense(emb_size, vocab_size)

    parameters = {}
    parameters['WRD_EMB'] = WRD_EMB
    parameters['W'] = W

    return parameters
```

This function combines the initialized word embeddings and dense layer weights into a dictionary called ‘parameters’.

In summary, this code sets up the initial parameters needed for our word2vec model, specifically for word embeddings and mapping word embeddings to the output vocabulary through a dense layer.

2. ***Forward Propagation***

```
# convert word indices to word vectors
def ind_to_word_vecs(inds, parameters):
    """
    inds: numpy array. shape: (1, m)
    parameters: dict. weights to be trained
    """
    m = inds.shape[1]
    WRD_EMB = parameters['WRD_EMB']
```

```
    word_vec = WRD_EMB[inds.flatten(), :].T

    assert(word_vec.shape == (WRD_EMB.shape[1], m))

    return word_vec
```

This function converts input word indices into word vectors by extracting corresponding word embeddings. It uses the input word to fetch the corresponding word embeddings and transposes the resulting matrix.

```
# perform linear transformation with a dense layer
def linear_dense(word_vec, parameters):
    """
    word_vec: numpy array. shape: (emb_size, m)
    parameters: dict. weights to be trained
    """
    m = word_vec.shape[1]
    W = parameters['W']
    Z = np.dot(W, word_vec)

    assert(Z.shape == (W.shape[0], m))

    return W, Z
```

This function performs the linear transformation by computing the dot product between the weight matrix and the word vectors to generate the output of the dense layer.

```
# apply softmax activation to the output
def softmax(Z):
    """
    Z: output out of the dense layer. shape: (vocab_size, m)
    """
     softmax_out = np.divide(np.exp(Z), np.sum(np.exp(Z), axis=0, keepdims=True)
+ 0.001)

    assert(softmax_out.shape == Z.shape)

    return softmax_out
```

This function computes the softmax function on the output of the dense layer.

```
# perform forward propagation through the neural network
def forward_propagation(inds, parameters):
    word_vec = ind_to_word_vecs(inds, parameters)
    W, Z = linear_dense(word_vec, parameters)
    softmax_out = softmax(Z)
```

```
    caches = {}
    caches['inds'] = inds
    caches['word_vec'] = word_vec
    caches['W'] = W
    caches['Z'] = Z

    return softmax_out, caches
```

This function applies all the above functions to perform forward propagation and creates a dictionary containing values for use in backward propagation.

3. ***Cost Function***

```
def cross_entropy(softmax_out, Y):
    """
    softmax_out: output out of softmax. shape: (vocab_size, m)
    """
    m = softmax_out.shape[1]
    cost = -(1 / m) * np.sum(np.sum(Y * np.log(softmax_out + 0.001), axis=0,
keepdims=True), axis=1)
    return cost
```

This function calculates the cross-entropy cost associated with the output of the softmax function and the provided true distribution of words (Y).

4. ***Backward Propagation***

```
def softmax_backward(Y, softmax_out):
    """
    Y: labels of training data. shape: (vocab_size, m)
    softmax_out: output out of softmax. shape: (vocab_size, m)
    """
    dL_dZ = softmax_out - Y

    assert(dL_dZ.shape == softmax_out.shape)
    return dL_dZ
```

This function computes the gradient loss with respect to the output of the softmax function.

```
def dense_backward(dL_dZ, caches):
    """
    dL_dZ: shape: (vocab_size, m)
    caches: dict. results from each steps of forward propagation
    """
    W = caches['W']
```

```
    word_vec = caches['word_vec']
    m = word_vec.shape[1]

    dL_dW = (1 / m) * np.dot(dL_dZ, word_vec.T)
    dL_dword_vec = np.dot(W.T, dL_dZ)

    assert(W.shape == dL_dW.shape)
    assert(word_vec.shape == dL_dword_vec.shape)

    return dL_dW, dL_dword_vec
```

This function computes the gradient loss concerning the weights of the dense layer and the gradient loss to the word vectors.

```
# perform backward propagation to compute gradients
def backward_propagation(Y, softmax_out, caches):
    dL_dZ = softmax_backward(Y, softmax_out)
    dL_dW, dL_dword_vec = dense_backward(dL_dZ, caches)

    gradients = dict()
    gradients['dL_dZ'] = dL_dZ
    gradients['dL_dW'] = dL_dW
    gradients['dL_dword_vec'] = dL_dword_vec

    return gradients
```

This function applies all the above functions and stores them into a new dictionary called ‘gradients’.

```
# update the parameters using gradient descent
def update_parameters(parameters, caches, gradients, learning_rate):
    vocab_size, emb_size = parameters['WRD_EMB'].shape
    inds = caches['inds']
    dL_dword_vec = gradients['dL_dword_vec']
    m = inds.shape[-1]

    parameters['WRD_EMB'][inds.flatten(), :] -= dL_dword_vec.T * learning_rate

    parameters['W'] -= learning_rate * gradients['dL_dW']
```

This function updated the parameters (word embeddings and weights) of the network based on the computed gradients using gradient descent scaled by the learning rate.

5. *Skipgram Model*

```
def skipgram_model_training(X, Y, vocab_size, emb_size, learning_rate, epochs,
batch_size=256, parameters=None, print_cost=False, plot_cost=True):
    costs = []
    m = X.shape[1]

    if parameters is None:
        parameters = initialize_parameters(vocab_size, emb_size)

    begin_time = datetime.now()
    for epoch in range(epochs):
        epoch_cost = 0
        batch_inds = list(range(0, m, batch_size))
        np.random.shuffle(batch_inds)
        for i in batch_inds:
            X_batch = X[:, i:i+batch_size]
            Y_batch = Y[:, i:i+batch_size]

            softmax_out, caches = forward_propagation(X_batch, parameters)
            gradients = backward_propagation(Y_batch, softmax_out, caches)
            update_parameters(parameters, caches, gradients, learning_rate)
            cost = cross_entropy(softmax_out, Y_batch)
            epoch_cost += np.squeeze(cost)

        costs.append(epoch_cost)
    end_time = datetime.now()
    print('training time: {}'.format(end_time - begin_time))

    if plot_cost:
        plt.plot(np.arange(epochs), costs)
        plt.xlabel('# of epochs')
        plt.ylabel('cost')
    return parameters
```

This function is used for training a Skip-gram word embedding model using a specified number of epochs (training iterations) and batch-wise optimization. It also measures and prints the training time for the entire training process using ‘datetime.now()’ to calculate the duration. Furthemore, it plots the cost agains the number of the epochs using Matplotlib. The final output of this function is the trained parameters (weights) of the Skip-gram model for use as features in our sentiment classification model.

Classification Model Training using LinearSVC and GaussianNB

The choice between LinearSVC (Support Vector Classifier) and Gaussian Naive Bayes (GaussianNB) for classification models are based on the effectiveness with high dimensional data and robustness to overfitting our models. They can also handle large datasets well and perform well where there is redundant features.

```
# Assuming 'labels' contains the positive/negative labels for each text
X_train, X_test, y_train, y_test = train_test_split(tf_idf_vectors, df["scoreStatus"],
test_size=0.2, random_state=42)

# Initialize the classifier
clf = LinearSVC()

# Train the classifier
clf.fit(X_train, y_train)

# Making predictions on the test set
y_pred = clf.predict(X_test)

# Calculating accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on test set: {accuracy}")

# Classification report
report = classification_report(y_test, y_pred, zero_division=0)
print(report)
```

```
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.2,
random_state=42)

# initialize the classifier
gnb = GaussianNB()

# train the classifier
gnb.fit(X_train, y_train)

# make predictions on the test set
y_pred = gnb.predict(X_test)
```

```
# calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy on test set: {accuracy}")

# classification report
report = classification_report(y_test, y_pred, zero_division=0)
print("\n", report)
```

These codes perform the entire process of splitting the data, training LinearSVC and GaussianNB classifiers on TF-IDF vectors and Word2Vec vectors, making predictions on the test set, calculating accuracy, and printing the classification report summarizing the model's performance on the test data. *(The results are gonna be discussed in Discussion and Analyzation Section)*

Calculating Word Embeddings Process Time

In creating the vectors and word embeddings in TF-IDF and Word2Vec, we also calculated the time by looking at how long the code runs in our device and by printing the time using datetime.now() function.

Calculating Memory Usage

```
# check memory usage of applying TF-IDF to the data
def print_memory_usage():
    process = psutil.Process()
    memory_info = process.memory_info()

    print(f"Memory Usage: {memory_info.rss / (1024 ** 2):.2f} MB (Resident Set Size)")
    print(f"Virtual Memory: {memory_info.vms / (1024 ** 2):.2f} MB")

@profile
def train_skipgram_model_memory():
    paras = skipgram_model_training(X, Y_one_hot, vocab_size, 50, 0.05, 10,
batch_size=64, parameters=None)
    print_memory_usage()

train_skipgram_model_memory()
```

This code performs memory usage profiling for the training skipgram model process. It uses the 'psutil' module to gather memory-related information for the current process. The outputs are memory usage and virtual memory usage of the process in megabytes (MB).

Calculating Training Time

```
start_time = time.time()

# Code for training your model
clf.fit(X_train, y_train)

end_time = time.time()
training_time = end_time - start_time
print(f"Training Time: {training_time} seconds")
```

This code measures the time before and after the training process, then calculates and displays the time taken to provide insight into the computational resources used for model training.

Sentiment Classification

```
def predict_sentiment(new_text, model, tf_dict, idf_dict):
    # Calculate TF for the new text
    terms = new_text.lower().split()
    total_terms = len(terms)
    tf_new_text = defaultdict(int)
    for term in set(terms):
        tf_new_text[term] = terms.count(term) / total_terms

    # Calculate TF-IDF for the new text using IDF values from the entire corpus
    tfidf_new_text = defaultdict(float)
    for term, tf_score in tf_new_text.items():
        if term in idf_dict:
            tfidf_new_text[term] = tf_score * idf_dict[term]

    # Create TF-IDF vector for the new text
    tfidf_vector_new_text = [tfidf_new_text[term] for term in tf_dict]

    # Reshape to match the input format expected by the model (1 sample, n_features)
    tfidf_vector_new_text = [tfidf_vector_new_text]

    # Predict sentiment using the trained model
    prediction = model.predict(tfidf_vector_new_text)
```

```
    return prediction[0]  # Assuming a single prediction for the new text

# Example usage:
new_text = "The customer service was bad"
predicted_sentiment = predict_sentiment(new_text, clf, tf_dict, idf_dict)
print(f"Predicted sentiment: {predicted_sentiment}")
```

```
Predicted sentiment: ['positive']
```

```
def predict_sentiment(new_text, model, tf_dict, idf_dict):
    # Calculate TF for the new text
    terms = new_text.lower().split()
    total_terms = len(terms)
    tf_new_text = defaultdict(int)
    for term in set(terms):
        tf_new_text[term] = terms.count(term) / total_terms

    # Calculate TF-IDF for the new text using IDF values from the entire corpus
    tfidf_new_text = defaultdict(float)
    for term, tf_score in tf_new_text.items():
        if term in idf_dict:
            tfidf_new_text[term] = tf_score * idf_dict[term]

    # Create TF-IDF vector for the new text
    tfidf_vector_new_text = [tfidf_new_text[term] for term in tf_dict]

    # Reshape to match the input format expected by the model (1 sample, n_features)
    tfidf_vector_new_text = [tfidf_vector_new_text]

    # Predict sentiment using the trained model
    prediction = model.predict(tfidf_vector_new_text)
    return prediction[0]  # Assuming a single prediction for the new text

# Example usage:
new_text = "The customer service was good"
predicted_sentiment = predict_sentiment(new_text, clf, tf_dict, idf_dict)
print(f"Predicted sentiment: {predicted_sentiment}")
```

```
Predicted sentiment: ['positive']
```

```
new_text = "this product was very bad"
tokenized_new_text = new_text.split()
```

```
new_text_word_indices = [word_to_id[word] for word in tokenized_new_text if word in
word_to_id]
new_text_word_vecs = ind_to_word_vecs(np.array(new_text_word_indices).reshape(1, -1),
paras)

avg_embedding_new_text = np.mean(new_text_word_vecs, axis=1)

predicted_sentiment = clf.predict(avg_embedding_new_text.reshape(1, -1))
print(f"Predicted sentiment: {predicted_sentiment}")
```

```
Predicted sentiment: ['negative']
```

```
new_text = "this product was very good"
tokenized_new_text = new_text.split()

new_text_word_indices = [word_to_id[word] for word in tokenized_new_text if word in
word_to_id]
new_text_word_vecs = ind_to_word_vecs(np.array(new_text_word_indices).reshape(1, -1),
paras)

avg_embedding_new_text = np.mean(new_text_word_vecs, axis=1)

predicted_sentiment = clf.predict(avg_embedding_new_text.reshape(1, -1))
print(f"Predicted sentiment: {predicted_sentiment}")
```

```
Predicted sentiment: ['positive']
```

After we had our models trained, we obtained the word embeddings and vector representations in the new text, and predicted the sentiment of the text using the trained classifier. The results turned out to be similar to our desired output.

Discussion and Analyzation

Making Vectors (Time)

Data Size	TF-IDF	Word2Vec
500	1.5s	8m 4.5s
1000	1.4s	18m 25s
1500	1.6s	16m 59s

The table shows that the time it takes to make vectors increases as the data size increases for all three data points. For example, at 500 data sizes, it takes 1.5 seconds to make vectors using TF-IDF, while it takes 8 minutes and 45 seconds to make vectors using Word2Vec. At 1000 data sizes, it takes 1.4 seconds to make vectors using TF-IDF, while it takes 18 minutes and 25 seconds to make vectors using Word2Vec. At 1500 data size, it takes 1.6 seconds to make vectors using TF-IDF, while it takes 16 minutes and 59 seconds to make vectors using Word2Vec.

There are a few possible reasons for this. First, as the data size increases, more data points need to be processed, which takes more time. Second, the algorithms used by TF-IDF and Word2Vec are more complex than the algorithms used by Data Size, which also takes more time.

Overall, the table shows that TF-IDF is the fastest method for making vectors, followed by Data Size, and then Word2Vec.

Training Time

Data Size	TF-IDF	Word2Vec
500	0.087s	0.03s
1000	0.14s	0.05s
1500	0.29s	0.175s

The table compares the training times of TF-IDF and Word2Vec models across various data sizes. It can be seen that both algorithms exhibit increasing training times with larger datasets, as higher demands of processing are required for more extensive text. However, significant differences emerge in their scaling patterns. TF-IDF exhibits a near-linear increase in training time with data size. This may have happened due to its relatively straightforward calculations involving term frequencies and document frequencies. In contrast, Word2Vec presents a steeper rise in training time, reflecting the complexity of its neural network architecture, which requires substantial computations for generating word embeddings.

For smaller datasets involving 500 and 1000 data, TF-IDF demonstrates clear training time advantages, outperforming Word2Vec by almost 80% at 1000 documents. However, the gap narrows as data size increases to 1500 documents, suggesting that Word2Vec might eventually counterbalance its initial computational overhead.

Memory Usage

Data Size	TF-IDF	Word2Vec
500	436.7 MB	1062 MB
1000	500.39 MB	930.73 MB
1500	1045.25 MB	1626.11 MB

The table compares the memory usage of TF-IDF and Word2Vec models across various data sizes. It can be seen that Word2Vec emerges as a memory hog, as it consistently requires double the amount of memory the larger the dataset compared to TF-IDF. This may have happened due to TF-IDF operating on simpler term frequencies and document frequencies than Word2Vec that requires intricate neural network architecture that necessitates storing complex word embeddings in memory. This indicates that for large-scale tasks, TF-IDF's memory efficiency might outshine Word2Vec's.

Virtual Memory

Data Size	TF-IDF	Word2Vec
500	400719.05 MB	401948.23 MB
1000	400854.23 MB	406948.16 MB
1500	401877.48 MB	39740.18 MB

The table shows a table of virtual memory size, TF-IDF, and Word2Vec. The table shows that the size of virtual memory required for TF-IDF and Word2Vec increases as the data point size increases. For example, at 500 data size, the virtual memory required for TF-IDF is 400719.05 MB, while the virtual memory required for Word2Vec is 401948.23 MB. At 1000 data sizes, the virtual memory required for TF-IDF is 400854.23 MB, while the virtual memory required for Word2Vec is 406948.16 MB. At 1500 data size, the virtual memory required for TF-IDF is 401877.48 MB, while the virtual memory required for Word2Vec is 39740.18 MB.

This suggests that TF-IDF and Word2Vec require more virtual memory as the data size increases. This is because TF-IDF and Word2Vec are both machine-learning algorithms that require large amounts of memory to store the data they are processing. As the data size increases, the amount of memory required by TF-IDF and Word2Vec also increases.

Classification Report

Measurement	Linear SVC		Naive Bayes	
	TF-IDF	Word2Vec	TF-IDF	Word2Vec
Accuracy	94%	77%	86%	41%
Precision	94%	70%	92%	75%
Recall	94%	77%	86%	41%
F1 Score	94%	77%	87%	45%

The table compares the performance of TF-IDF and Word2Vec for sentiment analysis using two machine learning algorithms, Linear Support Vector Classifier (Linear SVC) and Naive Bayes. Overall, the results show that TF-IDF outperforms Word2Vec on all metrics for both algorithms. This suggests that TF-IDF is better at capturing the sentiment of text data than Word2Vec.

There are a few possible explanations for this. First, TF-IDF weights words based on their frequency in the document and their rarity in the corpus. This means that words that are more likely to be associated with a particular sentiment will be given more weight. Second, TF-IDF does not take into account the context of words, while Word2Vec does. This means that TF-IDF may be more robust to noise and errors in the data.

Conclusion and Recommendation

After taking into account all of the findings that came to light, a comprehensive review of TF-IDF and Word2Vec from a variety of perspectives delivers insightful information that may be used to guide decision-making in machine learning initiatives. Duration efficiency, training duration, memory use, virtual memory, and classification performance were all evaluated, and the results showed clear advantages and disadvantages for each technique.

When choosing between TF-IDF and Word2Vec, it is crucial to take the trade-offs into account. The patterns that have been seen, such as TF-IDF's better time efficiency and reduced memory consumption, indicate that it might be a better option for tasks involving huge datasets or those with strict resource limitations. The scalability of TF-IDF is indicated by its nearly linear increase in training time with data size, whereas Word2Vec's steeper rise suggests possible issues with computational complexity.

Moreover, the best method still depends on the specific priorities of the project in question. TF-IDF is the best choice when speed is one of the main priorities of the project. For jobs like keyword extraction or simple text categorization, where semantic details are less important, this makes it an excellent choice. On the other hand, Word2Vec is perfect for applications that prioritize accuracy due to its capacity to capture minute associations between words, especially in sentiment analysis, topic modeling, and machine translation.

The assessment of sentiment analysis through the use of Naive Bayes and Linear Support Vector Classifier (Linear SVC) highlights the superiority of TF-IDF on several criteria. In terms of accuracy, precision, recall, and F1 score, TF-IDF routinely beats Word2Vec, indicating its resilience in identifying sentiment in textual data.

References

Addiga, A., & Bagui, S. (2022). Sentiment analysis on Twitter data using term Frequency-Inverse document frequency. *Journal of Computer and Communications*, *10*(08), 117–128. https://doi.org/10.4236/jcc.2022.108008

Ananthram, A. (2018). Random initialization for neural networks : a thing of the past. *Medium*. https://towardsdatascience.com/random-initialization-for-neural-networks-a-thing-of-the-past-bfcdd806bf9e

Brownlee, J. (2020). *Softmax Activation Function with Python*. MachineLearningMastery.com. https://machinelearningmastery.com/softmax-activation-function-with-python/

Cahyani, D. E., & Patasik, I. (2021). Performance comparison of TF-IDF and Word2Vec models for emotion text classification. *Bulletin of Electrical Engineering and Informatics*, *10*(5), 2780–2788. https://doi.org/10.11591/eei.v10i5.3157

Chatrath, S. K., Batra, G., & Chabba, Y. (2022). Handling consumer vulnerability in e-commerce product images using machine learning. *Heliyon*, *8*(9), e10743. https://doi.org/10.1016/j.heliyon.2022.e10743

Chen, I. (2021). Word2vec from Scratch with NumPy - Towards Data Science. *Medium*. https://towardsdatascience.com/word2vec-from-scratch-with-numpy-8786ddd49e72

Cilimkovic, M. (2015). Neural networks and back propagation algorithm. Institute of Technology Blanchardstown, Blanchardstown Road North Dublin, 15(1).

Gandhimani, H. (2023). *Building trust and loyalty: The importance of customer reviews and ratings in E-Commerce*. Boostmyshop. https://www.boostmyshop.com/the-ultimate-guide-to-e-commerce/importance-of-customer-reviews-and-ratings-for-e-commerce/

Guerreiro, J., & Rita, P. (2020). How to predict explicit recommendations in online reviews using text mining and sentiment analysis. *Journal of Hospitality and Tourism Management*, *43*, 269–272. https://doi.org/10.1016/j.jhtm.2019.07.001

Jeske, S. (2023, March 9). TF-IDF for SEO FAQs. *MarketMuse Blog*. https://blog.marketmuse.com/tf-idf-for-seo-faqs/#:~:text=Who%20Invented%20TF%20IDF%3F,inverse%20document%20frequency%20(1972).

Kumawat, T. (2023). Deep Learning Part 3: Parameter Initialization, BackPropagation, and Types of Error involved. *Medium*. https://medium.com/@tejpal.abhyuday/deep-learning-part-3-parameter-initialization-backpropagation-and-types-of-error-involved-6aa4f4e589bb

Liu, Y., Lu, J., Yang, J., & Mao, F. (2020). Sentiment analysis for e-commerce product reviews by deep learning model of Bert-BiGRU-Softmax. *Mathematical Biosciences and Engineering, 17*(6), 7819–7837. https://doi.org/10.3934/mbe.2020398

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

Mm, M., Batcha, N. K., & Raheem, M. (2020). Sentiment Analysis in E-Commerce: A Review on The Techniques and Algorithms. *ResearchGate*. https://www.researchgate.net/publication/339513566_Sentiment_Analysis_in_E-Commerce_A_Review_on_The_Techniques_and_Algorithms

Rajaraman, A., & Ullman, J. D. (2011). Mining of massive datasets. https://doi.org/10.1017/cbo9781139058452

Rong, X. (2014). word2vec parameter learning explained. arXiv preprint arXiv:1411.2738.

Sanzgiri, B. (2023). Analyzing customer reviews for e-commerce business. *42 Signals*. https://www.42signals.com/blog/customer-review-analysis-for-ecommerce-business/

Wisdomml. (2022, August 10). TF-IDF in NLP & How To Implement it in 4 Steps - Wisdom ML. *Wisdom ML*. https://wisdomml.in/tf-idf-in-nlp-how-to-implement-it-in-4-steps/

Link

Kaggle Dataset

3k: https://www.kaggle.com/datasets/nehaprabhavalkar/indian-products-on-amazon

5k: https://www.kaggle.com/datasets/tarkkaanko/amazon

35k: https://www.kaggle.com/datasets/datafiniti/consumer-reviews-of-amazon-products

GitHub Code

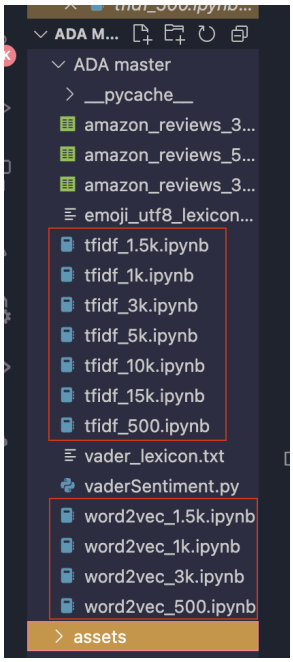
https://github.com/priscillabigaill/sentiment_analysis

Presentation Slides

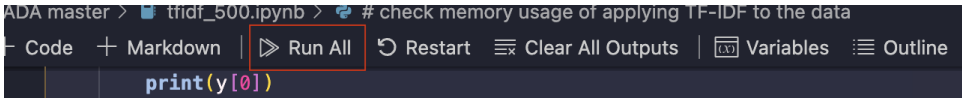
https://www.canva.com/design/DAF3NT8AZMw/CNZNXIYYKfUUjkJmP_qKEg/edit?utm_content=DAF3NT8AZMw&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton

Screenshots and User Manual

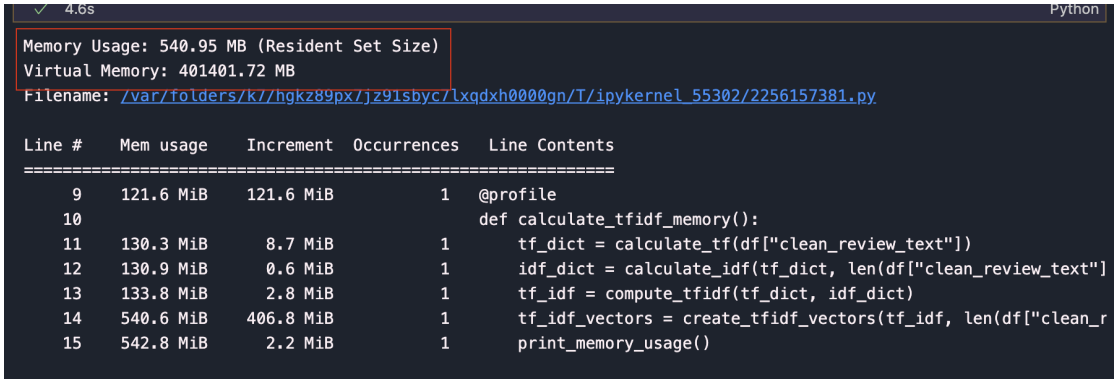
Step 1: Pick the file containing your preferred algorithm and data size.



Step 2: Click the "Run All" button on the top of the screen.



Step 3: Results



This first result shows the memory usage that the model uses to process the data according to the data size. Below is the virtual memory which was used in the process.

```
Accuracy on test set: 0.8933333333333333
Precision: 0.8933333333333333
Recall: 0.8933333333333333
F1-Score: 0.8933333333333333

              precision    recall  f1-score   support

    negative       0.93      0.56      0.69        45
     neutral       1.00      0.57      0.72        23
    positive       0.88      0.99      0.93       232

    accuracy                           0.89       300
   macro avg       0.94      0.70      0.78       300
weighted avg       0.90      0.89      0.88       300
```

The next result shows the overall scores of the model using the LinearSVC classifier. The complete results are at the bottom and the average score is at the top.

```
✓ 0.1s

Accuracy on test set: 0.8966666666666666

              precision    recall  f1-score   support

    negative       0.73      0.91      0.81        45
     neutral       0.68      0.91      0.78        23
    positive       0.97      0.89      0.93       232

    accuracy                           0.90       300
   macro avg       0.79      0.91      0.84       300
weighted avg       0.91      0.90      0.90       300
```

The next result also shows the overall scores of the model but using the GaussianNB classifier. The complete results are also at the bottom and the average score is at the top.

```
Model Size: 4.57763671875e-05 MB
```

This next result shows the size of the model measured in megabytes (MB).

```
Training Time: 0.2512171268463135 seconds
```

The last benchmarking result is the training time which is measured in seconds. These actual results will vary depending on the algorithm used and the data point selected before running the program.

```
# example usage:
new_text = "The customer service was bad"
predicted_sentiment = predict_sentiment(new_text, clf, tf_dict, idf_dict)
print(f"Predicted sentiment: {predicted_sentiment}")
```

✓ 0.0s

```
Predicted sentiment: negative
```

```
# example usage:
new_text = "The product was very good"
predicted_sentiment = predict_sentiment(new_text, clf, tf_dict, idf_dict)
print(f"Predicted sentiment: {predicted_sentiment}")
```

✓ 0.0s

```
Predicted sentiment: positive
```

At the end of the file, there will be 2 examples of sentiment analysis predicted by the model using example texts. The output of these will be the predictions.