

Real-time Social Media Sentiment Analysis

Statistical Learning and Data Analytics

Project Progress Report

Can Kocagil

Barış Kıcıman

EEE 485/585



Department of Electric & Electronics Engineering

Bilkent University

Ankara, Turkey

20.11.2021

TABLE OF CONTENTS

List of Figures	ii
List of Tables	ii
1. Introduction	1
2. Preprocessing Tweets	2
2.1. Lowercasing and removing unnecessary words	2
2.2. Text Normalization: Stemming and Lemmatization	3
2.3. Spell checking and unknown word removing	3
2.4. Non-frequent word removing	3
2.5. Non-frequent document removing	3
3. Tweet Analysis	4
4. Feature Extraction from Tweets	5
4.1. Count Vectorizing	5
4.2. TF-IDF	5
5. Modelling: Sentiment Analyzer	5
5.1. Train-test splitting	5
5.2. Mutual Information and Feature Selection	5
5.3. Naive Bayes	5
5.4. Logistic Regression	8
5.5. K-Nearest Neighbors	10
6. Work Packages and Gantt Chart	12
7. Conclusion	13
Appendix A. Code	14
References	67

LIST OF FIGURES

1	Positive-sentiment Tweets' Word Cloud	4
2	Negative-sentiment Tweets' Word Cloud	4
3	The visualization of the confusion matrix for Multinomial Naive Bayes	8
4	The visualization of the training & testing loss and accuracy of Logistic Regression-I	9
5	The visualization of the training & testing loss and accuracy of Logistic Regression-II	9
6	The visualization of the training & testing loss and accuracy of Logistic Regression-III	10
7	The excel sheet of work packages and gantt chart	12
8	The visualization of the work packages and Gantt chart	12

LIST OF TABLES

1	The confusion matrix for Bernoulli Naive Bayes	6
2	Mutual Information Feature Selection with Bernoulli Naive Bayes	7
3	The normalized confusion matrix for Multinomial Naive Bayes	8
4	The accuracies of k-NN with k=3 for various distance metrics	11
5	The accuracies of k-NN with k=5 for various distance metrics	11
6	The accuracies of k-NN with k=7 for various distance metrics	11

1. INTRODUCTION

Social media is a place where distributed social agents express their opinions, daily interactions, emotions, and feelings by posting multiple contents, tweets, images, videos, etc., such as on Facebook, Twitter, and Instagram. With the massive growth in social web services and media, these digital platforms inevitably reshape consumers' preferences and daily lives. As consumers' daily preferences are ever-changing, the value-driven enterprises moved their digital marketing agencies to social media, especially Facebook and Twitter. Hence, these ever-growing social data in any kind contains natural human-behavior-based information that facilitates autonomous decision-making. To extract behavioral information from complex and distributed social media data, sentiment analysis literature is emerged and is powered by statistical learning theories and deep learning. By acknowledging the social-economical context, we determined to develop behavior-aware machine learning algorithms specialized for understanding the social text data by extracting behavior-oriented sentiments.

As the project consists of multiple layers, we will introduce different learning algorithms for each phase by fixing the training data but varying the testing data. As the sentiment analysis task is supervised, we need social text data labels that represent the sentiments annotated by human specialists or autonomous annotation systems. To capture social media dynamics and stochasticity driven by human psychology, we looked for highly generic social text data which hopefully scalable and generalizable to real-life applications in the business context.

We found that Sentiment140 [3] dataset is appropriate for our needs and the scope of the project, as it consists of tweets and sentiments of brands & products that are highly scalable for brand needs, management, operations, and pollings.

By acknowledging the difficulty of producing sentiment analyzers by machines for multi-domain applications or general written language, we will be developing a sentiment analyzer for social brand-based products. The dataset consists of 1.6 million tweets with corresponding sentiments broadcasted between $[0, 4]$, being the polarity of the tweet $0 = \text{negative}$ and $4 = \text{positive}$.

By the inference time, we will fetch real-time tweet data from Twitter and produce predictions in real-time. Social tweets are highly unstructured, having a large corpus and stochastic by nature; the curse of dimensionality, sparsity, semantic information extraction, and real-time are the main challenges we will face, which constraints our computational comfort zone. Before blindly feeding the ML model by social text data, we need to incorporate comprehensive text preprocessing techniques into our pipeline, such as lower casing, stopping removal words, stemming, lemmatization and tokenization to convert unstructured text data representable numbers.

Further text preprocessing techniques, e.g., domain-agnostic or application-specific preprocessing, are applied, such as emoji, URL, date time, and non-linguistic text removals. On the other hand, human emotions are complex subjects to understand. Communication consists of several aggregators, such as verbals, tones, voices, modulations, micro-expressions, jests and mimics, and words; capturing the behavioral information by analyzing the text data is conceptually tricky. Understanding the tone is complicated to interpret verbally and more formidable to capture in textual data by machines.

As the social data is composed of both subjective and objective contexts, things are getting complicated while analyzing tones of massive textual sentiment datasets. Then, the polarity of words is another common challenge to overcome, as there are words such as "great" (strongly positive) or "worst" (strongly negative), which are pretty distinguishable. However, there are words in-between conjugations, such as "not so bad," that are a superficially arduous task for machines to capture. To understand the holistic side of the text, topic-based or aspect-based sentiment analysis can be applied.

Another linguistic challenge we will face is the sarcasm of the written language. People use irony and sarcasm in their social interactions, discussions, and negotiations. To clutch the irony and sarcasm of the social text data, capturing the semantic information lying in the social text data will be quite a significant aspect of the ML model as it can be trained accordingly.

Moreover, we acknowledge that non-text contents, such as emojis and images, can help us decode the content's sentiment. However, for this project's scope, we will not include the materials except for texts.

In this project's scope, by realizing our computing power and inability to use any deep learning or automatic differentiation tool, we will try to incorporate layers that can extract semantic information, e.g., Word2Vec Embedding layers or GloVe to perform cognition-aware sentiment analysis.

Then, from the machine learning perspective, we will design & develop classifier-based learning algorithms; as of the start, we introduced two versions of Naive Bayes that are Bernoulli and Multinomial, to construct our baseline model. Then, we moved to more advanced machine learning algorithms such as Logistic Regression. In addition to that, we implemented vectorized k-Nearest Neighbors algorithm.

Finally, we will construct Deep Neural Network (DNN) architectures with both embedding & linear layers to capture semantic information lying in the tweet data, with non-convex optimizers such as Stochastic Gradient Descent (SGD), AdaGrad, or Adam and cross entropy-based loss functions that hopefully accurately extract behavioral sentiments in the final demo.

2. PREPROCESSING TWEETS

Tweet data is unstructured by nature, as there are no rules to control the linguistic properties of texts such as grammar, semantics, punctuation, uniformity, and so on. Processing natural language requires vectorizing the texts by extracting linguistic properties from them. Before that, human specialists must incorporate prior linguistic knowledge into texts to decide what kind of textual information is required to perform sentiment-aware written language processing.

On the other hand, the computational requirements grow as the corpus size increases. So, there is an inevitable trade-off between corpus size and computational resources. Hence, the ultimate goal is to reduce the text size, and indirectly the corpus size, as much as possible while not losing linguistic information.

Acknowledging that we have limited computational resources and time to compute, prior language processing on texts is essential for most natural language processing tasks, especially for social media data. Hence, an in-depth text preprocessing pipeline is applied to the data before feeding the ML model. In the following subsections, applied preprocessing techniques will be discussed.

2.1. Lowercasing and removing unnecessary words. As of the start, 1.6 million tweets are lowercased by assuming capital versions of texts do not significantly impact the outcome. Then, social-data-specific word removal is applied. Each bullet list corresponds to a preprocessing technique for removing unnecessary phrases in tweets.

- **Removing mentions and hashtags**

As Twitter allows usage of mentions and hashtags, they are removed to reduce the corpus size and eliminate unnecessary overhead on the computing device.

- **Removing punctuation**

Punctuation removal is a common application for text preprocessing and is applied to 1.6M tweet data points.

- **Removing emojis**

Tweet data can contain emojis, and although emojis can interpret the semantics of the data, they are eliminated to reduce our corpus size.

- **Removing HTML codes**

We realized that partial HTML codes were included and subsequently removed as contain no extra information.

- **Removing URL components**

URLs are also removed from texts as they contain no semantically relevant information.

- **Removing multi-language stop words**

Removing stop words such as "a," "the," "is," "our" are also removed to reduce corpus size further. However, we realized that our dataset contains English phrases and includes another language such as Spanish. So, we determined to remove stop words that contain multiple languages. Also, since our initial corpus size is approximately 1.5 million, we incorporated diverse stop words from multiple resources.

2.2. Text Normalization: Stemming and Lemmatization. We normalize text to lessen its unpredictability and bring it closer to a predefined "standard." This reduces the quantity of diverse data that the computer has to cope with, resulting in increased efficiency. Normalization procedures such as stemming and lemmatization reduce a word's inflectional and occasionally derivationally related forms to a single base form.

2.2.1. Stemming. Stemming is the process of reducing words to their word stems or root forms. The purpose of stemming is to reduce related words to the same stem, even if the stem is not a dictionary word. For example, the words "connection," "connected," and "connecting" can all be reduced to the single word "connect."

2.2.2. Lemmatization. Unlike stemming, lemmatization lowers words to their base words, appropriately reducing inflected words and verifying that the root word belongs to the language. Because stemmers work on a single word without knowing the context, it is usually more complicated than stemming. A lemma is a root word in lemmatization. A lemma is a group of words in their canonical, dictionary, or citation form.

2.3. Spell checking and unknown word removing. Tweet data contains misspelled such as "mis-pelled" and non-linguistic, a spoken-language derivation of semantically meaningful words or sentence. However, as misspelled and non-linguistic words do not carry diverse semantics, and lead to additional overhead on the corpus size, are removed by using ready-to-go frameworks.

Considering the 1.6 million tweets with an average of 75 word-long documents and an in-depth pre-processing pipeline, the end-end text cleaning operation lasts over 9 hours. After accumulating the 9 hours of preprocessing, our corpus size has decreased from 1.5 million to 350 000.

2.4. Non-frequent word removing. Our corpus size is still over 300,000, which represents 350,000 features. Moreover, our 350,000 feature dimension will be highly-sparse, i.e., most entries are zeros. To further reduce our corpus, the non-frequent words are eliminated from the data. The frequency threshold is set as 100, which means that the words that occur less than 100 times in total data are removed.

2.5. Non-frequent document removing. As the last text preprocessing step, the tweet samples with under three words are eliminated, i.e., dropped out of the data.

3. TWEET ANALYSIS

The simple descriptive statistic-driven analysis is applied to tweet samples. The average length of tweet samples is 75, whereas the median is 69. The minimum-length tweet consists of 6 words, whereas the maximum one contains 374. Further analysis is done but not placed here for the sake of the fluency of the article.

Moreover, word-cloud visualizations of negative- and positive-sentiment are depicted. Here are the figures.



FIGURE 1. Positive-sentiment Tweets' Word Cloud



FIGURE 2. Negative-sentiment Tweets' Word Cloud

Even from the simple word-occurrence-based graphs, we can observe that there are highly overlapping words in-between positive- and negative-sentiment tweets.

4. FEATURE EXTRACTION FROM TWEETS

Feature extraction from words is accomplished by two methodologies: Count Vectorizing and TF-IDF. In the following sections, I will briefly talk about these algorithms.

4.1. Count Vectorizing. The Count vectorizer turns a set of text documents into a token count matrix. In other words, it is used to convert a collection of text documents to a vector of term/token counts.

4.2. TF-IDF. Term Frequency Inverse Document Frequency is abbreviated as TF-IDF. The Count Vectorizer calculates the number of times a word appears in a document, whereas the TF-IDF analyzes the total number of times a word appears in a document. The total number of documents is divided by the total number of documents containing the term "w." The inverse data frequency determines the weight of unusual words across all documents in the corpus.

5. MODELLING: SENTIMENT ANALYZER

For the scope of the project's progress, the algorithms are implemented and described in the following sections.

5.1. Train-test splitting. The vectorized tweet data is split as 80% training and 20% testing.

5.2. Mutual Information and Feature Selection. The mutual information (MI) of two random variables measures the mutual dependence between the two variables in the context of probability and information theory [6]. It quantifies the "amount of information" about one random variable received by observing the other random variable [6]. The entropy of a random variable, a fundamental notion in information theory that measures the expected "amount of information" carried in a random variable, is intimately tied to the concept of mutual information [6]. Mutual information is used as a feature ranking algorithm for the following ML models.

5.3. Naive Bayes. Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable [1]. Bayes' theorem states the following relationship, given class variable y and dependent feature vector x_1 through x_n [1]

$$(1) \quad P(y | x_1, \dots, x_n) = \frac{P(y)P(x_1, \dots, x_n | y)}{P(x_1, \dots, x_n)}$$

Using the naive conditional independence assumption that

$$(2) \quad P(x_i | y, x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n) = P(x_i | y)$$

for all, this relationship is simplified to

$$(3) \quad P(y | x_1, \dots, x_n) = \frac{P(y) \prod_{i=1}^n P(x_i | y)}{P(x_1, \dots, x_n)}$$

Since $P(x_1, \dots, x_n)$ is constant given the input, we can use the following classification rule [1]:

$$(4) \quad P(y | x_1, \dots, x_n) \propto P(y) \prod_{i=1}^n P(x_i | y) \quad \Downarrow$$

$$\hat{y} = \arg \max_y P(y) \prod_{i=1}^n P(x_i | y)$$

5.3.1. *Bernoulli Naive Bayes*. Bernoulli Naive Bayes algorithm is a variant of Naive Bayes algorithm which assumes attributes are independent and do not affect each other and gives all features equal weight [4]. A most important distinction of Bernoulli Naive Bayes is that it uses binary value features like true/false or 1/0. This algorithm assumes the prior distribution is a Bernoulli distribution and uses Bayes' Rule to maximize the posterior distribution.

$$(5) \quad p(x) = P[X = x] = \begin{cases} q = 1 - p & x = 0 \\ p & x = 1 \end{cases}$$

Bernoulli Naive Bayes implements Naive Bayes training and classification algorithms for data distributed according to multivariate Bernoulli distributions; several features may exist, but each is assumed to be a binary-valued (Bernoulli, boolean) variable [1]. As a result, samples must be represented as binary-valued feature vectors; if given any other type of data, a Bernoulli Naive Bayes instance may binarize it (depending on the binarize parameter) [1].

The decision rule for Bernoulli naive Bayes is based on

$$(6) \quad P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$$

which differs from Multinomial Naive Bayes rule in that it explicitly penalizes the non-occurrence of a feature that is an indicator for class, where the Multinomial variant would ignore a non-occurring feature [1]

We fitted Bernoulli Naive Bayes to our tweets, and the corresponding confusion matrix is provided below. One represents positive sentiments, whereas zero represents negative sentiments. Time duration is for fitting and predicting, and accuracy is also provided.

Time Consumed for fit: 16m 54s

Time Consumed for predict: 17.28 s

Accuracy Score: 72.323

Actual/Predicted	0	1
0	64799	26522
1	22700	64191

TABLE 1. The confusion matrix for Bernoulli Naive Bayes

Also, mutual information feature ranking is applied as follows. We initialize the training with 100 features that have the highest mutual information score against the target variables. Then, we extend the feature list by 100, i.e., we select 200 features with the highest mutual information score against the target variables. This process lasts until we utilize the complete feature set. This resulting table is provided below.

Iter num	Features	Feature Dim	Fitting Time (s)	Accuracy
7	[2, 513, 52, 218, 709, 943, 609, 681, 113, 619...	800	2022.8996	72.36
8	[223, 503, 477, 252, 745, 805, 874, 270, 608, ...	900	2069.8652	72.32
9	[514, 869, 549, 816, 155, 129, 296, 884, 101, ...	1000	2103.3668	72.32
6	[153, 278, 604, 836, 719, 475, 576, 654, 415, ...	700	1870.5676	72.31
5	[925, 686, 406, 700, 819, 201, 677, 330, 980, ...	600	1007.6962	72.19
4	[618, 161, 489, 285, 594, 145, 186, 151, 121, ...	500	864.9974	72.05
3	[172, 241, 731, 840, 94, 833, 6, 751, 688, 242...	400	923.5158	71.66
2	[66, 389, 711, 981, 737, 144, 588, 692, 983, 3...	300	17.5511	70.91
1	[388, 730, 525, 44, 942, 590, 842, 124, 229, 2...	200	13.9110	69.68
0	[722, 531, 405, 767, 54, 442, 316, 402, 834, 5...	100	989.1512	66.57

TABLE 2. Mutual Information Feature Selection with Bernoulli Naive Bayes

5.3.2. *Multinomial Naive Bayes.* The Multinomial Naive Bayes algorithm is also a variation of the Naive Bayes algorithm that makes the same naive assumptions about the data. The Multinomial Naive Bayes algorithm is specially designed for word processing by using word counts for calculating probability [2].

The Multinomial Naive Bayes is one of two standard naive Bayes versions used in text classification, and it implements the naive Bayes method for multinomially distributed data (where the data are typically represented as word vector counts, although tf-idf vectors are also known to work well in practice) [1].

The distribution is parametrized by vectors $\theta_y = (\theta_{y1}, \dots, \theta_{yn})$ for each class y , where n is the number of features (in text classification, the size of the vocabulary) and θ_{yi} is the probability $P(x_i | y)$ of feature i appearing in a sample belonging to class y .

The parameters θ_y is estimated by a smoothed version of maximum likelihood, i.e. relative frequency counting:

$$(7) \quad \hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

where $N_{yi} = \sum_{x \in T} x_i$ is the number of times feature i appears in a sample of class y in the training set T , and $N_y = \sum_{i=1}^n N_{yi}$ is the total count of all features for class y [1].

The smoothing priors $\alpha \geq 0$ accounts for features not present in the learning samples and prevents zero probabilities in further computations [1]. Setting $\alpha = 1$ is called Laplace smoothing, while $\alpha < 1$ is called Lidstone smoothing [1].

We fitted Multinomial Naive Bayes to our tweets, and the corresponding confusion matrix is provided below. One represents positive sentiments, whereas zero represents negative sentiments. Time duration is for fitting and predicting, and accuracy is also provided.

Accuracy Score: 72.38

The number of parameters to be estimated: 2001

Time Consumed for fit: 16m 49s

Time Consumed for predict: 0.75 s

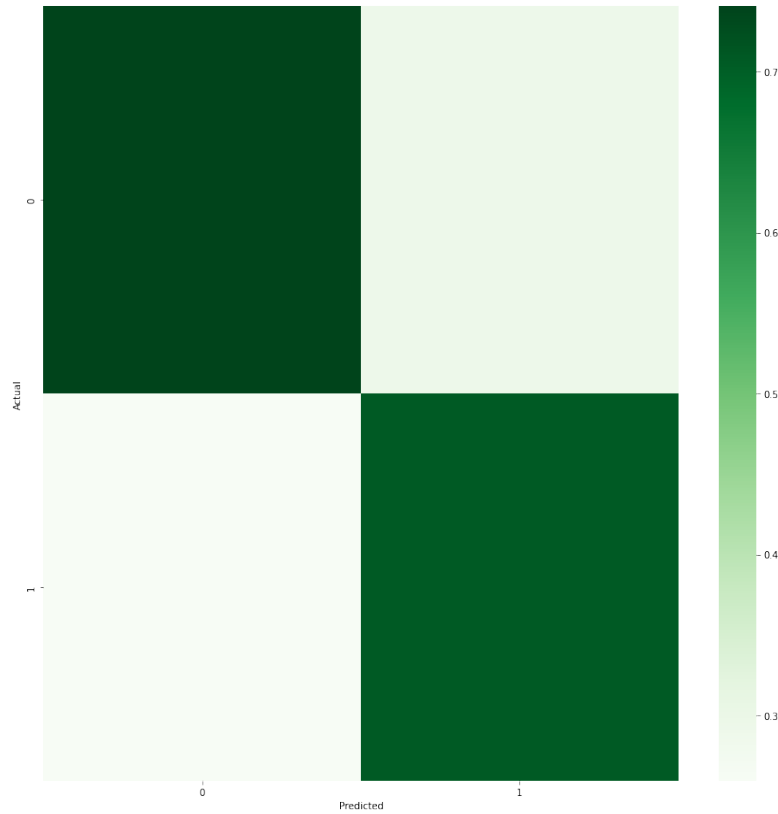


FIGURE 3. The visualization of the confusion matrix for Multinomial Naive Bayes

Actual/Predicted	0	1
0	0.740568	0.292373
1	0.259432	0.707627

TABLE 3. The normalized confusion matrix for Multinomial Naive Bayes

Long story short, we have accumulated to around 72% accuracy, which is our primary performance metric.

5.4. Logistic Regression. Logistic regression is a sigmoidal transformation of linear regression. Unlike its name, which includes regression, it is a linear classification algorithm. Given the weighted sum of input data, the sigmoid function outputs between 0 and 1 as probabilities of classes. Logistic regression assumes the output is discrete, and there is very little multicollinearity between features and a linear relationship between features and log-odds [5].

Given the input feature $\{X\}_{i=1}^n$, weight vector w and the bias term c , the positive class probability can be computed as follows

$$(8) \quad p(X_i) = \frac{1}{1 + e^{-(c + X_i^T w)}} \text{ for } i \in \{1, \dots, n\}$$

As an optimization problem, binary class ℓ_2 penalized logistic regression minimizes the following cost function:

$$(9) \quad \min_{w,c} \frac{1}{2} w^T w + C \sum_{i=1}^n \log(\exp(-y_i(X_i^T w + c)) + 1).$$

We fitted Logistic Regression to our tweets, and the corresponding learning curves, accuracy plots are provided below. One represents positive sentiments, whereas zero represents negative sentiments. Time duration is for fitting and predicting, and accuracy is also provided.

Time Consumed for fit: 32m 11s

Logistic Regression with hyperparameters (learning rate, batch size, epochs) = ((0.001, 64, 10000))

There are 1028 trainable parameters.

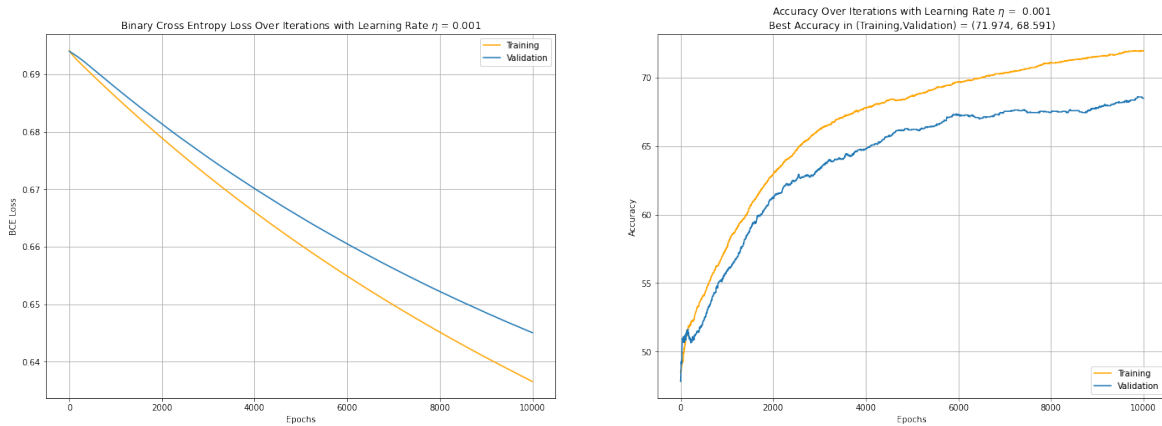


FIGURE 4. The visualization of the training & testing loss and accuracy of Logistic Regression-I

In the following figures and phrases, the results of logistic regression with different hyperparameters are provided.

Experiment-I

Time Consumed for fit: 32m 31s

Logistic Regression with hyperparameters (learning rate, batch size, epochs) = ((0.001, 64, 100000))

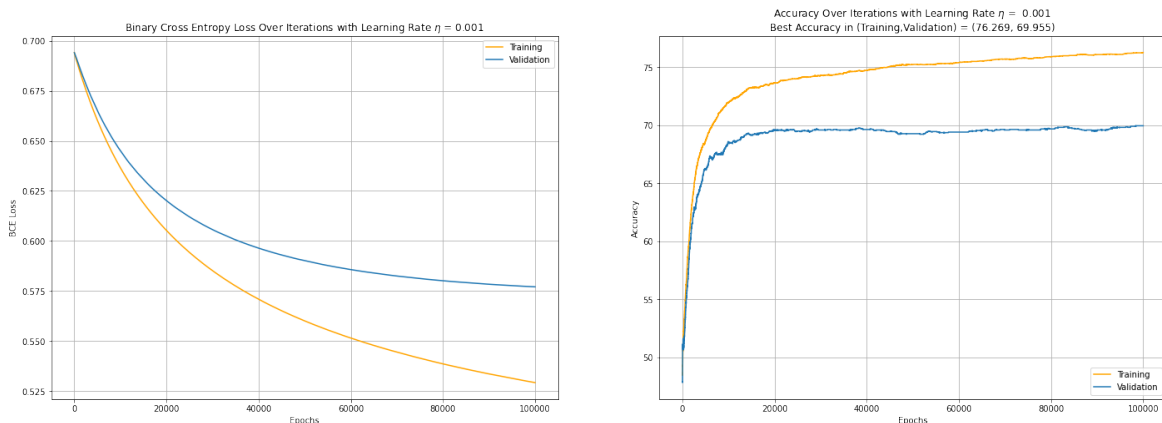


FIGURE 5. The visualization of the training & testing loss and accuracy of Logistic Regression-II

Experiment-II

Time Consumed for fit: 2m 53s

Logistic Regression with hyperparameters (learning rate,batch size,epochs) = ((0.009, 128, 10000))

There are 1028 number of trainable parameters

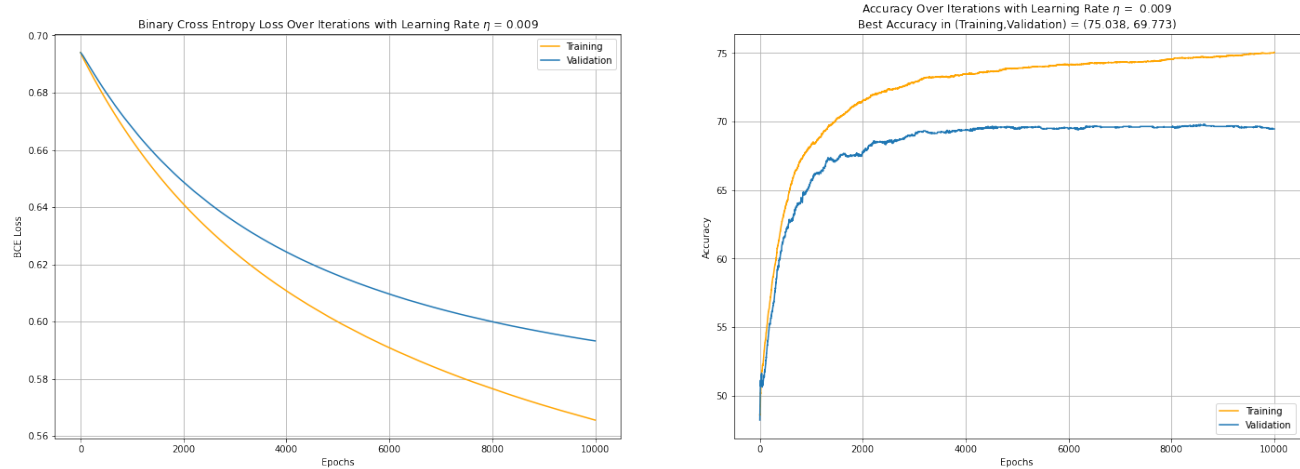


FIGURE 6. The visualization of the training & testing loss and accuracy of Logistic Regression-III

5.5. K-Nearest Neighbors. K-Nearest Neighbor is a simple distance-based supervised ML algorithm. K-Nearest Neighbor works by finding the distances between a query and all the examples in the data, selecting the specified number examples, say K, closest to the query, then votes for the most frequent label in the case of classification or averages the labels in the case of regression [1]. As our task is classification, we take the most frequent label in K nearest neighbors.

We utilized three cartesian space distance metrics as a distance metric: Euclidean, Manhattan, and Cosine. Euclidean distance is the most common one, as it computed the second-order distance between two points.

$$(10) \quad d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \cdots + (x_i - y_i)^2 + \cdots + (x_n - y_n)^2}.$$

In the generic form, it is Minkowski distance with $p = 2$. This distance metric is valid when real-valued vector spaces and the following conditions are satisfied.

- Non-negativity: $d(x, y) \geq 0$
- Identity: $d(x, y) = 0$ if and only if $x = y$
- Symmetry: $d(x, y) = d(y, x)$
- Triangle Inequality: $d(x, y) + d(y, z) \geq d(x, z)$

Then, the Minkowski distance, in general, can be computed as follows.

$$(11) \quad d(x, y) = \left(\sum_{i=1}^n |x_i - y_i|^p \right)^{1/p}.$$

If we set $p = 1$, then the $d(x, y)$ function become Manhattan distance, that is first-order distance metric. It computes the absolute value of the points, whereas Euclidean distance punishes the large distance points in a quadratic manner. Finally, we utilized the cosine distance, and it can be computed by $1 - S_C(A, B)$.

$$(12) \quad \text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \|\mathbf{B}\|} = \frac{\sum_{i=1}^n A_i B_i}{\sqrt{\sum_{i=1}^n A_i^2} \sqrt{\sum_{i=1}^n B_i^2}},$$

We fitted K-Nearest Neighbors to our tweets, and the corresponding distance-neighbor number-accuracy tables are provided below.

Experiment-I

Model	Distance	Accuracy
3-KK	Euclidean	49.99
3-KK	Manhattan	49.99
3-KK	Cosine	50.0

TABLE 4. The accuracies of k-NN with k=3 for various distance metrics

Experiment-II

Model	Distance	Accuracy
5-KK	Euclidean	50.00
5-KK	Manhattan	49.99
5-KK	Cosine	50.00

TABLE 5. The accuracies of k-NN with k=5 for various distance metrics

Experiment-III

Model	Distance	Accuracy
7-KK	Euclidean	50.01
7-KK	Manhattan	50.01
7-KK	Cosine	50.01

TABLE 6. The accuracies of k-NN with k=7 for various distance metrics

As a result, k-NN is the worst model among others in terms of accuracy scores. Moreover, each k-NN's training lasts approximately 3 hours. As our data size is in the order of millions, k-NN was not the scalable-predictive solution for us due to its sequential inference.

6. WORK PACKAGES AND GANTT CHART

We have already implemented more than three algorithms from scratch as the project progresses and will implement more advanced neural network-based algorithms for the final demo. Work packages, their timelines, corresponding team members, and percentage of completeness are provided in the following figures.

TASK NAME	START DATE	DAY OF MONTH*	END DATE	DURATION* (WORK DAYS)	DAYS COMPLETE*	DAYS REMAINING*	TEAM MEMBER	PERCENT COMPLETE
Project Progress								
Preprocessing Tweets	11/1	1	11/5	4	4	0	Can Kocagil	100%
Tweet Data Analysis	11/5	5	11/7	2	2	0	Bariş Kıcıman	100%
Feature Extraction from Tweets	11/7	7	11/11	4	4	0	Can Kocagil	100%
Mutual Information and Feature Selection Implementation	11/11	11	11/13	2	2	0	Can Kocagil	100%
Multinomial Naive Bayes Implementation	11/13	13	11/15				Can Kocagil	100%
Bernoulli Naive Bayes Implementation	11/15	15	11/19	4	4	0	Can Kocagil	100%
Logistic Regression Implementation	11/19	19	11/22	3	3	0	Can Kocagil	100%
k-Nearest Neighbors Implementation	11/22	22	11/24	2	2	0	Can Kocagil	100%
Final Demo								
Multi-layer Perceptron Implementation	11/29	29	12/5	6	3	3	Can Kocagil and Bariş Kıcıman	50%
Deep Neural Network Implementation with Adaptive Optimization Algorithms	12/5	5	12/10	5	1.5	3.5	Can Kocagil	30%
Word2Vec or Word Embedding Implementation	12/10	10	12/18	8	0.8	7.2	Can Kocagil	10%
Pretrained Word Vector Integration (Optional)	12/18	18	12/25	7	0	7	Can Kocagil and Bariş Kıcıman	0%

FIGURE 7. The excel sheet of work packages and gantt chart

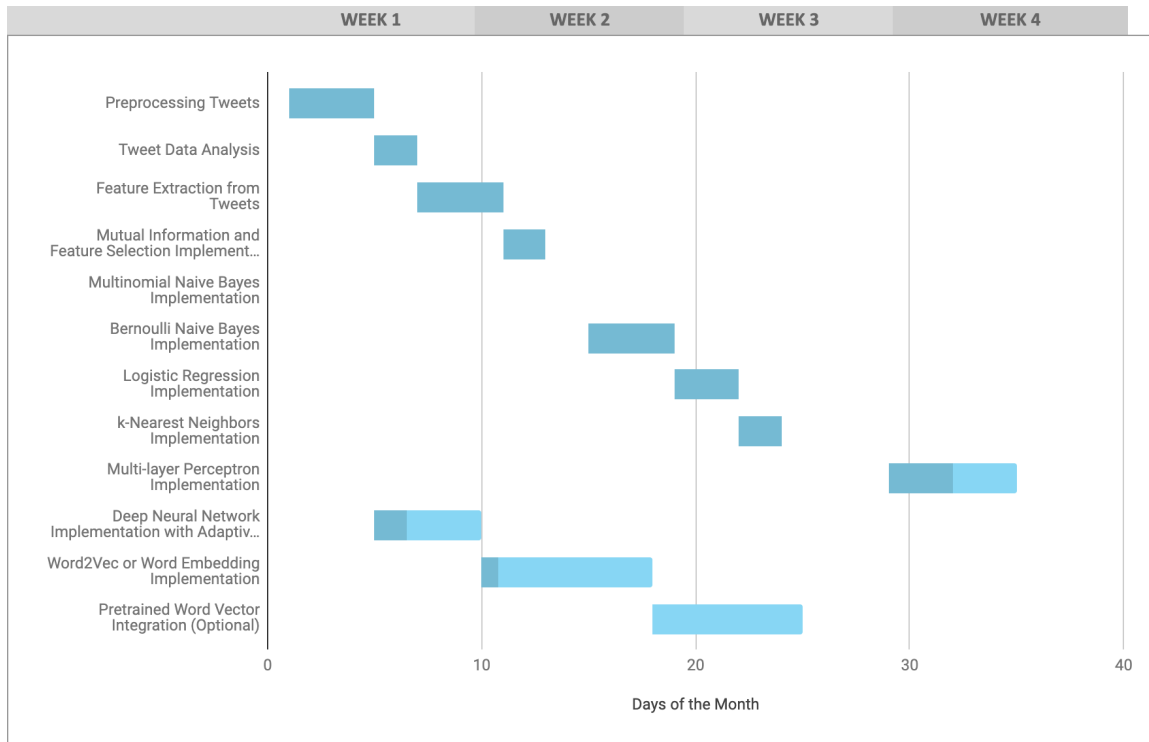


FIGURE 8. The visualization of the work packages and Gantt chart

In the above figure, dark blues represent the completeness of the work package. Light blue regions represent the December month.

Hence, for the final demo, we will implement Multi-layer perceptron, a deep neural network (fully connected) with adaptive optimization algorithms such as AdaGrad and Adam, and Word2Vec, or word embedding. As an optional feature for us, we will implement a neural network integrated with word pre-trained vectors, which can be found in open-source frameworks such as Word2Vec and GloVe.

7. CONCLUSION

By recognizing the significance of autonomous analysis of social media data representing human preferences, daily interactions, and behavioral cognition, we designed machine learning algorithms, from Naive Bayes to Logistic Regression, to capture the social-economical dynamics, extracting sentimental information. After in-depth text preprocessing, we vectorized the tweets and passed them to ML models to learn from them. As behavior-oriented data fuels the cognitive machine learning models, we determined to process the Sentiment140 [3] dataset that consists of 1.6 million social tweets from brands and products to train our models. Our machine learning models' performance has accumulated to around 70% accuracy, and we set that as our baseline. In the final demo, we will try to surpass it with advanced neural networks.

APPENDIX A. CODE

```
1 from __future__ import print_function, division
2
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 import pandas as pd
6 import numpy as np
7 import warnings
8 import random
9 import pickle
10 import json
11 import time
12 import abc
13 import os
14
15 from typing import (
16     Callable,
17     Iterable,
18     List,
19     Union,
20     Tuple,
21 )
22 from collections import OrderedDict
23
24 def mutual_information(
25     x1:np.ndarray,
26     x2:np.ndarray
27 ) -> np.float:
28
29     jh = np.histogram2d(
30         x1,
31         x2,
32         bins = (
33             256, 256
34         )
35     )[0]
36
37     jh = jh + 1e-15
38
39     sh = np.sum(jh)
40     jh = jh / sh
41
42     y1 = np.sum(
43         jh,
44         axis=0
45     ).reshape(
46         (-1, jh.shape[0])
47     )
48
49     y2 = np.sum(
50         jh,
51         axis=1
52     ).reshape(
53         (jh.shape[1], -1)
54     )
55
56
```

```

57     return (
58         np.sum(jh * np.log(jh)) - np.sum(y1 * np.log(y1)) - np.sum(y2 * np.log(y2))
59     )
60
61
62 class BackwardElimination:
63     def __init__(self, pipeline):
64         self.pipe = pipeline
65         self._cache = []
66
67     def fit(
68         self,
69         X_train,
70         y_train,
71         X_test,
72         y_test,
73         feature_subset,
74         verbose=0,
75     ):
76
77         max_iteration_number = len(feature_subset) + 1
78         features = feature_subset.copy()
79         accuracies = [0.0]
80
81
82         for iteration in range(1, max_iteration_number):
83
84
85             candidate_feature = features[iteration - 1]
86
87             if verbose: print(f"Candidate feature to be eliminated is
88                 ↪ {candidate_feature}")
89
90             feature_subset.remove(candidate_feature)
91
92             if verbose: print(f"Remaining features: {feature_subset}")
93
94             since = time.time()
95
96             self.pipe.fit(
97                 X_train = X_train[feature_subset].values,
98                 y_train = y_train.values
99             )
100
101
102             score = self.pipe.score(
103                 X_test = X_test[feature_subset].values,
104                 y_test = y_test.values
105             )
106
107             time_passed = time.time() - since
108             accuracies.append(
109                 score['accuracy']
110             )
111
112             self._cache.append(
113                 (
114                     "-" . join(

```

```

115         feature_subset
116     ),
117     score['accuracy'],
118     round(float(time_passed), 4)
119 )
120 )
121
122
123     if accuracies[iteration] < accuracies[iteration - 1]:
124         feature_subset.append(candidate_feature)
125
126     if verbose: print(f"Candidate feature is restored :
127                  ↪ {candidate_feature}")
128
129
130     self._best_features = feature_subset
131     self._scores = accuracies
132
133     return self
134
135     def best_features(self):
136         return self._best_features
137
138     def cache(self):
139         return self._cache
140
141     def scores(self):
142         return self._scores
143
144 from __future__ import (
145     print_function,
146     division
147 )
148
149 import matplotlib.pyplot as plt
150 import seaborn as sns
151 import pandas as pd
152 import numpy as np
153 import warnings
154 import random
155 import pickle
156 import json
157 import time
158 import abc
159 import os
160
161 from collections import OrderedDict
162
163 from typing import (
164     Callable,
165     Iterable,
166     List,
167     Union,
168     Tuple,
169 )
170
171 from utils import (
172     Classifier,

```

```

173 Pipeline,
174 Vocabulary,
175 json_print,
176 timeit,
177 random_seed
178 )
179
180
181 class KNeighborsClassifier(Classifier):
182     """
183     K-NeighborsClassifier based on geometric distance metrics
184
185     """
186     def __init__(
187         self,
188         k_neighbors:int = 9,
189         distance_metric:str = "euclidean"
190     ):
191         """
192         Args:
193
194             k_neighbors: int
195                 - Number of Neighbors (Default = 9)
196             distance_metric: str
197                 - Distance metric (Default = "euclidean")
198                 - Available metrics = [euclidean, manhattan, cosine]
199         """
200         super(KNeighborsClassifier, self).__init__()
201
202         self.k_neighbors = k_neighbors
203         self.distance_metric = distance_metric
204
205
206         self._hyperparams['k_neighbors'] = self.k_neighbors
207         self._hyperparams['distance_metric'] = self.distance_metric
208         self._name = 'K-Nearest Neighbors'
209
210     def fit(self, X_train, y_train, *fit_params):
211         self.X_train, self.y_train = X_train, y_train
212
213         return self
214
215     def euclidean_distance(self, x1, x2):
216         return np.sqrt(
217             np.einsum(
218                 'ij,ij->i...',
219                 x1 - x2,
220                 x1 - x2
221             )
222         )
223
224     def manhattan_distance(self, x1, x2):
225         return np.linalg.norm(
226             x1 - x2,
227             axis = 1,
228             ord = 1
229         )
230
231     def cosine_distance(self, x1, x2):

```

```

232     y = np.einsum(
233         'ij,ij->i',
234         x2,
235         x2
236     )
237     x = np.einsum(
238         'ij,ij->i',
239         x1,
240         x1
241     )[:, np.newaxis]
242
243     sumxy = x1 @ x2.T
244     return 1 - (
245         sumxy / np.sqrt(x)
246     ) / np.sqrt(y)
247
248
249 def predict(self, X_test):
250
251     if self.distance_metric == "euclidean":
252         distances = np.array([
253             self.euclidean_distance(x_test, self.X_train) for x_test in X_test
254         ])
255
256     elif self.distance_metric == "manhattan":
257         distances = np.array([
258             self.manhattan_distance(x_test, self.X_train) for x_test in X_test
259         ])
260
261     elif self.distance_metric == "cosine":
262         distances = self.cosine_distance(
263             X_test,
264             self.X_train
265         )
266
267
268     sorted_neighbors = distances.argsort(axis=1)[..., : self.k_neighbors]
269     nearest_labels = self.y_train[sorted_neighbors]
270
271     predictions = np.apply_along_axis(
272         lambda x: np.bincount(x).argmax(),
273         axis = 1,
274         arr = nearest_labels
275     )
276
277     return predictions
278
279 class MultiNominalNaiveBayes(Classifier):
280     def __init__(self, alpha=0.0001):
281         super(MultiNominalNaiveBayes, self).__init__()
282         self.alpha = alpha
283
284         self._hyperparams['alpha'] = self.alpha
285         self._name = 'MultiNominal NaiveBayes Classifier'
286
287     @timeit
288     def fit(
289         self,
290         X_train:pd.DataFrame,

```

```

291     y_train: pd.DataFrame,
292     **fit_params
293 ):
294
295     m, n = X_train.shape
296     self.classes = np.unique(y_train)
297     n_classes = len(self.classes)
298
299     if not isinstance(X_train, pd.DataFrame):
300         X_train = pd.DataFrame(X_train)
301
302     self.priors = y_train.value_counts(normalize = True).values
303     self.counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
304     likelihoods = self.counts.T / self.counts.sum(1).values.reshape(-1, n_classes)
305     ↪ + self.alpha
306     self.likelihoods = likelihoods.values #.T
307     self.log_priors = np.log(self.priors)
308
309     return self
310
311 @timeit
312 def predict(self, X_test):
313
314     if isinstance(X_test, pd.DataFrame):
315         X_test = X_test.values
316
317     self.log_likelihoods = X_test @ np.log(self.likelihoods)
318     ↪ #(np.log(self.likelihoods) @ X_test.T).T
319     self.posterioriors = self.log_likelihoods + self.log_priors
320
321     return self.classes[
322         self.posterioriors.argmax(1)
323     ]
324
325 def posteriors(self):
326     return self.posterioriors
327
328 class BernaulliNaiveBayes(Classifier):
329     def __init__(self, alpha = 0.001):
330         super(BernaulliNaiveBayes, self).__init__()
331         self.alpha = alpha
332
333         self._hyperparams['alpha'] = self.alpha
334         self._name = 'Bernaulli NaiveBayes Classifier'
335
336 @timeit
337 def fit(self, X_train, y_train, **fit_params):
338     self.classes = np.unique(y_train)
339
340     n_classes = len(self.classes)
341
342     if not isinstance(X_train, pd.DataFrame):
343         X_train = pd.DataFrame(X_train)
344
345     self.priors = y_train.value_counts(normalize = True).values
346     self.log_priors = np.log(self.priors)
347
348     counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')

```

```

347     likelihoods = counts.T / counts.sum(1).values.reshape(-1, n_classes) +
        ↪ self.alpha
348     self.likelihoods = likelihoods.T.values
349
350     return self
351
352 @timeit
353 def predict(self, X_test):
354
355     self.posterioriors = np.array(
356         [
357             (
358                 (np.log(self.likelihoods) * x) + (np.log(1 - self.likelihoods) *
        ↪ np.abs(x - 1))
359             ).sum(axis = 1) + self.log_priors for x in X_test
360         ]
361     )
362
363     return self.classes[
364         self.posterioriors.argmax(1)
365     ]
366
367 class LogisticRegression(Classifier):
368     def __init__(self):
369         super().__init__()
370
371     def init_params(self,
372         input_shape:int,
373         output_shape:int = 1
374     ):
375         self.__random_seed()
376
377         #assert self.X_train.shape[1] == self.X_test.shape[1], 'Improper feature
        ↪ dimension!'
378
379         W_high = self.__init_xavier(input_shape, output_shape)
380         W_low = - W_high
381         W_size = (input_shape, output_shape)
382         B_size = (1, output_shape)
383
384         self.W = np.random.uniform(
385             W_low,
386             W_high,
387             size = W_size
388         )
389
390         self.b = np.random.uniform(
391             W_low,
392             W_high,
393             size = B_size
394         )
395
396
397     def __random_seed(self, seed = 32):
398         """ Random seed for reproducibility """
399         random.seed(seed)
400         np.random.seed(seed)
401
402

```

```

403 def __init_xavier(self, L_pre, L_post):
404     """ Given the size of the input node and hidden node, initialize the weights
        ↳ drawn from uniform distribution ~ Uniform[- sqrt(6/(L_pre + L_post)) ,
        ↳ sqrt(6/(L_pre + L_post))] """
405     return np.sqrt(6/(L_pre + L_post))
406
407 def __train_config(self,
408     lr:float,
409     batch_size:int,
410     epochs:int,
411 ):
412     self.lr = lr
413     self.batch_size = batch_size
414     self.epochs = epochs
415
416 def sigmoid(self,X, grad = False):
417     """ Computing sigmoid and it's gradient w.r.t. it's input """
418     sig = 1/(1 + np.exp(-X))
419
420     return sig * (1-sig) if grad else sig
421
422 def __forward(self, X):
423
424     Z = (X @ self.W) + self.b
425     A = self.sigmoid(Z)
426
427     return {
428         "Z": Z,
429         "A": A
430     }
431
432
433 def __SGD(self, grads):
434     self.W -= self.lr * grads['W']
435     self.b -= self.lr * grads['b']
436
437
438 def matrix_back_prop(self, outs, X, Y):
439     """ Matrix form backward propagation """
440     m = self.batch_size
441
442     Z = outs['Z']
443     A = outs['A']
444
445     dZ = (A-Y) * self.sigmoid(Z, grad = True)
446     dW = (1 / m) * (X.T @ dZ)
447     db = (1 / m) * np.sum(dZ, axis=0, keepdims=True)
448
449     assert self.W.shape == dW.shape, f'Error in weight shapes!, {dW.shape} does not
        ↳ match with {self.W.shape}'
450     assert self.b.shape == db.shape, f'Error in bias shapes!, {db.shape} does not
        ↳ match with {self.b.shape}'
451
452     grads = {}
453     grads['W'] = dW
454     grads['b'] = db
455
456     return grads
457

```



```

458
459     def backward(self,
460                 outs,
461                 X,
462                 Y
463             ):
464         return self.matrix_back_prop(
465             outs,
466             X,
467             Y
468         )
469
470
471     def BinaryCrossEntropyLoss(self, pred, label):
472         m = pred.shape[0]
473         preds = np.clip(pred, 1e-16, 1 - 1e-16)
474         loss = np.sum(-label * np.log(preds + 1e-20) - (1 - label) * np.log(1 - preds +
475             ↪ 1e-20))
476         return loss / m
477
478     def eval(self, x, y, knob:float = 0.5):
479         predictions = self.__forward(x)
480         predictions = predictions['A']
481         predictions[predictions >= knob] = 1
482         predictions[predictions < knob] = 0
483         acc_score = self.accuracy(predictions, y)
484
485         return acc_score
486
487     def __accuracy(self, pred, label):
488         return np.sum(pred == label) / pred.shape[0]
489
490
491     @timeit
492     def fit(
493         self,
494         X_train,
495         y_train,
496         X_test,
497         y_test,
498         lr:float = 1e-2,
499         batch_size:int = 32,
500         epochs:int = 100,
501         verbose = True
502     ):
503         """
504         Given the training dataset, their labels and number of epochs
505         fitting the model, and measure the performance
506         by validating training dataset.
507         """
508
509         self.init_params(
510             input_shape = X_train.shape[1]
511         )
512
513         self.__train_config(
514             lr,
515             batch_size,
516             epochs,

```

```

516     )
517
518     self.history = {}
519
520     self.history['train'] = {
521         'loss': [],
522         'acc' : []
523     }
524
525     self.history['val'] = {
526         'loss': [],
527         'acc' : []
528     }
529
530     m = self.batch_size
531
532     self.sample_size_train = X_train.shape[0]
533
534     for epoch in range(self.epochs):
535
536         perm = np.random.permutation(self.sample_size_train)
537
538         for i in range(self.sample_size_train // m):
539
540
541             shuffled_index = perm[i*m: (i+1)*m]
542
543             X_feed = X_train[shuffled_index]
544             y_feed = y_train[shuffled_index]
545
546             outs = self.__forward(X_feed)
547             grads = self.backward(
548                 outs,
549                 X_feed,
550                 y_feed
551             )
552             self.__SGD(grads)
553
554             loss_train = self.BinaryCrossEntropyLoss(
555                 self.__forward(X_train)[:, 'A'],
556                 y_train
557             )
558
559             acc_train = self.eval(
560                 X_train,
561                 y_train
562             )
563
564             self.history['train']['loss'].append(loss_train)
565             self.history['train']['acc'].append(acc_train)
566
567
568
569             loss_val = self.BinaryCrossEntropyLoss(
570                 self.__forward(X_test)[:, 'A'],
571                 y_test
572             )
573
574             acc_val = self.eval(

```

```

575         X_test,
576         y_test
577     )
578
579     self.history['val']['loss'].append(loss_val)
580     self.history['val']['acc'].append(acc_val)
581
582     if verbose:
583         print(f"[{epoch}/{self.epochs}] -----> Training : BCE: {loss_train} and
584             ↳ Acc: {acc_train}")
585         print(f"[{epoch}/{self.epochs}] -----> Testing : BCE: {loss_val} and
586             ↳ Acc: {acc_val}")
587
588     def __str__(self):
589         model = LogisticRegression().__class__.__name__
590         model += f' with hyperparameters (learning rate, batch_size, epochs) =
591             ↳ ({self.lr, self.batch_size, self.epochs})'
592         num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] *
593             ↳ self.b.shape[1]
594         model += f'\n There are {num_params} number of trainable parameters'
595         return model
596
597     def __repr__(self):
598         model = LogisticRegression().__class__.__name__
599         model += f' with hyperparameters (learning rate, batch_size, epochs) =
600             ↳ ({self.lr, self.batch_size, self.epochs})'
601         num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] *
602             ↳ self.b.shape[1]
603         model += f'\n There are {num_params} number of trainable parameters'
604         return model
605
606     def plot_history(self):
607
608         fig, axes = plt.subplots(1, 2, figsize = (24, 8))
609         axes[0].plot(self.history['train']['loss'], color = 'orange', label = 'Training')
610         axes[0].plot(self.history['val']['loss'], label = 'Validation')
611         axes[0].set_xlabel('Epochs')
612         axes[0].set_ylabel('BCE Loss')
613         axes[0].set_title(f'Binary Cross Entropy Loss Over Iterations with Learning Rate
614             ↳  $\eta$  = {self.lr}')
615         axes[0].legend(loc="upper right")
616         axes[0].grid()
617
618         axes[1].plot(self.history['train']['acc'], color = 'orange', label = 'Training')
619         axes[1].plot(self.history['val']['acc'], label = 'Validation')
620         axes[1].set_xlabel('Epochs')
621         axes[1].set_ylabel('Accuracy')
622         maxs = round(max(self.history['train']['acc']), 3),
623             ↳ round(max(self.history['val']['acc']), 3)
624         axes[1].set_title(f'Accuracy Over Iterations with Learning Rate  $\eta$  =
625             ↳ {self.lr} \n Best Accuracy in (Training, Validation) = {maxs} ')
626         axes[1].legend(loc="lower right")
627         axes[1].grid()
628
629     class MLP(Classifier):

```

```

624     def __init__(self,
625         input_size = X_train.shape,
626         batch_size = 19 ,
627         n_neurons = 76 ,
628         mean = 0,
629         std = 1,
630         lr = 1e-1,
631         distribution = 'Xavier'
632     ):
633
634
635         np.random.seed(15)
636         self.lr = lr
637         self.mse_train = {}
638         self.mce_train = {}
639         self.mse_test = {}
640         self.mce_test = {}
641
642         self.sample_size = input_size[0]
643         self.feature_size = input_size[1]
644         self.batch_size = batch_size
645         self.n_neurons = n_neurons
646         self.mean, self.std = mean, std
647
648         self.dist = distribution
649
650
651         self.n_update = round((self.sample_size/self.batch_size))
652
653         self.W1_size = self.feature_size, self.n_neurons
654         self.W2_size = self.n_neurons, 1
655
656         self.B1_size = 1, self.n_neurons
657         self.B2_size = 1, 1
658
659         self.B1 = np.random.normal(loc = self.mean, scale = self.std, size =
660             ↳ (self.B1_size)) * 0.01
661         self.B2 = np.random.normal(loc = self.mean, scale = self.std, size =
662             ↳ (self.B2_size)) * 0.01
663
664         self.he_scale1 = np.sqrt(2/self.feature_size)
665         self.he_scale2 = np.sqrt(2/self.n_neurons)
666         self.xavier_scale1 = np.sqrt(2/(self.feature_size+self.n_neurons))
667         self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))
668
669         if (self.dist == 'Zero') :
670             self.W1 = np.zeros((self.W1_size))
671             self.W2 = np.zeros((self.W2_size))
672
673         elif (self.dist == 'Gauss'):
674             self.W1 = np.random.normal(loc = self.mean, scale = self.std, size =
675                 ↳ (self.W1_size))* 0.01
676             self.W2 = np.random.normal(loc = self.mean, scale = self.std, size =
677                 ↳ (self.W2_size))* 0.01
678
679         elif (self.dist == 'He'):
680             self.W1 = np.random.randn(self.W1_size[0], self.W1_size[1]) * self.he_scale1
681             self.W2 = np.random.randn(self.W2_size[0], self.W2_size[1]) * self.he_scale2

```

```

679     elif (self.dist == 'Xavier'):
680
681         self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) *
        ↪ self.xavier_scale1
682         self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) *
        ↪ self.xavier_scale2
683
684
685
686     def forward(self,X):
687
688         Z1 = (X @ self.W1) + self.B1
689         A1 = np.tanh(Z1)
690         Z2 = (A1 @ self.W2) + self.B2
691         A2 = np.tanh(Z2)
692
693         return {
694             "Z1": Z1,
695             "A1": A1,
696             "Z2": Z2,
697             "A2": A2
698         }
699
700
701     def tanh(self,X):
702         return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))
703
704     def tanh_der(self,X):
705         return 1-(np.tanh(X)**2)
706
707     def backward(self,outs, X, Y):
708         m = (self.batch_size)
709
710         Z1 = outs['Z1']
711         A1 = outs['A1']
712         Z2 = outs['Z2']
713         A2 = outs['A2']
714
715         dZ2 = (A2-Y)* self.tanh_der(Z2)
716         dW2 = (1/m) * (A1.T @ dZ2)
717         dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)
718
719         dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
720         dW1 = (1/m) * (X.T @ dZ1)
721         dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)
722
723
724         return {
725             "dW1": dW1,
726             "dW2": dW2,
727             "dB1": dB1,
728             "dB2": dB2
729         }
730
731     def Loss(self,pred, y_true, knob = 0):
732
733         mse = np.square(pred-y_true).mean()
734
735         pred[pred>=knob]=1

```

```

736     pred[pred<knob]==-1
737
738     mce = (pred == y_true).mean()
739
740     return {
741         'MSE':mse,
742         'MCE':mce
743     }
744
745
746     def SGD(self,grads):
747         self.W1 -= self.lr * grads['dW1']
748         self.W2 -= self.lr * grads['dW2']
749         self.B1 -= self.lr * grads['dB1']
750         self.B2 -= self.lr * grads['dB2']
751
752     def fit(self,X,Y,X_test,y_test,epochs = 300,verbose=True):
753         """
754         Given the traning dataset,their labels and number of epochs
755         fitting the model, and measure the performance
756         by validating training dataset.
757         """
758
759         m = self.batch_size
760
761         for epoch in range(epochs):
762             perm = np.random.permutation(self.sample_size)
763
764             for i in range(self.n_update):
765
766
767                 batch_start = i * m
768                 batch_finish = (i+1) * m
769                 index = perm[batch_start:batch_finish]
770
771                 X_feed = X[index]
772                 y_feed = Y[index]
773
774
775                 outs = self.forward(X_feed)
776                 loss = self.Loss(
777                     outs['A2'],
778                     y_feed
779                 )
780
781                 outs_test = self.forward(X_test)
782                 loss_test = self.Loss(
783                     outs_test['A2'],
784                     y_test
785                 )
786
787                 grads = self.backward(
788                     outs,
789                     X_feed,
790                     y_feed
791                 )
792
793                 self.SGD(grads)
794

```

```

795         self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
796         self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
797         self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
798         self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']
799
800         if verbose:
801             print(f"[{epoch}/{epochs}] -----> Training :MSE: {loss['MSE']} and MCE:
802                 ↳ {loss['MCE']}")
803             print(f"[{epoch}/{epochs}] -----> Testing :MSE: {loss_test['MSE']} and
804                 ↳ MCE: {loss_test['MCE']}")
805
806     def history(self):
807         return {
808             'Train_MSE' : self.mse_train,
809             'Train_MCE' : self.mce_train,
810             'Test_MSE' : self.mse_test,
811             'Test_MCE' : self.mce_test
812         }
813
814 from __future__ import print_function, division
815
816 import matplotlib.pyplot as plt
817 import seaborn as sns
818 import pandas as pd
819 import numpy as np
820 import warnings
821 import random
822 import pickle
823 import json
824 import time
825 import abc
826 import os
827
828 from typing import (
829     Callable,
830     Iterable,
831     List,
832     Union,
833     Tuple,
834 )
835
836 from collections import OrderedDict
837
838 def save_obj(
839     obj:object,
840     path:str = None
841 ) -> None:
842     """ Saves Python Object as pickle"""
843     with open(path + '.pkl', 'wb') as f:
844         pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)
845
846 def load_obj(
847     path:str = None
848 ) -> object:
849     """ Loads Python Object from pickle"""
850     with open(path + '.pkl', 'rb') as f:
851         return pickle.load(f)

```

```

852 class ClassifierCharacteristics:
853
854     def confusion_matrix(
855         self,
856         preds: Iterable[list or np.ndarray],
857         labels: Iterable[list or np.ndarray]
858     ) -> pd.DataFrame:
859
860         """Given the labels and predictions, calculate confusion matrix. """
861         label = pd.Series(
862             labels,
863             name = 'Actual'
864         )
865         pred = pd.Series(
866             preds,
867             name = 'Predicted'
868         )
869
870         return pd.crosstab(
871             label,
872             pred
873         )
874
875     def accuracy(
876         self,
877         preds: Iterable[list or np.ndarray],
878         labels: Iterable[list or np.ndarray],
879         scale:bool = True
880     ) -> np.float:
881         """Given the labels and predictions, calculate accuracy score. """
882         return np.mean(preds == labels) * 100 if scale else np.mean(preds == labels)
883
884     def visualize_confusion_matrix(
885         self,
886         data:Iterable[list or np.ndarray],
887         normalize:bool = True,
888         title:str = " ",
889     ) -> None:
890
891         if normalize:
892             data /= np.sum(data)
893
894         plt.figure(
895             figsize = (
896                 15, 15
897             )
898         )
899         sns.heatmap(
900             data,
901             fmt='.2%',
902             cmap = 'Greens'
903         )
904
905         plt.title(title)
906         plt.show()
907
908     @staticmethod
909     def timeit(
910         Func:Callable

```



```

911 ):
912     """ Calculate time spend of the function
913
914     Usage:
915         >> @timeit
916         >> def func(x):
917         >>     return x
918     """
919     def _timeStamp(*args, **kwargs):
920         since = time.time()
921         result = Func(*args, **kwargs)
922         time_elapsed = time.time() - since
923
924         if time_elapsed > 60:
925             print('Time Consumed : {:.0f}m {:.0f}s'.format(time_elapsed // 60,
926                 ↪ time_elapsed % 60))
927         else:
928             print('Time Consumed : ' , round((time_elapsed), 4) , 's')
929         return result
930     return _timeStamp
931
932 @staticmethod
933 def random_seed(
934     Func:Callable,
935     seed:int = 42
936 ):
937     """
938     Decorator random seed.
939
940     Usage:
941         >> @random_seed
942         >> def func(*args):
943         >>     return [arg for arg in args]
944     """
945     def _random_seed(*args, **kwargs):
946         np.random.seed(seed)
947         random.seed(seed)
948         result = Func(
949             *args,
950             **kwargs
951         )
952         return result
953     return _random_seed
954
955 def save_obj(
956     self,
957     obj:object,
958     path:str = None
959 ) -> None:
960     """ Saves Python Object as pickle"""
961     with open(path + '.pkl', 'wb') as f:
962         pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)
963
964 def load_obj(
965     self,
966     path:str = None
967 ) -> object:
968     """ Loads Python Object from pickle"""

```

```

969         with open(path + '.pkl', 'rb') as f:
970             return pickle.load(f)
971
972
973     def save_numpy(
974         self,
975         data: Iterable[list or np.ndarray] = None,
976         path: str = None
977     ) -> None:
978         """ Saves NumPy array or Python object as .npy """
979         np.save(
980             path + '.npy',
981             data,
982             allow_pickle=True
983         )
984
985     def load_numpy(
986         self,
987         path: str = None
988     ) -> np.ndarray:
989         """ Loads NumPy array or Python object as .npy """
990         return np.load(
991             path + '.npy',
992             allow_pickle=True
993         )
994
995     class Classifier(ClassifierCharacteristics):
996         #__metaclass__ = abc.ABCMeta
997
998         def __init__(self):
999             super(Classifier, self).__init__()
1000             self._hyperparams = {}
1001             self._scores = {}
1002             self._name = ""
1003             self._params = OrderedDict()
1004
1005
1006         def save(self, filename: str) -> None:
1007             self.save_obj(
1008                 self.__dict__,
1009                 filename
1010             )
1011
1012         def load(self, filename: str) -> None:
1013             self.__dict__ = self.load_obj(filename)
1014
1015
1016         @ClassifierCharacteristics.timeit
1017         def fit(
1018             self,
1019             X_train: Union[np.ndarray, pd.DataFrame],
1020             y_train: Union[np.ndarray, pd.DataFrame],
1021             *fit_params
1022         ) -> None:
1023             return NotImplementedError()
1024
1025
1026         @ClassifierCharacteristics.timeit

```

```

1028     def predict(
1029         self,
1030         X_test: Union[np.ndarray, pd.DataFrame]
1031     ) -> Union[pd.DataFrame, np.ndarray]:
1032         return NotImplementedError()
1033
1034
1035
1036     @ClassifierCharacteristics.timeit
1037     def fit_predict(
1038         self,
1039         X_train: Union[np.ndarray, pd.DataFrame],
1040         y_train: Union[np.ndarray, pd.DataFrame],
1041         X_test: Union[np.ndarray, pd.DataFrame],
1042         *fit_params
1043     ) -> Union[pd.DataFrame, np.ndarray]:
1044
1045         self.fit(X_train, y_train, *fit_params)
1046
1047         return self.predict(X_test)
1048
1049     def score(
1050         self,
1051         X_test: Union[np.ndarray, pd.DataFrame],
1052         y_test: Union[np.ndarray, pd.DataFrame],
1053         metric_list: List[Callable] = []
1054     ) -> np.float:
1055
1056         predictions = self.predict(X_test)
1057         accuracy = self.accuracy(predictions, y_test)
1058
1059
1060         self._scores['accuracy'] = accuracy
1061
1062         if len(metric_list) != 0:
1063             for metric in metric_list:
1064                 self._scores[metric.__name__] = metric(predictions, y_test)
1065
1066         return self._scores
1067
1068
1069     def params(self) -> OrderedDict:
1070         return self._params
1071
1072
1073     def hyperparameters(self):
1074         return self._hyperparams
1075
1076
1077     def name(self) -> str:
1078         return self._name
1079
1080
1081     def __str__(self) -> str:
1082         return f"{self._name} with hyperparameters {json.dumps(self._hyperparams,
1083             ↪ sort_keys=True, indent=4)}"
1084
1085     def __repr__(self) -> str:

```

```

1086         return f"{self._name} with hyperparameters {json.dumps(self._hyperparams,
1087                               ↪ sort_keys=True, indent=4)}"
1088
1089     class Pipeline:
1090         """ Generic ML Operation Pipeline """
1091         def __init__(
1092             self,
1093             pipeline: List[Tuple[str, object]] = []
1094         ):
1095             super(Pipeline, self).__init__()
1096             self.pipeline = pipeline
1097             self.model = None
1098             self._name = 'ML Pipeline'
1099             self._scores = {}
1100
1101         def fit(
1102             self,
1103             X_train: Union[np.ndarray, pd.DataFrame],
1104             y_train: Union[np.ndarray, pd.DataFrame],
1105             verbose: int = 0,
1106             *fit_params
1107         ) -> None:
1108
1109             yield_data = X_train.copy()
1110
1111             for pipe_name, pipe_op in self.pipeline:
1112
1113                 if verbose: print(f"{pipe_name} operation is applying")
1114
1115                 if issubclass(pipe_op.__class__, Classifier):
1116
1117                     self.model = pipe_op
1118                     self.model.fit(
1119                         yield_data,
1120                         y_train,
1121                         *fit_params
1122                     )
1123
1124                 elif isinstance(pipe_op.__class__, FeatureEngineer):
1125                     yield_data = pipe_op.fit_transform(yield_data)
1126
1127                 else:
1128                     raise Exception(f"{pipe_name} operator could not be decoded!")
1129
1130
1131         def predict(
1132             self,
1133             X_test: Union[np.ndarray, pd.DataFrame]
1134         ) -> Union[pd.DataFrame, np.ndarray]:
1135             return self.model.predict(X_test)
1136
1137
1138         def score(
1139             self,
1140             X_test: Union[np.ndarray, pd.DataFrame],
1141             y_test: Union[np.ndarray, pd.DataFrame],
1142             metric_list: List[Callable] = []
1143         ) -> np.float:

```

```

1144
1145     predictions = self.model.predict(X_test)
1146     accuracy = self.model.accuracy(predictions, y_test)
1147
1148
1149     self._scores['accuracy'] = accuracy
1150
1151     if len(metric_list) != 0:
1152         for metric in metric_list:
1153             self._scores[metric.__name__] = metric(predictions, y_test)
1154
1155     return self._scores
1156
1157     def name(self) -> str:
1158         return self._name
1159
1160
1161     def __str__(self) -> str:
1162         return "\n".join([
1163             str(pape_op) for _, pape_op in self.pipeline
1164         ])
1165     def __repr__(self) -> str:
1166         return "\n".join([
1167             str(pape_op) for _, pape_op in self.pipeline
1168         ])
1169
1170
1171
1172 class Vocabulary:
1173     def __init__(
1174         self,
1175         root_dir:str,
1176         filename:str,
1177         delimiter: str = '\n'
1178     ):
1179         super(Vocabulary, self).__init__()
1180         self.vocab = open(
1181             os.path.join(
1182                 root_dir,
1183                 filename
1184             )
1185         ).read()
1186
1187         self.list_vocab = self.vocab.split(delimiter)[: -1]
1188
1189         self.word2id = {
1190             word: i for i, word in enumerate(self.list_vocab)
1191         }
1192
1193         self.id2word = {
1194             i: word for word, i in self.word2id.items()
1195         }
1196
1197     def __getitem__(self, idx):
1198
1199         if isinstance(idx, (list, np.ndarray)):
1200             return [self.id2word[i] for i in idx]
1201
1202

```

```

1203         return self.id2word[idx]
1204
1205     def __len__(self):
1206         return len(self.list_vocab)
1207
1208     def get_vocab(self):
1209         return self.word2id
1210
1211
1212 def json_print(data):
1213     return json.dumps(
1214         data,
1215         sort_keys=True,
1216         indent=4
1217     )
1218
1219
1220
1221 def timeit(
1222     Func:Callable
1223 ):
1224     """ Calculate time spend of the function
1225
1226     Usage:
1227     >> @timeit
1228     >> def func(x):
1229     >>     return x
1230     """
1231
1232     def _timeStamp(*args, **kwargs):
1233         since = time.time()
1234         result = Func(*args, **kwargs)
1235         time_elapsed = time.time() - since
1236
1237         if time_elapsed > 60:
1238             print('Time Consumed for {}: {:.0f}m {:.0f}s'.format(Func.__name__,
1239                 ↪ time_elapsed // 60, time_elapsed % 60))
1240         else:
1241             print(f'Time Consumed for {Func.__name__}: {round((time_elapsed), 4)} s')
1242         return result
1243     return _timeStamp
1244
1245 def random_seed(
1246     Func:Callable,
1247     seed:int = 42
1248 ):
1249     """
1250     Decorator random seed.
1251
1252     Usage:
1253     >> @random_seed
1254     >> def func(*args):
1255     >>     return [arg for arg in args]
1256     """
1257
1258     def _random_seed(*args, **kwargs):
1259         np.random.seed(seed)
1260         random.seed(seed)
1261         result = Func(
1262             *args,
1263             **kwargs

```

```

1261         )
1262         return result
1263     return _random_seed
1264
1265     # %%
1266     from __future__ import (
1267         print_function,
1268         division
1269     )
1270
1271     import matplotlib.pyplot as plt
1272     import stopwordsiso as swiso
1273     import seaborn as sns
1274     import pandas as pd
1275     import numpy as np
1276     import cleantext
1277     import warnings
1278     import random
1279     import string
1280     import pickle
1281     import spacy
1282     import json
1283     import nltk
1284     import time
1285     import abc
1286     import os
1287     import re
1288     import sys
1289     sys.path.append('./src')
1290
1291     from spellchecker import SpellChecker
1292     from stop_words import get_stop_words
1293     from collections import OrderedDict
1294     from nltk.stem import PorterStemmer
1295     from collections import Counter
1296     from nltk.corpus import stopwords
1297     from wordcloud import WordCloud
1298
1299
1300     from textblob import (
1301         TextBlob,
1302         Word
1303     )
1304
1305     from typing import (
1306         Callable,
1307         Iterable,
1308         List,
1309         Union,
1310         Tuple,
1311     )
1312
1313     from utils import (
1314         Classifier,
1315         Pipeline,
1316         Vocabulary,
1317         json_print,
1318         timeit,
1319         random_seed

```

```

1320 )
1321 from supervised import (
1322     KNeighborsClassifier,
1323     MultiNominalNaiveBayes,
1324     BernaulliNaiveBayes
1325 )
1326 from feature import (
1327     BackwardElimination,
1328     #mutual_information
1329 )
1330
1331
1332
1333 # %%
1334 cols = [
1335     'date',
1336     'id',
1337     'text',
1338     'query_string',
1339     'user',
1340     'sentiment',
1341 ]
1342
1343 df = pd.read_csv(
1344     '../data/training.1600000.processed.noemoticon.csv',
1345     encoding= 'latin1',
1346     names = [
1347         'sentiment',
1348         'id',
1349         'date',
1350         'query_string',
1351         'user',
1352         'text'
1353     ]
1354 )
1355
1356 df = df[cols]
1357
1358 # %%
1359 df.head()
1360
1361 # %%
1362 df.info()
1363
1364 # %%
1365 df.set_index('id', inplace=True)
1366 df.drop(
1367     columns= [
1368         'date', 'query_string'
1369     ],
1370     axis=1,
1371     inplace=True
1372 )
1373
1374 # %%
1375 df.head(7)
1376
1377 # %%
1378 df['sentiment'].value_counts(

```



```

1379     normalize = True
1380     ).plot.bar(
1381         title = 'Sentiment Distribution',
1382         xlabel = ['Neg', 'Pos']
1383     )
1384
1385     # %%
1386     print(f"Max length of the tweet {df.text.str.len().max()}")
1387     print(f"Min length of the tweet {df.text.str.len().min()}")
1388     print(f"Average length of the tweet {df.text.str.len().mean()}")
1389     print(f"Median length of the tweet {df.text.str.len().median()}")
1390
1391     # %%
1392     len_stats = pd.DataFrame(df.text.str.len().describe())
1393     len_stats.col = 'tweet_len_stats'
1394     len_stats
1395
1396     # %%
1397     df['sentiment'] = df['sentiment'].replace(
1398         {
1399             4:1
1400         }
1401     )
1402
1403     df['sentiment'].sample(5)
1404
1405     # %%
1406     class Cleaner:
1407         def __init__(self, operations = []):
1408             self.operations = operations
1409
1410         def __call__(self, text):
1411             for operation in self.operations:
1412                 text = operation(text)
1413             return text
1414
1415     def lower_case(text):
1416         return text.lower()
1417
1418     def remove_mentions_and_hashtag(text):
1419         return re.sub('@[^\s]+|#[^\s]+', '', text)
1420
1421     def remove_punctuation(text):
1422         return text.replace('[^A-Za-z0-9 ]', '')
1423
1424     def remove_punctuations_layer_2(text):
1425         punct = re.compile(r'[\w\s]')
1426
1427         return punct.sub(r'', text)
1428
1429     def remove_stop_words(text):
1430         #stop_words = get_stop_words()
1431         return " ".join(word for word in text.split() if word not in stop_words)
1432
1433
1434     def remove_emoji(text):
1435         """ Reference : https://gist.github.com/slowkow/7a7f61f495e3d8bb7e3d767f97bd7304b """
1436         emoji_pattern = re.compile("[
1437             u"\U0001F600-\U0001F64F" # emoticons

```

```

1438         u"\U0001F300-\U0001F5FF" # symbols & pictographs
1439         u"\U0001F680-\U0001F6FF" # transport & map symbols
1440         u"\U0001F1E0-\U0001F1FF" # flags
1441         u"\U00002702-\U000027B0"
1442         u"\U000024C2-\U0001F251"
1443         "]" + ",
1444         flags=re.UNICODE)
1445     return emoji_pattern.sub(r'', text)
1446
1447
1448 def remove_HTML(text):
1449     tag = re.compile(r'<.*?>')
1450
1451     return tag.sub(r'', text)
1452
1453 def remove_URL(text):
1454     url = re.compile(r'https?:\/\/\S+|www\.\S+')
1455
1456     return url.sub(r'', text)
1457
1458 def get_stop_words_(all_languages:bool = False):
1459     stop_words_1 = stopwords.words('english')
1460     stop_words_2 = get_stop_words('english')
1461     stop_words_3 = list(swiso.stopwords('en'))
1462
1463
1464     stop_words = stop_words_1 + stop_words_2 + stop_words_3
1465
1466     if all_languages:
1467         stop_words_4 = list(
1468             swiso.stopwords(
1469                 swiso.langs()
1470             )
1471         )
1472         stop_words = stop_words + stop_words_4
1473
1474     stop_words = set(stop_words)
1475
1476     return stop_words
1477
1478 def stemmization(text):
1479     st = PorterStemmer()
1480     return " ".join([st.stem(word) for word in text.split()])
1481
1482 def lemmatization(text):
1483     return " ".join([Word(word).lemmatize() for word in text.split()])
1484
1485 def spell_check(text):
1486     return str(
1487         TextBlob(text).correct()
1488     )
1489 def spell_check_layer_2(text):
1490     spell = SpellChecker()
1491     return spell.correction(text)
1492
1493
1494 def spacy_lemmatization(text):
1495     nlp = spacy.load("en_core_web_sm")
1496     return " ".join([word.lemma_ for word in nlp(text)])

```

```

1497
1498 def final_cleaner(text):
1499     return cleantext.clean(text, all= True)
1500
1501 stop_words = get_stop_words_()
1502
1503 # %%
1504 %%time
1505 clean_text(
1506     df.text.iloc[0],
1507     operations = [
1508         remove_HTML,
1509         remove_URL,
1510         lower_case,
1511         remove_mentiones_and_hashtag,
1512         remove_punctuation,
1513         remove_punctuations_layer_2,
1514         remove_emoji,
1515         remove_stop_words,
1516         #spell_check,
1517         #spell_check_layer_2,
1518         stemmization,
1519         lemmatization,
1520         #spacy_lemmatization,
1521         final_cleaner
1522     ]
1523 )
1524
1525 # %%
1526 df['clean_text'] = df['text'].apply(
1527     Cleaner(
1528         operations = [
1529             remove_HTML,
1530             remove_URL,
1531             lower_case,
1532             remove_mentiones_and_hashtag,
1533             remove_punctuation,
1534             remove_punctuations_layer_2,
1535             remove_emoji,
1536             remove_stop_words,
1537             #spell_check,
1538             #spell_check_layer_2,
1539             stemmization,
1540             lemmatization,
1541             #spacy_lemmatization,
1542             final_cleaner
1543         ]
1544     )
1545 )
1546
1547 df.to_csv('data/cleaned_preprocessed_data.csv')
1548
1549 # %%
1550 df = pd.read_csv('../data/cleaned_preprocessed_data.csv')
1551
1552 cols = [
1553     'id',
1554     'text',
1555     'clean_text',

```

```

1556     'user',
1557     'sentiment',
1558 ]
1559
1560 df = df[cols]
1561
1562 df['clean_text'] = df['clean_text'].str.replace(", ", ' ')
1563 df['clean_text'] = df['clean_text'].str.replace(".", ' ')
1564 df['clean_text'] = df['clean_text'].str.strip()
1565
1566 df.drop(
1567     ['id'],
1568     axis = 1,
1569     inplace = True
1570 )
1571
1572 # %%
1573 neg_tweets = df.loc[df['sentiment']==0]
1574 pos_tweets = df.loc[df['sentiment']==1]
1575
1576 # %%
1577 neg_string = neg_tweets['clean_text'].str.cat(sep=' ')
1578 pos_string = pos_tweets['clean_text'].str.cat(sep=' ')
1579
1580 # %%
1581 plt.figure(figsize=(12,10))
1582 wordcloud_neg = WordCloud(max_font_size=200,
1583     ↳ background_color="white").generate(neg_string)
1584 plt.imshow(wordcloud_neg, interpolation="bilinear")
1585 plt.axis('off')
1586 plt.title("Tweets with Negative sentiment")
1587
1588 # %%
1589 plt.figure(figsize=(12,10))
1590 wordcloud_pos = WordCloud(max_font_size=200,
1591     ↳ background_color="white").generate(pos_string)
1592 plt.imshow(wordcloud_pos, interpolation="bilinear")
1593 plt.axis('off')
1594 plt.title("Tweets with Positive sentiment")
1595
1596 # %%
1597 def remove_unknown_(text):
1598     s = SpellChecker()
1599     unknown_words = s.unknown(text.split())
1600     return " ".join(word for word in text.split() if word not in unknown_words)
1601
1602 # %%
1603 df['unknown_removed_cleaned'] = df['clean_text'].apply(remove_unknown_)
1604
1605 # %%
1606 df.to_csv('data/cleaned_preprocessed_unknown_removed_data.csv')
1607
1608 # %% [markdown]
1609 # # Imports and Read Data
1610
1611 # %%
1612 from __future__ import (

```

```

1613     print_function,
1614     division
1615 )
1616
1617 import matplotlib.pyplot as plt
1618 import stopwordsiso as swiso
1619 import seaborn as sns
1620 import pandas as pd
1621 import numpy as np
1622 import cleantext
1623 import warnings
1624 import random
1625 import string
1626 import pickle
1627 import spacy
1628 import json
1629 import nltk
1630 import time
1631 import abc
1632 import os
1633 import re
1634 import sys
1635 sys.path.append('./src')
1636
1637 from spellchecker import SpellChecker
1638 from stop_words import get_stop_words
1639 from collections import OrderedDict
1640 from nltk.stem import PorterStemmer
1641 from collections import Counter
1642 from nltk.corpus import stopwords
1643 from wordcloud import WordCloud
1644
1645
1646 from textblob import (
1647     TextBlob,
1648     Word
1649 )
1650
1651 from typing import (
1652     Callable,
1653     Iterable,
1654     List,
1655     Union,
1656     Tuple,
1657 )
1658
1659 from utils import (
1660     Classifier,
1661     Pipeline,
1662     json_print,
1663     timeit,
1664     random_seed,
1665     #save_obj
1666 )
1667
1668 from supervised import (
1669     KNeighborsClassifier,
1670     MultiNominalNaiveBayes,
1671     BernaulliNaiveBayes

```

```

1672 from feature import (
1673     BackwardElimination,
1674     #mutual_information
1675 )
1676
1677 # %%
1678 df = pd.read_csv('../data/cleaned_preprocessed_data.csv')
1679
1680 # %% [markdown]
1681 # # Post-preprocessing
1682
1683 # %%
1684 cols = [
1685     'id',
1686     'text',
1687     'clean_text',
1688     'user',
1689     'sentiment',
1690 ]
1691
1692 df = df[cols]
1693
1694 df['clean_text'] = df['clean_text'].str.replace(", ", ' ')
1695 df['clean_text'] = df['clean_text'].str.replace(".", ' ')
1696 df['clean_text'] = df['clean_text'].str.strip()
1697
1698 df.drop(
1699     ['id'],
1700     axis = 1,
1701     inplace = True
1702 )
1703
1704
1705 df['doc_count'] = df['clean_text'].apply(lambda t: len(
1706     str(t).split()
1707 ))
1708 )
1709
1710 df.drop(
1711     df[df['doc_count'] <= 2].index,
1712     inplace = True
1713 )
1714
1715 # %%
1716 df.head()
1717
1718 # %%
1719 df.isna().sum()
1720
1721 # %% [markdown]
1722 # # Construction of Corpus
1723
1724 # %%
1725 df_vocab = df['clean_text'].str.split(expand=True).stack().value_counts().reset_index()
1726 df_vocab.columns = [
1727     'word',
1728     'frequency'
1729 ]
1730 df_vocab.head(10)

```

```

1731
1732 # %%
1733 len(df_vocab)
1734
1735 # %%
1736 word_freq = Counter(
1737     df['clean_text'].str.cat(
1738         sep = ' '
1739     ).split()
1740 )
1741
1742 print(word_freq.most_common(10))
1743
1744 # %% [markdown]
1745 # # Post-processing with Word Count Data
1746
1747 # %%
1748 dict_freq = dict(word_freq)
1749 freq_threshold = 1000
1750
1751 # %%
1752 df['clean_freq_removed_text'] = df['clean_text'].apply(
1753     lambda text : " ".join(
1754         [
1755             word for word in text.split() if dict_freq[word] > freq_threshold
1756         ]
1757     )
1758 )
1759
1760 # %%
1761 df['doc_count_clean_freq_removed_text'] = df['clean_freq_removed_text'].apply(
1762     lambda t: len(
1763         str(t).split()
1764     )
1765 )
1766
1767 df.drop(
1768     df[df['doc_count_clean_freq_removed_text'] <= 2].index,
1769     inplace = True
1770 )
1771
1772 # %%
1773 df_vocab_greq_removed =
1774     ↪ df['clean_freq_removed_text'].str.split(expand=True).stack().value_counts().reset_index()
1775 df_vocab_greq_removed.columns = [
1776     'word',
1777     'frequency'
1778 ]
1779
1780 # %%
1781 class Vocabulary:
1782     def __init__(
1783         self,
1784         vocab_dict,
1785     ):
1786         super(Vocabulary, self).__init__()
1787
1788         assert 'word' in vocab_dict.keys() and 'frequency' in vocab_dict.keys()

```

```

1789
1790     self.vocab_dict = vocab_dict
1791
1792     self.id2word = vocab_dict['word']
1793     self.frequency = vocab_dict['frequency']
1794
1795     self.word2id = {
1796         word: i for i, word in enumerate(self.id2word)
1797     }
1798
1799     def __getitem__(self, idx):
1800
1801         if isinstance(idx, (list, np.ndarray)):
1802             return [self.id2word[i] for i in idx]
1803
1804         return self.id2word[idx]
1805
1806     def __str__(self):
1807         return json_print(self.id2word)
1808
1809     def __repr__(self):
1810         return json_print(self.id2word)
1811
1812
1813     def __len__(self):
1814         return len(self.word2id)
1815
1816     def get_vocab(self):
1817         return self.word2id
1818
1819     def get_frequency_dict(self):
1820         return {self.id2word[i] : freq for i, freq in self.frequency.items()}
1821
1822     def save(self, filename: str) -> None:
1823         self.save_obj(
1824             self.__dict__,
1825             filename
1826         )
1827
1828     def load(self, filename: str) -> None:
1829         self.__dict__ = self.load_obj(filename)
1830
1831     def save_obj(
1832         self,
1833         obj:object,
1834         path:str = None
1835     ) -> None:
1836         """ Saves Python Object as pickle"""
1837         with open(path + '.pkl', 'wb') as f:
1838             pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)
1839
1840
1841     def load_obj(
1842         self,
1843         path:str = None
1844     ) -> object:
1845         """ Loads Python Object from pickle"""
1846         with open(path + '.pkl', 'rb') as f:
1847             return pickle.load(f)

```



```

1848
1849
1850 vocab = Vocabulary(
1851     df_vocab.to_dict()
1852 )
1853
1854
1855 vocab_freq_removed = Vocabulary(
1856     df_vocab_greq_removed.to_dict()
1857 )
1858
1859
1860
1861
1862 # %%
1863 vocab.save('data/vocabulary')
1864 vocab_freq_removed.save('data/vocabulary_freq_removed')
1865
1866 # %%
1867 df.to_parquet('data/final_training_data.parquet')
1868
1869
1870 # %%
1871 """
1872     INFORMATION:
1873
1874     Course      : EEE485/585
1875     Name        : Can Kocagil
1876     ID          : 21602218
1877     E-mail      : can.kocagil@ug.bilkent.edu.tr
1878     Assignment  : Progress Report
1879
1880 """
1881
1882 from __future__ import (
1883     print_function,
1884     division
1885 )
1886
1887 import matplotlib.pyplot as plt
1888 import stopwordsiso as swiso
1889 import seaborn as sns
1890 import pandas as pd
1891 import numpy as np
1892 import cleantext
1893 import warnings
1894 import random
1895 import string
1896 import pickle
1897 import spacy
1898 import json
1899 import nltk
1900 import time
1901 import abc
1902 import os
1903 import re
1904 import sys
1905 sys.path.append('./src')
1906

```

```

1907 from sklearn.utils.validation import (
1908     check_X_y,
1909     check_array
1910 )
1911
1912 from sklearn.metrics import (
1913     confusion_matrix,
1914     precision_score
1915 )
1916
1917 from sklearn.feature_extraction.text import (
1918     CountVectorizer,
1919     TfidfVectorizer
1920 )
1921
1922 from sklearn.model_selection import train_test_split
1923
1924 from typing import (
1925     Callable,
1926     Iterable,
1927     List,
1928     Union,
1929     Tuple,
1930 )
1931
1932 from utils import (
1933     Pipeline,
1934     Classifier,
1935     json_print,
1936     timeit,
1937     random_seed
1938 )
1939
1940 from supervised import (
1941     KNeighborsClassifier,
1942     MultiNominalNaiveBayes,
1943     BernaulliNaiveBayes
1944 )
1945
1946 from feature import (
1947     BackwardElimination,
1948     #mutual_information
1949 )
1950
1951 # %%
1952 df = pd.read_parquet('../data/final_training_data.parquet')
1953
1954 df = df.sample(10000)
1955
1956 # %% [markdown]
1957 # # Feature Extraction
1958
1959 # %%
1960 doc = df['clean_freq_removed_text']
1961
1962 vectorizer= TfidfVectorizer(
1963     #max_features = 10000,
1964     ngram_range = (1, 1),
1965     analyzer='word',
1966     stop_words='english',
1967     norm='l2'

```

```

1966 )
1967
1968 vectorizer.fit(doc)
1969
1970 features = vectorizer.transform(doc)
1971
1972 # %%
1973 features.shape
1974
1975 # %% [markdown]
1976 # # Train and Test Split
1977
1978 # %%
1979 concat_doc_count = False
1980
1981 if concat_doc_count:
1982     X = np.concatenate(
1983         [
1984             features.toarray(),
1985             df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
1986         ],
1987         axis = 1
1988     )
1989
1990 else:
1991     X = features.toarray()
1992
1993 y = df['sentiment']
1994
1995 # %%
1996 X_train, X_test, y_train, y_test = train_test_split(
1997     X, y,
1998     test_size = 0.22,
1999     random_state = 21
2000 )
2001
2002 # %%
2003 X_train = pd.DataFrame(X_train)
2004 X_test = pd.DataFrame(X_test)
2005 y_train = pd.DataFrame(y_train)
2006 y_train.columns = ['class']
2007
2008 print(f"X_train shape: {X_train.shape}")
2009 print(f"X_test shape: {X_test.shape}")
2010 print(f"y_train shape: {y_train.shape}")
2011 print(f"y_test shape: {y_test.shape}")
2012
2013 # %% [markdown]
2014 # # Modelling
2015
2016 # %% [markdown]
2017 # ## Multinomial Naive Bayes
2018
2019 # %%
2020 doc = df['clean_text']
2021
2022 vectorizer = CountVectorizer(
2023     max_features = 1000,
2024     ngram_range = (1, 1)

```

```

2025 )
2026 vectorizer.fit(doc)
2027
2028 features = vectorizer.transform(doc)
2029
2030 concat_doc_count = False
2031
2032 if concat_doc_count:
2033     X = np.concatenate(
2034         [
2035             features.toarray(),
2036             df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
2037         ],
2038         axis = 1
2039     )
2040
2041 else:
2042     X = features.toarray()
2043
2044 y = df['sentiment']
2045
2046
2047 X_train, X_test, y_train, y_test = train_test_split(
2048     X, y,
2049     test_size = 0.22,
2050     random_state = 21
2051 )
2052
2053 X_train = pd.DataFrame(X_train)
2054 X_test = pd.DataFrame(X_test)
2055 y_train = pd.DataFrame(y_train)
2056 y_train.columns = ['class']
2057 y_train = y_train.reset_index().drop('index', 1)
2058
2059 print(f"X_train shape: {X_train.shape}")
2060 print(f"X_test shape: {X_test.shape}")
2061 print(f"y_train shape: {y_train.shape}")
2062 print(f"y_test shape: {y_test.shape}")
2063
2064 # %%
2065 class MultiNominalNaiveBayes(Classifier):
2066     def __init__(self, alpha=0.0001):
2067         super(MultiNominalNaiveBayes, self).__init__()
2068         self.alpha = alpha
2069
2070         self._hyperparams['alpha'] = self.alpha
2071         self._name = 'MultiNominal NaiveBayes Classifier'
2072
2073     @timeit
2074     def fit(
2075         self,
2076         X_train:pd.DataFrame,
2077         y_train: pd.DataFrame,
2078         **fit_params
2079     ):
2080
2081         m, n = X_train.shape
2082         self.classes = np.unique(y_train)
2083         n_classes = len(self.classes)

```

```

2084
2085     if not isinstance(X_train, pd.DataFrame):
2086         X_train = pd.DataFrame(X_train)
2087
2088     self.priors = y_train.value_counts(normalize = True).values
2089     self.counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
2090     likelihoods = self.counts.T / self.counts.sum(1).values.reshape(-1, n_classes)
2091     ↪ + self.alpha
2092     self.likelihoods = likelihoods.values #.T
2093     self.log_priors = np.log(self.priors)
2094
2095     return self
2096
2097 @timeit
2098 def predict(self, X_test):
2099
2100     if isinstance(X_test, pd.DataFrame):
2101         X_test = X_test.values
2102
2103     self.log_likelihoods = X_test @ np.log(self.likelihoods)
2104     ↪ #(np.log(self.likelihoods) @ X_test.T).T
2105     self.posterioriors = self.log_likelihoods + self.log_priors
2106
2107     return self.classes[
2108         self.posterioriors.argmax(1)
2109     ]
2110
2111 def posteriors(self):
2112     return self.posterioriors
2113
2114 # %%
2115 m = MultiNominalNaiveBayes().fit(X_train.values, y_train)
2116 (m.predict(X_test) == y_test.values).mean()
2117
2118 # %%
2119 pipe = Pipeline([
2120     ('classifier',
2121      MultiNominalNaiveBayes(
2122      ))
2123 ])
2124
2125 pipe.fit(
2126     X_train = X_train,
2127     y_train = y_train
2128 )
2129
2130
2131 score = pipe.score(
2132     X_test = X_test.values,
2133     y_test = y_test
2134 )
2135
2136 predictions = pipe.predict(
2137     X_test = X_test.values,
2138 )
2139
2140 conf_matrix = pipe.model.confusion_matrix(

```

```

2141     predictions.reshape(-1, ),
2142     y_test.values.reshape(-1, ),
2143 )
2144
2145 pipe.model.visualize_confusion_matrix(conf_matrix)
2146
2147 print(f"Accuracy Score: {score['accuracy']}")
2148 print(f"Confusion Matrix: \n {conf_matrix}")
2149 print(f"The number of parameters to be estimated: {pipe.model.priors.size +
    ↪ pipe.model.likelihoods.size - 1}")
2150
2151
2152 # %% [markdown]
2153 # ## Bernoulli Naive Bayes
2154
2155 # %%
2156 class BernoulliNaiveBayes(Classifier):
2157     def __init__(self, alpha = 0.001):
2158         super(BernoulliNaiveBayes, self).__init__()
2159         self.alpha = alpha
2160
2161         self._hyperparams['alpha'] = self.alpha
2162         self._name = 'Bernoulli NaiveBayes Classifier'
2163
2164 @timeit
2165 def fit(self, X_train, y_train, **fit_params):
2166     self.classes = np.unique(y_train)
2167
2168     n_classes = len(self.classes)
2169
2170     if not isinstance(X_train, pd.DataFrame):
2171         X_train = pd.DataFrame(X_train)
2172
2173     self.priors = y_train.value_counts(normalize = True).values
2174     self.log_priors = np.log(self.priors)
2175
2176     counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
2177     likelihoods = counts.T / counts.sum(1).values.reshape(-1, n_classes) +
    ↪ self.alpha
2178     self.likelihoods = likelihoods.T.values
2179
2180     return self
2181
2182 @timeit
2183 def predict(self, X_test):
2184
2185     self.posterioriors = np.array(
2186         [
2187             (
2188                 (np.log(self.likelihoods) * x) + (np.log(1 - self.likelihoods) *
    ↪ np.abs(x - 1))
2189             ).sum(axis = 1) + self.log_priors for x in X_test
2190         ]
2191     )
2192
2193
2194     return self.classes[
2195         self.posterioriors.argmax(1)
2196     ]

```

```

2197
2198 # %%
2199 pipe = Pipeline([
2200     ('classifier',
2201      BernaulliNaiveBayes(
2202          alpha=0.00019
2203      )
2204   ])
2205 ])
2206
2207
2208 pipe.fit(
2209     X_train = np.where(X_train >= 1, 1, 0),
2210     y_train = y_train
2211 )
2212
2213 score = pipe.score(
2214     X_test = np.where(X_test >= 1, 1, 0),
2215     y_test = y_test
2216 )
2217
2218 conf_matrix = pipe.model.confusion_matrix(
2219     predictions,
2220     y_test.values
2221 )
2222
2223 print(f"Accuracy Score: {score['accuracy']}")
2224 print(f"Confusion Matrix: \n {conf_matrix}")
2225
2226 # %%
2227 def mutual_information(
2228     x1:np.ndarray,
2229     x2:np.ndarray
2230 ) -> np.float:
2231
2232     jh = np.histogram2d(
2233         x1,
2234         x2,
2235         bins = (
2236             256, 256
2237         )
2238     )[0]
2239
2240     jh = jh + 1e-15
2241
2242     sh = np.sum(jh)
2243     jh = jh / sh
2244
2245     y1 = np.sum(
2246         jh,
2247         axis=0
2248     ).reshape(
2249         (-1, jh.shape[0])
2250     )
2251
2252     y2 = np.sum(
2253         jh,
2254         axis=1
2255     ).reshape(

```

```

2256         (jh.shape[1], -1)
2257     )
2258
2259
2260     return (
2261         np.sum(jh * np.log(jh)) - np.sum(y1 * np.log(y1)) - np.sum(y2 * np.log(y2))
2262     )
2263
2264
2265     # %%
2266     mutual_infos = []
2267     feature_dimension = X_train.shape[1]
2268
2269     for feature in range(feature_dimension):
2270
2271         mutual_infos.append(
2272             mutual_information(
2273                 X_train.values[:, feature].reshape(-1, ),
2274                 y_train.values.reshape(-1, )
2275             )
2276         )
2277
2278     mutual_infos = np.array(mutual_infos)
2279
2280
2281     # %%
2282     sorted_mutual_infos = np.flip(
2283         mutual_infos.argsort(
2284             axis = 0
2285         )
2286     )
2287
2288     cache = []
2289     selected_features = []
2290
2291
2292     for iteration in range(0, 1000, 100):
2293
2294         features = sorted_mutual_infos[iteration: iteration + 100]
2295
2296         selected_features.extend(
2297             features
2298         )
2299
2300
2301         pipe = Pipeline([
2302             ('classifier',
2303              BernaulliNaiveBayes(
2304                  alpha=0.00019
2305              )
2306         ])
2307
2308
2309         since = time.time()
2310
2311         pipe.fit(
2312             X_train = np.where(X_train >= 1, 1, 0)[: , selected_features],
2313             y_train = y_train,
2314         )

```



```

2315
2316     time_passed = time.time() - since
2317
2318     score = pipe.score(
2319         X_test = np.where(X_test >= 1, 1, 0)[: , selected_features],
2320         y_test = y_test.values
2321     )
2322
2323     cache.append(
2324         {
2325             'Iter num'           : iteration,
2326             'Features'           : features,
2327             'Feature Dim'        : len(selected_features),
2328             'Fitting Time (s)'    : round(time_passed, 4),
2329             'Accuracy'           : score['accuracy']
2330         }
2331     )
2332
2333
2334 df_cache = pd.DataFrame(cache).sort_values('Accuracy', ascending = False)
2335 df_cache['Iter num'] = (df_cache['Iter num'] / 100).astype(int)
2336
2337 # %%
2338 df_cache
2339
2340 # %% [markdown]
2341 # ## k-Nearest Neighbor
2342
2343 # %%
2344 doc = df['clean_freq_removed_text']
2345
2346 vectorizer= TfidfVectorizer(
2347     max_features = 10000,
2348     ngram_range = (1, 1),
2349     analyzer='word',
2350     stop_words='english',
2351     norm='l2'
2352 )
2353
2354 vectorizer.fit(doc)
2355
2356 features = vectorizer.transform(doc)
2357
2358 concat_doc_count = False
2359
2360 if concat_doc_count:
2361     X = np.concatenate(
2362         [
2363             features.toarray(),
2364             df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
2365         ],
2366         axis = 1
2367     )
2368
2369 else:
2370     X = features.toarray()
2371
2372 y = df['sentiment']
2373

```

```

2374 X_train, X_test, y_train, y_test = train_test_split(
2375     X, y,
2376     test_size = 0.22,
2377     random_state = 21
2378 )
2379
2380 X_train = pd.DataFrame(X_train)
2381 X_test = pd.DataFrame(X_test)
2382 y_train = pd.DataFrame(y_train)
2383 y_train.columns = ['class']
2384
2385 print(f"X_train shape: {X_train.shape}")
2386 print(f"X_test shape: {X_test.shape}")
2387 print(f"y_train shape: {y_train.shape}")
2388 print(f"y_test shape: {y_test.shape}")
2389
2390 # %%
2391 distances = [
2392     'euclidean',
2393     'manhattan',
2394     'cosine'
2395 ]
2396
2397 k_neighbors = 7
2398
2399 cache_model = []
2400
2401 for distance in distances:
2402     pipe = Pipeline([
2403         ('classifier',
2404          KNeighborsClassifier(
2405              k_neighbors = k_neighbors,
2406              distance_metric = distance
2407          )
2408     ])
2409
2410     pipe.fit(
2411         X_train = X_train.values,
2412         y_train = y_train.values
2413     )
2414
2415     score = pipe.score(
2416         X_test = X_test.values,
2417         y_test = y_test.values
2418     )
2419
2420
2421     cache_model.append(
2422         (
2423             pipe.model,
2424             pipe.model.distance_metric,
2425             score['accuracy']
2426         )
2427     )
2428
2429
2430 result_df = pd.DataFrame(
2431     cache_model,
2432     columns = [

```

```

2433         'Model',
2434         'Distance',
2435         'Accuracy'
2436     ]
2437 )
2438
2439 result_df
2440
2441 # %% [markdown]
2442 # # Logistic Regression
2443
2444 # %%
2445 class LogisticRegression(Classifier):
2446     def __init__(self):
2447         super().__init__()
2448
2449     def init_params(self,
2450         input_shape:int,
2451         output_shape:int = 1
2452     ):
2453         self.__random_seed()
2454
2455         #assert self.X_train.shape[1] == self.X_test.shape[1], 'Improper feature
2456         ↪ dimension!'
2457
2458         W_high = self.__init_xavier(input_shape, output_shape)
2459         W_low = - W_high
2460         W_size = (input_shape, output_shape)
2461         B_size = (1, output_shape)
2462
2463         self.W = np.random.uniform(
2464             W_low,
2465             W_high,
2466             size = W_size
2467         )
2468
2469         self.b = np.random.uniform(
2470             W_low,
2471             W_high,
2472             size = B_size
2473         )
2474
2475     def __random_seed(self, seed = 32):
2476         """ Random seed for reproducibility """
2477         random.seed(seed)
2478         np.random.seed(seed)
2479
2480     def __init_xavier(self, L_pre, L_post):
2481         """ Given the size of the input node and hidden node, initialize the weights
2482         ↪ drawn from uniform distribution ~ Uniform[- sqrt(6/(L_pre + L_post)) ,
2483         ↪ sqrt(6/(L_pre + L_post))] """
2484         return np.sqrt(6/(L_pre + L_post))
2485
2486     def __train_config(self,
2487         lr:float,
2488         batch_size:int,
2489         epochs:int,
2490     ):

```

```

2489     self.lr = lr
2490     self.batch_size = batch_size
2491     self.epochs = epochs
2492
2493     def sigmoid(self, X, grad = False):
2494         """ Computing sigmoid and it's gradient w.r.t. it's input """
2495         sig = 1/(1 + np.exp(-X))
2496
2497         return sig * (1-sig) if grad else sig
2498
2499     def __forward(self, X):
2500
2501         Z = (X @ self.W) + self.b
2502         A = self.sigmoid(Z)
2503
2504         return {
2505             "Z": Z,
2506             "A": A
2507         }
2508
2509
2510     def __SGD(self, grads):
2511         self.W -= self.lr * grads['W']
2512         self.b -= self.lr * grads['b']
2513
2514
2515     def matrix_back_prop(self, outs, X, Y):
2516         """ Matrix form backward propagation """
2517         m = self.batch_size
2518
2519         Z = outs['Z']
2520         A = outs['A']
2521
2522         dZ = (A-Y) * self.sigmoid(Z, grad = True)
2523         dW = (1 / m) * (X.T @ dZ)
2524         db = (1 / m) * np.sum(dZ, axis=0, keepdims=True)
2525
2526         assert self.W.shape == dW.shape, f'Error in weight shapes!, {dW.shape} does not
2527         ↪ match with {self.W.shape}'
2528         assert self.b.shape == db.shape, f'Error in bias shapes!, {db.shape} does not
2529         ↪ match with {self.b.shape}'
2530
2531         grads = {}
2532         grads['W'] = dW
2533         grads['b'] = db
2534
2535         return grads
2536
2537     def backward(self,
2538                 outs,
2539                 X,
2540                 Y
2541                 ):
2542         return self.matrix_back_prop(
2543             outs,
2544             X,
2545             Y
2546         )

```

```

2546
2547
2548 def BinaryCrossEntropyLoss(self, pred, label):
2549     m = pred.shape[0]
2550     preds = np.clip(pred, 1e-16, 1 - 1e-16)
2551     loss = np.sum(-label * np.log(preds + 1e-20) - (1 - label) * np.log(1 - preds +
↪ 1e-20))
2552     return loss / m
2553
2554 def eval(self, x, y, knob:float = 0.5):
2555     predictions = self.__forward(x)
2556     predictions = predictions['A']
2557     predictions[predictions>=knob] = 1
2558     predictions[predictions< knob] = 0
2559     acc_score = self.accuracy(predictions, y)
2560
2561     return acc_score
2562
2563 def __accuracy(self, pred, label):
2564     return np.sum(pred == label) / pred.shape[0]
2565
2566
2567 @timeit
2568 def fit(
2569     self,
2570     X_train,
2571     y_train,
2572     X_test,
2573     y_test,
2574     lr:float = 1e-2,
2575     batch_size:int = 32,
2576     epochs:int = 100,
2577     verbose = True
2578 ):
2579     """
2580     Given the training dataset, their labels and number of epochs
2581     fitting the model, and measure the performance
2582     by validating training dataset.
2583     """
2584
2585     self.init_params(
2586         input_shape = X_train.shape[1]
2587     )
2588
2589     self.__train_config(
2590         lr,
2591         batch_size,
2592         epochs,
2593     )
2594
2595     self.history = {}
2596
2597     self.history['train'] = {
2598         'loss': [],
2599         'acc' : []
2600     }
2601
2602     self.history['val'] = {
2603         'loss': [],

```

```

2604         'acc' : []
2605     }
2606
2607     m = self.batch_size
2608
2609     self.sample_size_train = X_train.shape[0]
2610
2611     for epoch in range(self.epochs):
2612
2613         perm = np.random.permutation(self.sample_size_train)
2614
2615         for i in range(self.sample_size_train // m):
2616
2617             shuffled_index = perm[i*m: (i+1)*m]
2618
2619             X_feed = X_train[shuffled_index]
2620             y_feed = y_train[shuffled_index]
2621
2622             outs = self.__forward(X_feed)
2623             grads = self.backward(
2624                 outs,
2625                 X_feed,
2626                 y_feed
2627             )
2628             self.__SGD(grads)
2629
2630
2631             loss_train = self.BinaryCrossEntropyLoss(
2632                 self.__forward(X_train)[:, 'A'],
2633                 y_train
2634             )
2635
2636             acc_train = self.eval(
2637                 X_train,
2638                 y_train
2639             )
2640
2641             self.history['train']['loss'].append(loss_train)
2642             self.history['train']['acc'].append(acc_train)
2643
2644
2645
2646             loss_val = self.BinaryCrossEntropyLoss(
2647                 self.__forward(X_test)[:, 'A'],
2648                 y_test
2649             )
2650
2651             acc_val = self.eval(
2652                 X_test,
2653                 y_test
2654             )
2655
2656             self.history['val']['loss'].append(loss_val)
2657             self.history['val']['acc'].append(acc_val)
2658
2659             if verbose:
2660                 print(f"[{epoch}/{self.epochs}] -----> Training : BCE: {loss_train} and

```

↪ Acc: {acc_train}")

```

2661         print(f"[{epoch}/{self.epochs}] -----> Testing : BCE: {loss_val} and
2662             ↳ Acc: {acc_val}")
2663
2664     def __str__(self):
2665         model = LogisticRegression().__class__.__name__
2666         model += f' with hyperparameters (learning rate,batch_size,epochs) =
2667             ↳ ({self.lr,self.batch_size,self.epochs})'
2668         num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] *
2669             ↳ self.b.shape[1]
2670         model += f'\n There are {num_params} number of traniable parameters'
2671         return model
2672
2673     def __repr__(self):
2674         model = LogisticRegression().__class__.__name__
2675         model += f' with hyperparameters (learning rate,batch_size,epochs) =
2676             ↳ ({self.lr,self.batch_size,self.epochs})'
2677         num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] *
2678             ↳ self.b.shape[1]
2679         model += f'\n There are {num_params} number of traniable parameters'
2680         return model
2681
2682     def plot_history(self):
2683
2684         fig,axs = plt.subplots(1,2,figsize = (24,8))
2685         axs[0].plot(self.history['train']['loss'],color = 'orange',label = 'Training')
2686         axs[0].plot(self.history['val']['loss'], label = 'Validation')
2687         axs[0].set_xlabel('Epochs')
2688         axs[0].set_ylabel('BCE Loss')
2689         axs[0].set_title(f'Binary Cross Entropy Loss Over Iterations with Learning Rate
2690             ↳  $\eta$  = {self.lr}')
2691         axs[0].legend(loc="upper right")
2692         axs[0].grid()
2693
2694         axs[1].plot(self.history['train']['acc'],color = 'orange',label = 'Training')
2695         axs[1].plot(self.history['val']['acc'], label = 'Validation')
2696         axs[1].set_xlabel('Epochs')
2697         axs[1].set_ylabel('Accuracy')
2698         maxs = round(max(self.history['train']['acc']),3),
2699             ↳ round(max(self.history['val']['acc']),3)
2700         axs[1].set_title(f'Accuracy Over Iterations with Learning Rate  $\eta$  =
2701             ↳ {self.lr} \n Best Accuracy in (Training,Validation) = {maxs} ')
2702         axs[1].legend(loc="lower right")
2703         axs[1].grid()
2704
2705     # %%
2706     train_config = dict(
2707         lr = 1e-3,
2708         batch_size = 64,
2709         epochs = 1000,
2710         verbose = False,
2711     )
2712
2713     model = LogisticRegression()
2714     model.fit(
2715         X_train.values,
2716         y_train.values.reshape(-1, 1),
2717         X_test.values,
2718         y_test.values.reshape(-1, 1),

```

```

2712     **train_config
2713 )
2714 model.plot_history()
2715 print(model)
2716
2717 # %%
2718 train_config = dict(
2719     lr          = 1e-3,
2720     batch_size  = 64,
2721     epochs      = 100000,
2722     verbose     = False,
2723 )
2724
2725 model = LogisticRegression()
2726 model.fit(
2727     X_train.values,
2728     y_train.values.reshape(-1, 1),
2729     X_test.values,
2730     y_test.values.reshape(-1, 1),
2731     **train_config
2732 )
2733 model.plot_history()
2734 print(model)
2735
2736 # %%
2737 train_config = dict(
2738     lr          = 9e-3,
2739     batch_size  = 128,
2740     epochs      = 10000,
2741     verbose     = False,
2742 )
2743
2744 model = LogisticRegression()
2745 model.fit(
2746     X_train.values,
2747     y_train.values.reshape(-1, 1),
2748     X_test.values,
2749     y_test.values.reshape(-1, 1),
2750     **train_config
2751 )
2752 model.plot_history()
2753 print(model)
2754
2755 # %% [markdown]
2756 # # MLP (Multi-Layer Perceptron)
2757
2758 # %%
2759 class MLP(Classifier):
2760
2761     def __init__(self,
2762         input_size = X_train.shape,
2763         batch_size = 19 ,
2764         n_neurons  = 76 ,
2765         mean       = 0,
2766         std        = 1,
2767         lr         = 1e-1,
2768         distribution = 'Xavier'
2769     ):
2770

```



```

2771
2772     np.random.seed(15)
2773     self.lr = lr
2774     self.mse_train = {}
2775     self.mce_train = {}
2776     self.mse_test = {}
2777     self.mce_test = {}
2778
2779     self.sample_size = input_size[0]
2780     self.feature_size = input_size[1]
2781     self.batch_size = batch_size
2782     self.n_neurons = n_neurons
2783     self.mean, self.std = mean, std
2784
2785     self.dist = distribution
2786
2787
2788     self.n_update = round((self.sample_size/self.batch_size))
2789
2790     self.W1_size = self.feature_size, self.n_neurons
2791     self.W2_size = self.n_neurons, 1
2792
2793     self.B1_size = 1, self.n_neurons
2794     self.B2_size = 1, 1
2795
2796     self.B1 = np.random.normal(loc = self.mean, scale = self.std, size =
↳ (self.B1_size)) * 0.01
2797     self.B2 = np.random.normal(loc = self.mean, scale = self.std, size =
↳ (self.B2_size)) * 0.01
2798
2799     self.he_scale1 = np.sqrt(2/self.feature_size)
2800     self.he_scale2 = np.sqrt(2/self.n_neurons)
2801     self.xavier_scale1 = np.sqrt(2/(self.feature_size+self.n_neurons))
2802     self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))
2803
2804     if (self.dist == 'Zero') :
2805         self.W1 = np.zeros((self.W1_size))
2806         self.W2 = np.zeros((self.W2_size))
2807
2808     elif (self.dist == 'Gauss'):
2809         self.W1 = np.random.normal(loc = self.mean, scale = self.std, size =
↳ (self.W1_size))* 0.01
2810         self.W2 = np.random.normal(loc = self.mean, scale = self.std, size =
↳ (self.W2_size))* 0.01
2811
2812     elif (self.dist == 'He'):
2813         self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) * self.he_scale1
2814         self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) * self.he_scale2
2815
2816     elif (self.dist == 'Xavier'):
2817
2818         self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) *
↳ self.xavier_scale1
2819         self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) *
↳ self.xavier_scale2
2820
2821
2822     def forward(self,X):
2823

```

```

2824
2825     Z1 = (X @ self.W1) + self.B1
2826     A1 = np.tanh(Z1)
2827     Z2 = (A1 @ self.W2) + self.B2
2828     A2 = np.tanh(Z2)
2829
2830     return {
2831         "Z1": Z1,
2832         "A1": A1,
2833         "Z2": Z2,
2834         "A2": A2
2835     }
2836
2837
2838     def tanh(self,X):
2839         return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))
2840
2841     def tanh_der(self,X):
2842         return 1-(np.tanh(X)**2)
2843
2844     def backward(self,outs, X, Y):
2845         m = (self.batch_size)
2846
2847         Z1 = outs['Z1']
2848         A1 = outs['A1']
2849         Z2 = outs['Z2']
2850         A2 = outs['A2']
2851
2852         dZ2 = (A2-Y)* self.tanh_der(Z2)
2853         dW2 = (1/m) * (A1.T @ dZ2)
2854         dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)
2855
2856         dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
2857         dW1 = (1/m) * (X.T @ dZ1)
2858         dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)
2859
2860
2861         return {
2862             "dW1": dW1,
2863             "dW2": dW2,
2864             "dB1": dB1,
2865             "dB2": dB2
2866         }
2867
2868     def Loss(self,pred, y_true, knob = 0):
2869
2870         mse = np.square(pred-y_true).mean()
2871
2872         pred[pred>=knob]=1
2873         pred[pred<knob]=-1
2874
2875         mce = (pred == y_true).mean()
2876
2877         return {
2878             'MSE':mse,
2879             'MCE':mce
2880         }
2881
2882

```

```

2883 def SGD(self, grads):
2884     self.W1 -= self.lr * grads['dW1']
2885     self.W2 -= self.lr * grads['dW2']
2886     self.B1 -= self.lr * grads['dB1']
2887     self.B2 -= self.lr * grads['dB2']
2888
2889 def fit(self, X, Y, X_test, y_test, epochs = 300, verbose=True):
2890     """
2891     Given the training dataset, their labels and number of epochs
2892     fitting the model, and measure the performance
2893     by validating training dataset.
2894     """
2895
2896     m = self.batch_size
2897
2898     for epoch in range(epochs):
2899         perm = np.random.permutation(self.sample_size)
2900
2901         for i in range(self.n_update):
2902
2903
2904             batch_start = i * m
2905             batch_finish = (i+1) * m
2906             index = perm[batch_start:batch_finish]
2907
2908             X_feed = X[index]
2909             y_feed = Y[index]
2910
2911
2912             outs = self.forward(X_feed)
2913             loss = self.Loss(
2914                 outs['A2'],
2915                 y_feed
2916             )
2917
2918             outs_test = self.forward(X_test)
2919             loss_test = self.Loss(
2920                 outs_test['A2'],
2921                 y_test
2922             )
2923
2924             grads = self.backward(
2925                 outs,
2926                 X_feed,
2927                 y_feed
2928             )
2929
2930             self.SGD(grads)
2931
2932             self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
2933             self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
2934             self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
2935             self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']
2936
2937         if verbose:
2938             print(f"[{epoch}/{epochs}] -----> Training :MSE: {loss['MSE']} and MCE:
2939                 ↳ {loss['MCE']}")
2940             print(f"[{epoch}/{epochs}] -----> Testing :MSE: {loss_test['MSE']} and
2941                 ↳ MCE: {loss_test['MCE']}")

```

```

2940
2941     def history(self):
2942         return {
2943             'Train_MSE' : self.mse_train,
2944             'Train_MCE' : self.mce_train,
2945             'Test_MSE'  : self.mse_test,
2946             'Test_MCE'  : self.mce_test
2947         }
2948
2949     # %%
2950     initialize = 'Xavier'
2951     input_size = X_train.shape
2952     batch_size = 64
2953     hidden_neurons = 256
2954     epochs = 100
2955
2956     model = MLP(
2957         input_size,
2958         batch_size,
2959         hidden_neurons,
2960         lr=1e-3,
2961         distribution = 'Gauss'
2962     )
2963
2964     # %%
2965     model.fit(
2966         X_train.values,
2967         y_train.values.reshape(-1, 1),
2968         X_test.values,
2969         y_test.values.reshape(-1, 1),
2970         epochs
2971     )
2972
2973     # %%
2974     history = model.history()
2975
2976     # %%
2977     plt.rcParams['figure.figsize'] = (9,6)
2978     plt.plot(history['Train_MCE'].values())
2979     plt.xlabel('# of Epoch')
2980     plt.ylabel('Mean Classification Error')
2981     plt.title('MCE versus Epoch in Training')
2982     plt.show()
2983
2984     plt.plot(history['Train_MSE'].values(),color = 'green')
2985     plt.xlabel('# of Epoch')
2986     plt.ylabel('Mean Squared Error')
2987     plt.title('MSE versus Epoch in Training')
2988     plt.show()
2989
2990     plt.plot(history['Test_MCE'].values(),color = 'orange')
2991     plt.xlabel('# of Epoch')
2992     plt.ylabel('Mean Classification Error')
2993     plt.title('MCE versus Epoch in Validation')
2994     plt.show()
2995
2996
2997     plt.plot(history['Test_MSE'].values(),color = 'blue')
2998     plt.xlabel('# of Epoch')

```

```
2999 plt.ylabel('Mean Squared Error')
3000 plt.title('MSE versus Epoch in Validation')
3001 plt.show()
```

REFERENCES

- [1] 1.9. *Naive Bayes*. URL: https://scikit-learn.org/stable/modules/naive_bayes.html.
- [2] Ethan Z. Booker. *Multinomial/ Multimodal naive Bayes*. Nov. 2020. URL: <https://iq.opengenus.org/multinomial-naive-bayes/>.
- [3] Alec Go, Richa Bhayani, and Lei Huang. “Twitter sentiment classification using distant supervision”. In: *CS224N project report, Stanford* 1.12 (2009), p. 2009.
- [4] Namita Mutha. *Bernoulli naive Bayes*. May 2020. URL: <https://iq.opengenus.org/bernoulli-naive-bayes/>.
- [5] Anamika Thanda et al. *What is logistic regression? A beginner’s guide [2021]*. Sept. 2021. URL: <https://careerfoundry.com/en/blog/data-analytics/what-is-logistic-regression/>.
- [6] Wikipedia contributors. *Mutual information* — *Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Mutual_information&oldid=1056818318. [Online; accessed 25-November-2021]. 2021.