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

List of Tables

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 $0 =$ negative and $4 =$ 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 are 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 linguistical 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 linguistical 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 sentience. 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 preprocessing 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.

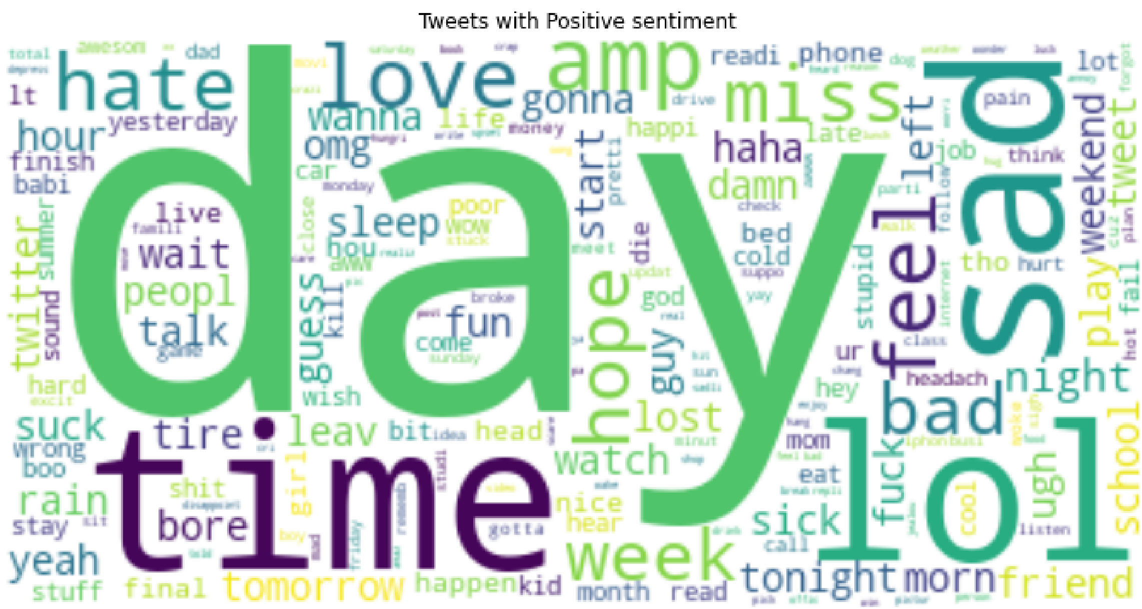

FIGURE 1. Positive-sentiment Tweets' Word Cloud

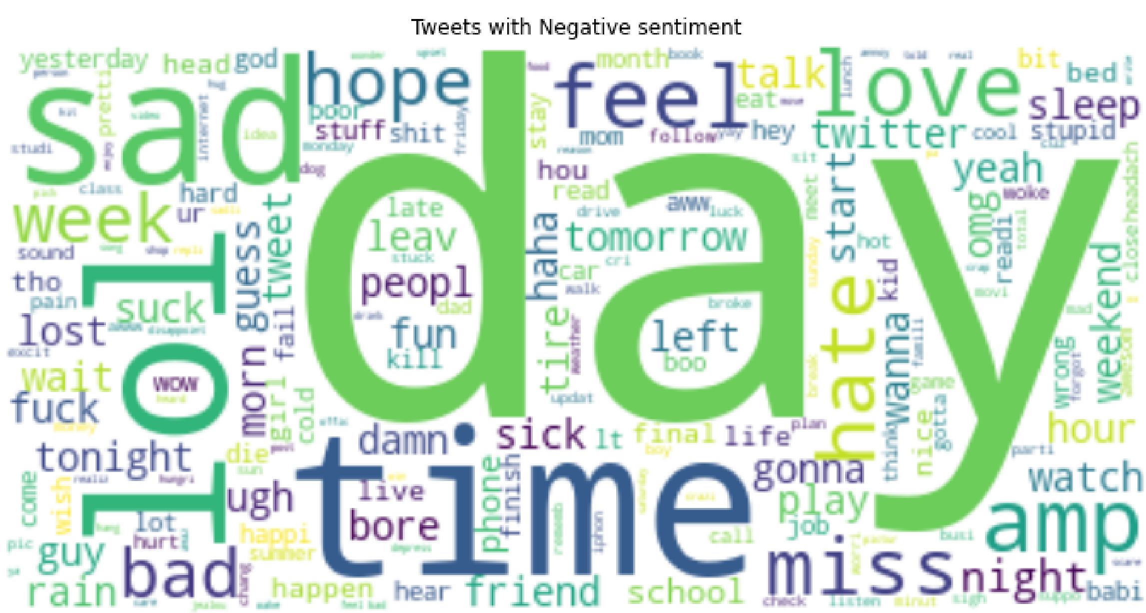

FIGURE 2. Negative-sentiment Tweets' Word Cloud

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]

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)} \tag{1}$$

Using the naive conditional independence assumption that

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

for all, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)\prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)} \tag{3}$$

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

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$
$$\Downarrow \tag{4}$$
$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

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.

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

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

$$P(x_i \mid y) = P(i \mid y) x_i + (1 - P(i \mid y))(1 - x_i) \tag{6}$$

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 $\theta_y = (\theta_{y1}, \ldots, \theta_{yn})$ 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 $P(x_i \mid y)$ 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:

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

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

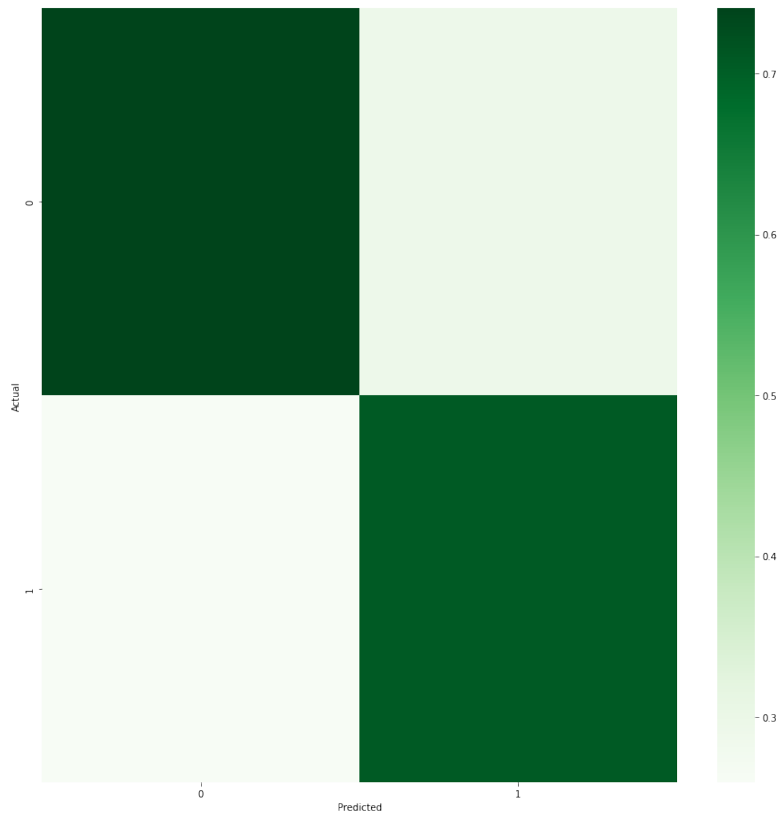

FIGURE 3. The visualization of the confusion matrix for Multinomial Naive Bayes

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

$$p(X_i) = \frac{1}{1 + e^{-(c + X_i^T w)}} \; \textit{for } i \in \{1, \ldots, n\} \tag{8}$$

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

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

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.

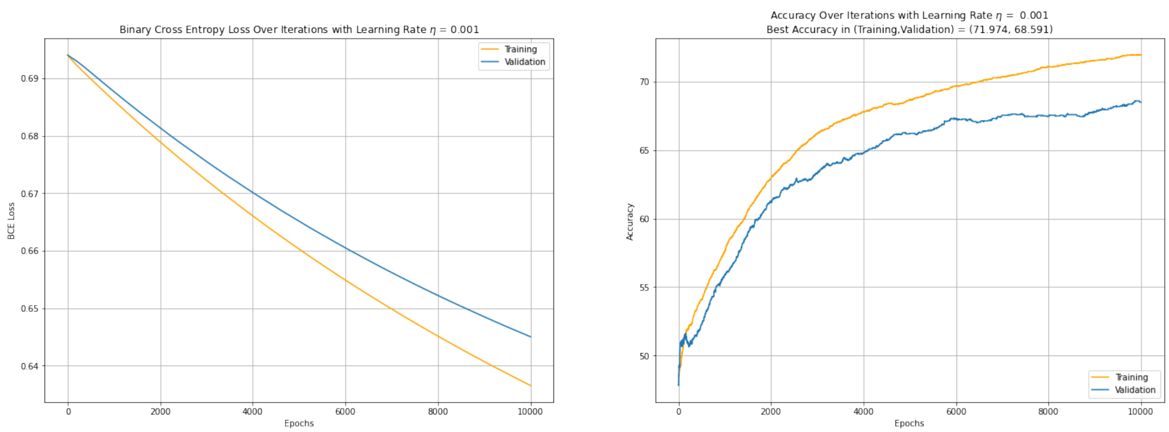

FIGURE 4. The visualization of the training & testing loss and accuracy of Logistic Regression-I

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))

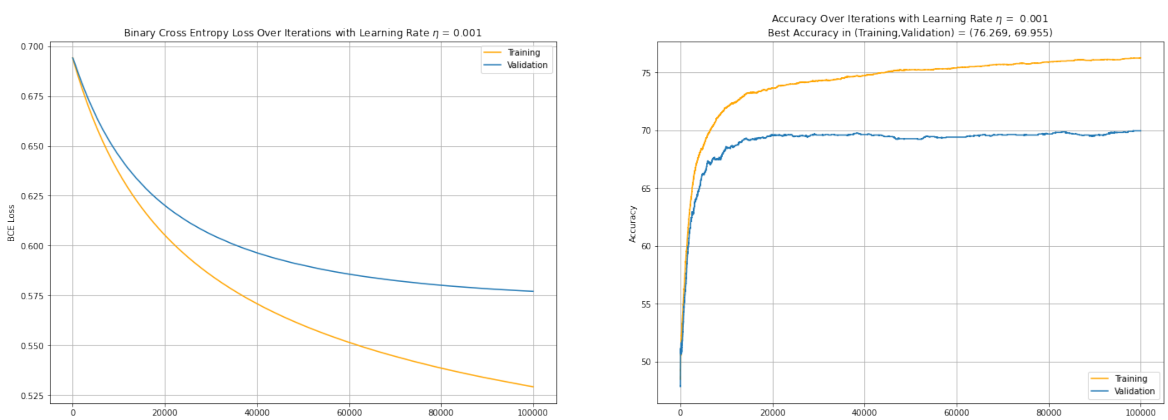

FIGURE 5. The visualization of the training & testing loss and accuracy of Logistic Regression-II

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

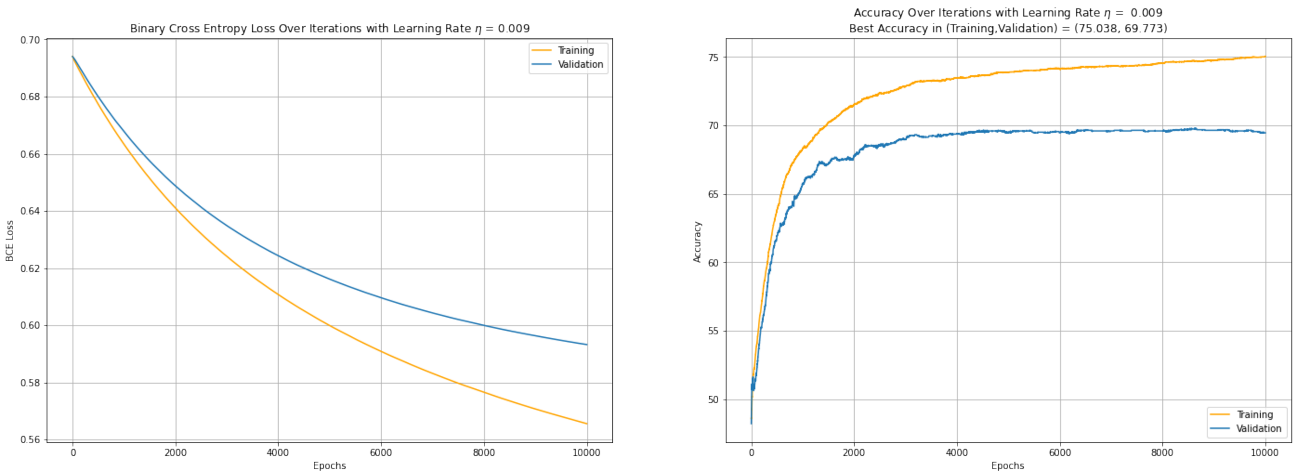

FIGURE 6. The visualization of the training & testing loss and accuracy of Logistic Regression-III

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.

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

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.

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

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 $1 - S_C(A, B)$.

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

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	Barış Kıçıman	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 Barış Kıçıman	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 Barış Kıçıman	0%

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

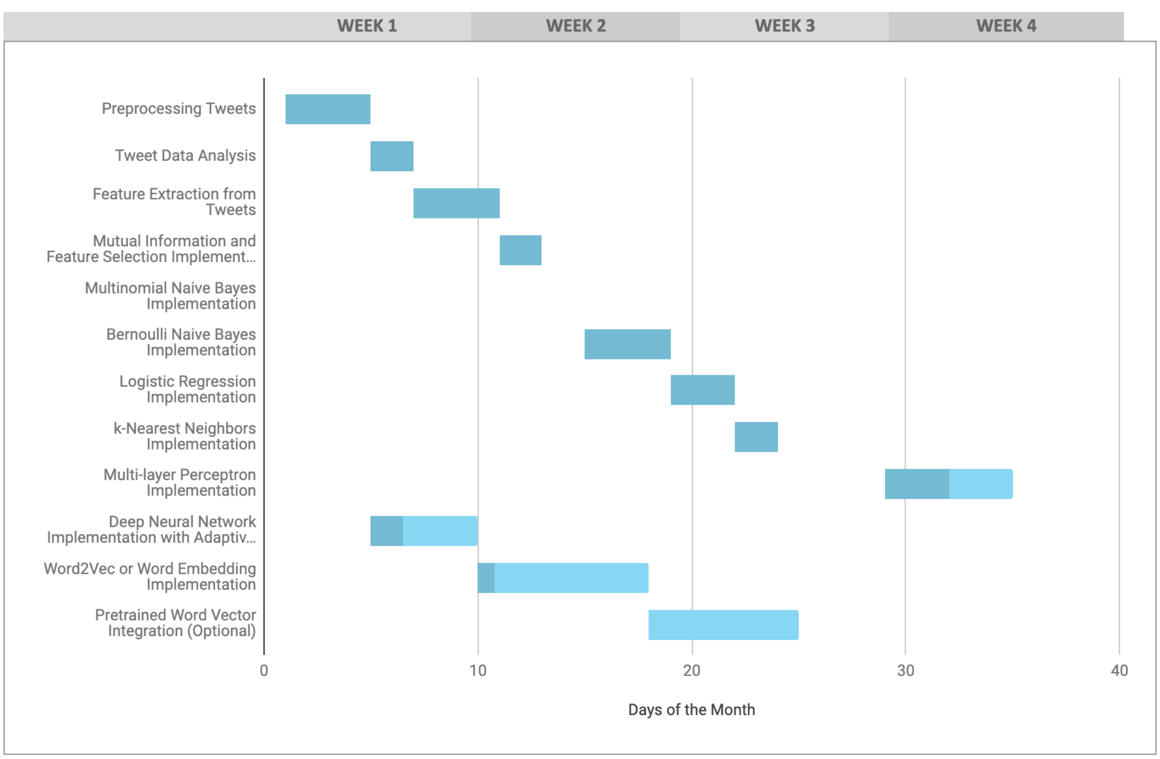

FIGURE 8. The visualization of the work packages and Gantt chart

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

```
from __future__ import print_function, division

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import warnings
import random
import pickle
import json
import time
import abc
import os

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)
from collections import OrderedDict

def mutual_information(
    x1:np.ndarray,
    x2:np.ndarray
) -> np.float:

    jh = np.histogram2d(
        x1,
        x2,
        bins = (
            256, 256
        )
    )[0]

    jh = jh + 1e-15

    sh = np.sum(jh)
    jh = jh / sh

    y1 = np.sum(
        jh,
        axis=0
    ).reshape(
        ( -1, jh.shape[0])
    )

    y2 = np.sum(
        jh,
        axis=1
    ).reshape(
        (jh.shape[1], -1)
    )


```

```
    return  (
        np.sum(jh * np.log(jh)) - np.sum(y1 * np.log(y1)) - np.sum(y2 * np.log(y2))
    )


class BackwardElimination:
    def __init__(self, pipeline):
        self.pipe = pipeline
        self._cache = []

    def fit(
        self,
        X_train,
        y_train,
        X_test,
        y_test,
        feature_subset,
        verbose=0,
    ):

        max_iteration_number = len(feature_subset) + 1
        features = feature_subset.copy()
        accuracies = [0.0]


        for iteration in range(1, max_iteration_number):


            candidate_feature = features[iteration - 1]

            if verbose: print(f"Candidate feature to be eliminated is {candidate_feature}")

            feature_subset.remove(candidate_feature)


            if verbose: print(f"Remaining features: {feature_subset}")

            since = time.time()

            self.pipe.fit(
                X_train = X_train[feature_subset].values,
                y_train = y_train.values
            )


            score = self.pipe.score(
                X_test = X_test[feature_subset].values,
                y_test = y_test.values
            )

            time_passed = time.time() - since
            accuracies.append(
                score['accuracy']
            )

            self._cache.append(
                (
                    "-".join(
```

```
                        feature_subset
                    ),
                    score['accuracy'],
                    round(float(time_passed), 4)
                )
            )


            if accuracies[iteration] < accuracies[iteration - 1]:
                feature_subset.append(candidate_feature)

                if verbose: print(f"Candidate feature is restored :
                ↪  {candidate_feature}")



        self._best_features = feature_subset
        self._scores = accuracies

        return self

    def best_features(self):
        return self._best_features

    def cache(self):
        return self._cache

    def scores(self):
        return self._scores

from __future__ import (
    print_function,
    division
)

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import warnings
import random
import pickle
import json
import time
import abc
import os

from collections import OrderedDict

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)

from utils import (
    Classifier,
```

```
    Pipeline,
    Vocabulary,
    json_print,
    timeit,
    random_seed
)


class KNeighborsClassifier(Classifier):
    """
        K-NeighborsClassifier based on geometric distance metrics

    """
    def __init__(
        self,
        k_neighbors:int = 9,
        distance_metric:str = "euclidean"
    ):
        """
            Args:

                k_neighbors: int
                    - Number of Neighbors (Default = 9)
                distance_metric: str
                    - Distance metric (Default = "euclidean")
                    - Available metrics = [euclidean, manhattan, cosine]
        """
        super(KNeighborsClassifier, self).__init__()

        self.k_neighbors = k_neighbors
        self.distance_metric = distance_metric


        self._hyperparams['k_neighbors'] = self.k_neighbors
        self._hyperparams['distance_metric'] = self.distance_metric
        self._name = 'K-Nearest Neighbors'

    def fit(self, X_train, y_train, *fit_params):
        self.X_train, self.y_train = X_train, y_train

        return self

    def euclidean_distance(self, x1, x2):
        return np.sqrt(
            np.einsum(
                'ij,ij->i...',
                x1 - x2,
                x1 - x2
            )
        )

    def manhattan_distance(self, x1, x2):
        return np.linalg.norm(
            x1 - x2,
            axis = 1,
            ord = 1
        )

    def cosine_distance(self, x1, x2):
```

```
        y = np.einsum(
            'ij,ij->i',
            x2,
            x2
        )
        x = np.einsum(
            'ij,ij->i',
            x1,
            x1
        )[:, np.newaxis]

        sumxy = x1 @ x2.T
        return 1 - (
            sumxy / np.sqrt(x)
        ) / np.sqrt(y)


    def predict(self, X_test):

        if self.distance_metric == "euclidean":
            distances = np.array([
                self.euclidean_distance(x_test, self.X_train) for x_test in X_test
            ])

        elif self.distance_metric == "manhattan":
            distances = np.array([
                self.manhattan_distance(x_test, self.X_train) for x_test in X_test
            ])

        elif self.distance_metric == "cosine":
            distances = self.cosine_distance(
                X_test,
                self.X_train
            )


        sorted_neighbors = distances.argsort(axis=1)[...,: self.k_neighbors]
        nearest_labels = self.y_train[sorted_neighbors]

        predictions = np.apply_along_axis(
            lambda x: np.bincount(x).argmax(),
            axis = 1,
            arr = nearest_labels
        )

        return predictions

class MultiNominalNaiveBayes(Classifier):
    def __init__(self, alpha=0.0001):
        super(MultiNominalNaiveBayes, self).__init__()
        self.alpha = alpha

        self._hyperparams['alpha'] = self.alpha
        self._name = 'MultiNominal NaiveBayes Classifier'

    @timeit
    def fit(
        self,
        X_train:pd.DataFrame,
```

```python
        y_train: pd.DataFrame,
        **fit_params
    ):

        m, n = X_train.shape
        self.classes = np.unique(y_train)
        n_classes = len(self.classes)

        if not isinstance(X_train, pd.DataFrame):
            X_train = pd.DataFrame(X_train)

        self.priors = y_train.value_counts(normalize = True).values
        self.counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
        likelihoods = self.counts.T / self.counts.sum(1).values.reshape(-1,  n_classes) + self.alpha
        self.likelihoods = likelihoods.values #.T
        self.log_priors = np.log(self.priors)

        return self

    @timeit
    def predict(self, X_test):

        if isinstance(X_test, pd.DataFrame):
            X_test = X_test.values

        self.log_likelihoods = X_test @ np.log(self.likelihoods) #(np.log(self.likelihoods) @ X_test.T).T
        self.posteriors = self.log_likelihoods + self.log_priors

        return self.classes[
            self.posteriors.argmax(1)
        ]

    def posteriors(self):
        return self.posteriors

class BernaulliNaiveBayes(Classifier):
    def __init__(self, alpha = 0.001):
        super(BernaulliNaiveBayes, self).__init__()
        self.alpha = alpha

        self._hyperparams['alpha'] = self.alpha
        self._name = 'Bernaulli NaiveBayes Classifier'

    @timeit
    def fit(self, X_train, y_train, **fit_params):
        self.classes = np.unique(y_train)

        n_classes = len(self.classes)

        if not isinstance(X_train, pd.DataFrame):
            X_train = pd.DataFrame(X_train)

        self.priors = y_train.value_counts(normalize = True).values
        self.log_priors = np.log(self.priors)

        counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
```

```python
        likelihoods = counts.T / counts.sum(1).values.reshape(-1,  n_classes) +
        ↪  self.alpha
        self.likelihoods = likelihoods.T.values

        return self

    @timeit
    def predict(self, X_test):

        self.posteriors = np.array(
            [
                (
                    (np.log(self.likelihoods) * x) + (np.log(1 - self.likelihoods) *
                    ↪  np.abs(x - 1))
                ).sum(axis = 1) + self.log_priors for x in X_test
            ]

        )

        return self.classes[
            self.posteriors.argmax(1)
        ]

class LogisticRegression(Classifier):
    def __init__(self):
        super().__init__()

    def init_params(self,
        input_shape:int,
        output_shape:int = 1
    ):
        self.__random_seed()

        #assert self.X_train.shape[1] == self.X_test.shape[1], 'Improper feature
        ↪  dimension!'

        W_high = self.__init_xavier(input_shape, output_shape)
        W_low  = - W_high
        W_size = (input_shape, output_shape)
        B_size = (1, output_shape)

        self.W = np.random.uniform(
            W_low,
            W_high,
            size = W_size
        )

        self.b = np.random.uniform(
            W_low,
            W_high,
            size = B_size
        )


    def __random_seed(self, seed = 32):
        """ Random seed for reproducebility """
        random.seed(seed)
        np.random.seed(seed)

```

```python
    def __init_xavier(self, L_pre, L_post):
        """ 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))]  """
        return np.sqrt(6/(L_pre + L_post))

    def __train_config(self,
        lr:float,
        batch_size:int,
        epochs:int,
    ):
        self.lr = lr
        self.batch_size = batch_size
        self.epochs = epochs

    def sigmoid(self,X, grad = False):
        """ Computing sigmoid and it's gradient w.r.t. it's input """
        sig = 1/(1 + np.exp(-X))

        return sig * (1-sig) if grad else sig

    def __forward(self, X):

        Z = (X @ self.W)  + self.b
        A = self.sigmoid(Z)

        return {
            "Z": Z,
            "A": A
        }


    def __SGD(self, grads):
        self.W -= self.lr * grads['W']
        self.b -= self.lr * grads['b']


    def matrix_back_prop(self, outs, X, Y):
        """ Matrix form backward propagation """
        m = self.batch_size

        Z = outs['Z']
        A = outs['A']

        dZ = (A-Y) * self.sigmoid(Z, grad = True)
        dW = (1 / m) * (X.T @ dZ)
        db = (1 / m) * np.sum(dZ, axis=0, keepdims=True)

        assert self.W.shape == dW.shape, f'Error in weight shapes!, {dW.shape} does not match with {self.W.shape}'
        assert self.b.shape == db.shape, f'Error in bias shapes!, {db.shape} does not match with {self.b.shape}'

        grads = {}
        grads['W'] = dW
        grads['b'] = db

        return grads

```

```
    def backward(self,
        outs,
        X,
        Y
    ):
        return self.matrix_back_prop(
            outs,
            X,
            Y
        )


    def BinaryCrossEntropyLoss(self, pred, label):
        m = pred.shape[0]
        preds = np.clip(pred, 1e-16, 1 - 1e-16)
        loss = np.sum(-label * np.log(preds + 1e-20) - (1 - label) * np.log(1 - preds + 1e-20))
        return loss / m

    def eval(self, x, y, knob:float = 0.5):
        predictions = self.__forward(x)
        predictions = predictions['A']
        predictions[predictions>=knob] = 1
        predictions[predictions< knob] = 0
        acc_score = self.accuracy(predictions, y)

        return acc_score

    def __accuracy(self,pred,label):
        return np.sum(pred == label) / pred.shape[0]


    @timeit
    def fit(
        self,
        X_train,
        y_train,
        X_test,
        y_test,
        lr:float = 1e-2,
        batch_size:int = 32,
        epochs:int = 100,
        verbose = True
    ):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        self.init_params(
            input_shape = X_train.shape[1]
            )

        self.__train_config(
            lr,
            batch_size,
            epochs,
```

```
        )

        self.history = {}

        self.history['train'] = {
            'loss': [],
            'acc' : []
        }

        self.history['val'] =   {
            'loss': [],
            'acc' : []
        }

        m = self.batch_size

        self.sample_size_train = X_train.shape[0]

        for epoch in range(self.epochs):

            perm = np.random.permutation(self.sample_size_train)

            for i in range(self.sample_size_train // m):


                shuffled_index = perm[i*m: (i+1)*m]

                X_feed = X_train[shuffled_index]
                y_feed = y_train[shuffled_index]

                outs = self.__forward(X_feed)
                grads = self.backward(
                    outs,
                    X_feed,
                    y_feed
                )
                self.__SGD(grads)

            loss_train = self.BinaryCrossEntropyLoss(
                self.__forward(X_train)['A'],
                y_train
            )

            acc_train = self.eval(
                X_train,
                y_train
            )

            self.history['train']['loss'].append(loss_train)
            self.history['train']['acc'].append(acc_train)



            loss_val = self.BinaryCrossEntropyLoss(
                self.__forward(X_test)['A'],
                y_test
            )

            acc_val  = self.eval(
```

```
                X_test,
                y_test
            )

            self.history['val']['loss'].append(loss_val)
            self.history['val']['acc'].append(acc_val)

            if verbose:
                print(f"[{epoch}/{self.epochs}] ------> Training : BCE: {loss_train} and Acc: {acc_train}")
                print(f"[{epoch}/{self.epochs}] ------> Testing  : BCE: {loss_val}   and Acc: {acc_val}")


    def __str__(self):
        model = LogisticRegression().__class__.__name__
        model += f' with hyperparameters (learning rate,batch_size,epochs) = ({self.lr},{self.batch_size},{self.epochs})'
        num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] * self.b.shape[1]
        model += f'\n There are {num_params} number of traniable parameters'
        return model

    def __repr__(self):
        model = LogisticRegression().__class__.__name__
        model += f' with hyperparameters (learning rate,batch_size,epochs) = ({self.lr},{self.batch_size},{self.epochs})'
        num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] * self.b.shape[1]
        model += f'\n There are {num_params} number of traniable parameters'
        return model

    def plot_history(self):

        fig,axs = plt.subplots(1,2,figsize = (24,8))
        axs[0].plot(self.history['train']['loss'],color = 'orange',label = 'Training')
        axs[0].plot(self.history['val']['loss'], label = 'Validation')
        axs[0].set_xlabel('Epochs')
        axs[0].set_ylabel('BCE Loss')
        axs[0].set_title(f'Binary Cross Entropy Loss Over Iterations with Learning Rate $\eta$ = {self.lr}')
        axs[0].legend(loc="upper right")
        axs[0].grid()

        axs[1].plot(self.history['train']['acc'],color ='orange',label = 'Training')
        axs[1].plot(self.history['val']['acc'], label = 'Validation')
        axs[1].set_xlabel('Epochs')
        axs[1].set_ylabel('Accuracy')
        maxs = round(max(self.history['train']['acc']),3), round(max(self.history['val']['acc']),3)
        axs[1].set_title(f'Accuracy Over Iterations with Learning Rate $\eta$ = {self.lr} \n Best Accuracy in (Training,Validation) = {maxs} ')
        axs[1].legend(loc="lower right")
        axs[1].grid()


class MLP(Classifier):

```

```
    def __init__(self,
        input_size = X_train.shape,
        batch_size = 19 ,
        n_neurons = 76 ,
        mean = 0,
        std = 1,
        lr = 1e-1,
        distribution = 'Xavier'
    ):


        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
        self.batch_size = batch_size
        self.n_neurons = n_neurons
        self.mean, self.std = mean, std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.n_neurons
        self.W2_size = self.n_neurons,1

        self.B1_size = 1,self.n_neurons
        self.B2_size = 1, 1

        self.B1 = np.random.normal(loc = self.mean, scale = self.std, size = (self.B1_size)) * 0.01
        self.B2 = np.random.normal(loc = self.mean, scale = self.std, size = (self.B2_size)) * 0.01

        self.he_scale1 = np.sqrt(2/self.feature_size)
        self.he_scale2 = np.sqrt(2/self.n_neurons)
        self.xavier_scale1 = np.sqrt(2/(self.feature_size+self.n_neurons))
        self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))

        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))

        elif (self.dist == 'Gauss'):
            self.W1 = np.random.normal(loc = self.mean, scale = self.std, size = (self.W1_size))* 0.01
            self.W2 = np.random.normal(loc = self.mean, scale = self.std, size = (self.W2_size))* 0.01

        elif (self.dist == 'He'):
            self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) * self.he_scale1
            self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) * self.he_scale2

```

```
        elif (self.dist == 'Xavier'):

            self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) * self.xavier_scale1
            self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) * self.xavier_scale2



    def forward(self,X):

        Z1 = (X @ self.W1)  + self.B1
        A1 = np.tanh(Z1)
        Z2 = (A1 @ self.W2) + self.B2
        A2 = np.tanh(Z2)

        return {
            "Z1": Z1,
            "A1": A1,
            "Z2": Z2,
            "A2": A2
        }


    def tanh(self,X):
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def tanh_der(self,X):
        return  1-(np.tanh(X)**2)

    def backward(self,outs, X, Y):
        m = (self.batch_size)

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        dZ2 = (A2-Y)* self.tanh_der(Z2)
        dW2 = (1/m) * (A1.T @ dZ2)
        dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

        dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
        dW1 = (1/m) * (X.T @ dZ1)
        dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)


        return {
            "dW1": dW1,
            "dW2": dW2,
            "dB1": dB1,
            "dB2": dB2
        }

    def Loss(self,pred, y_true, knob = 0):

        mse = np.square(pred-y_true).mean()

        pred[pred>=knob]=1
```

```
        pred[pred<knob]=-1

        mce = (pred == y_true).mean()

        return {
            'MSE':mse,
            'MCE':mce
        }


    def SGD(self,grads):
        self.W1 -= self.lr * grads['dW1']
        self.W2 -= self.lr * grads['dW2']
        self.B1 -= self.lr * grads['dB1']
        self.B2 -= self.lr * grads['dB2']

    def fit(self,X,Y,X_test,y_test,epochs = 300,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        m = self.batch_size

        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)

            for i in range(self.n_update):


                batch_start = i * m
                batch_finish = (i+1) * m
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(
                    outs['A2'],
                    y_feed
                )

                outs_test = self.forward(X_test)
                loss_test = self.Loss(
                    outs_test['A2'],
                    y_test
                )

                grads = self.backward(
                    outs,
                    X_feed,
                    y_feed
                )

                self.SGD(grads)

```

```python
            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE: {loss['MSE']} and MCE: {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing :MSE: {loss_test['MSE']} and MCE: {loss_test['MCE']}")

    def history(self):
        return {
            'Train_MSE' : self.mse_train,
            'Train_MCE' : self.mce_train,
            'Test_MSE'  : self.mse_test,
            'Test_MCE'  : self.mce_test
        }

from __future__ import print_function, division

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import warnings
import random
import pickle
import json
import time
import abc
import os

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)
from collections import OrderedDict


def save_obj(
    obj:object,
    path:str = None
) -> None:
    """ Saves Python Object as pickle"""
    with open(path + '.pkl', 'wb') as f:
        pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


def load_obj(
    path:str = None
) -> object:
    """ Loads Python Object from pickle"""
    with open(path + '.pkl', 'rb') as f:
        return pickle.load(f)

```

```python
class ClassifierCharacteristics:

    def confusion_matrix(
        self,
        preds: Iterable[list or np.ndarray],
        labels: Iterable[list or np.ndarray]
    ) -> pd.DataFrame:

        """Given the labels and predictions, calculate confusion matrix. """
        label = pd.Series(
            labels,
            name = 'Actual'
        )
        pred = pd.Series(
            preds,
            name = 'Predicted'
        )

        return pd.crosstab(
            label,
            pred
        )

    def accuracy(
        self,
        preds: Iterable[list or np.ndarray],
        labels: Iterable[list or np.ndarray],
        scale:bool = True
    ) -> np.float:
        """Given the labels and predictions, calculate accuracy score. """
        return np.mean(preds == labels) * 100 if scale else np.mean(preds == labels)

    def visualize_confusion_matrix(
        self,
        data:Iterable[list or np.ndarray],
        normalize:bool = True,
        title:str = " ",
    ) -> None:

        if normalize:
            data /= np.sum(data)

        plt.figure(
            figsize = (
                15, 15
            )
        )
        sns.heatmap(
            data,
            fmt='.2%',
            cmap = 'Greens'
        )

        plt.title(title)
        plt.show()

    @staticmethod
    def timeit(
        Func:Callable
```

```python
    ):
        """ Calculate time spend of the function

        Usage:
            >>  @timeit
            >>  def func(x):
            >>      return x
        """
        def _timeStamp(*args, **kwargs):
            since = time.time()
            result = Func(*args, **kwargs)
            time_elapsed = time.time() - since

            if time_elapsed > 60:
                print('Time Consumed : {:.0f}m {:.0f}s'.format(time_elapsed // 60,
                ↪  time_elapsed % 60))
            else:
                print('Time Consumed : ' , round((time_elapsed), 4) , 's')
            return result
        return _timeStamp

    @staticmethod
    def random_seed(
        Func:Callable,
        seed:int = 42
    ):
        """
        Decorator random seed.

        Usage:
            >>  @random_seed
            >>  def func(*args):
            >>      return [arg for arg in args]
        """
        def _random_seed(*args, **kwargs):
            np.random.seed(seed)
            random.seed(seed)
            result = Func(
                *args,
                **kwargs
            )
            return result
        return _random_seed

    def save_obj(
        self,
        obj:object,
        path:str = None
    ) -> None:
        """ Saves Python Object as pickle"""
        with open(path + '.pkl', 'wb') as f:
            pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


    def load_obj(
        self,
        path:str = None
    ) -> object:
        """ Loads Python Object from pickle"""
```

```python
        with open(path + '.pkl', 'rb') as f:
            return pickle.load(f)


    def save_numpy(
        self,
        data:Iterable[list or np.ndarray] = None,
        path:str = None
    ) -> None:
        """ Saves NumPy array or Python object as .npy"""
        np.save(
            path + '.npy',
            data,
            allow_pickle=True
        )

    def load_numpy(
        self,
        path:str = None
    ) -> np.ndarray:
        """ Loads NumPy array or Python object as .npy"""
        return np.load(
            path + '.npy',
            allow_pickle=True
        )

class Classifier(ClassifierCharacteristics):
    #__metaclass__ = abc.ABCMeta

    def __init__(self):
        super(Classifier, self).__init__()
        self._hyperparams = {}
        self._scores = {}
        self._name = ""
        self._params = OrderedDict()


    def save(self, filename: str) -> None:
        self.save_obj(
            self.__dict__,
            filename
        )

    def load(self, filename: str) -> None:
        self.__dict__ = self.load_obj(filename)


    @ClassifierCharacteristics.timeit
    def fit(
        self,
        X_train: Union[np.ndarray, pd.DataFrame],
        y_train: Union[np.ndarray, pd.DataFrame],
        *fit_params
    ) -> None:
        return NotImplementedError()



    @ClassifierCharacteristics.timeit
```

```python
    def predict(
        self,
        X_test: Union[np.ndarray, pd.DataFrame]
    ) -> Union[pd.DataFrame, np.ndarray]:
        return NotImplementedError()



    @ClassifierCharacteristics.timeit
    def fit_predict(
        self,
        X_train: Union[np.ndarray, pd.DataFrame],
        y_train: Union[np.ndarray, pd.DataFrame],
        X_test: Union[np.ndarray, pd.DataFrame],
        *fit_params
    ) -> Union[pd.DataFrame, np.ndarray]:

        self.fit(X_train, y_train, *fit_params)

        return self.predict(X_test)

    def score(
        self,
        X_test: Union[np.ndarray, pd.DataFrame],
        y_test: Union[np.ndarray, pd.DataFrame],
        metric_list: List[Callable] = []
    ) -> np.float:

        predictions = self.predict(X_test)
        accuracy = self.accuracy(predictions, y_test)


        self._scores['accuracy'] = accuracy

        if len(metric_list) != 0:
            for metric in metric_list:
                self._scores[metric.__name__] = metric(predictions, y_test)

        return self._scores


    def params(self) -> OrderedDict:
        return self._params


    def hyperparameters(self):
        return self._hyperparams


    def name(self) -> str:
        return self._name


    def __str__(self)-> str:
        return f"{self._name} with hyperparameters {json.dumps(self._hyperparams,
        ↪  sort_keys=True, indent=4)}"


    def __repr__(self)-> str:
```

```python
        return f"{self._name} with hyperparameters {json.dumps(self._hyperparams, sort_keys=True, indent=4)}"


class Pipeline:
    """ Generic ML Operation Pipeline """
    def __init__(
        self,
        pipeline: List[Tuple[str, object]] = []
    ):
        super(Pipeline, self).__init__()
        self.pipeline = pipeline
        self.model = None
        self._name = 'ML Pipeline'
        self._scores = {}

    def fit(
        self,
        X_train: Union[np.ndarray, pd.DataFrame],
        y_train: Union[np.ndarray, pd.DataFrame],
        verbose: int = 0,
        *fit_params
    ) -> None:

        yield_data = X_train.copy()

        for pipe_name, pipe_op in self.pipeline:

            if verbose: print(f"{pipe_name} operation is applying")

            if issubclass(pipe_op.__class__, Classifier):

                self.model = pipe_op
                self.model.fit(
                    yield_data,
                    y_train,
                    *fit_params
                )

            elif isinstance(pipe_op.__class__, FeatureEngineer):
                yield_data = pipe_op.fit_transform(yield_data)

            else:
                raise Exception(f"{pipe_name} operator could not be decoded!")


    def predict(
        self,
        X_test: Union[np.ndarray, pd.DataFrame]
    ) -> Union[pd.DataFrame, np.ndarray]:
        return self.model.predict(X_test)


    def score(
        self,
        X_test: Union[np.ndarray, pd.DataFrame],
        y_test: Union[np.ndarray, pd.DataFrame],
        metric_list: List[Callable] = []
    ) -> np.float:
```

```python

        predictions = self.model.predict(X_test)
        accuracy = self.model.accuracy(predictions, y_test)


        self._scores['accuracy'] = accuracy

        if len(metric_list) != 0:
            for metric in metric_list:
                self._scores[metric.__name__] = metric(predictions, y_test)

        return self._scores

    def name(self) -> str:
        return self._name


    def __str__(self) -> str:
        return "\n".join([
            str(pape_op) for _, pape_op in self.pipeline
        ])
    def __repr__(self) -> str:
        return "\n".join([
            str(pape_op) for _, pape_op in self.pipeline
        ])



class Vocabulary:
    def __init__(
        self,
        root_dir:str,
        filename:str,
        delimiter: str = '\n'

    ):
        super(Vocabulary, self).__init__()
        self.vocab = open(
            os.path.join(
                root_dir,
                filename
            )
        ).read()

        self.list_vocab = self.vocab.split(delimiter)[:-1]

        self.word2id = {
            word: i for i, word in enumerate(self.list_vocab)
        }

        self.id2word = {
            i: word for word, i in self.word2id.items()
        }

    def __getitem__(self, idx):

        if isinstance(idx, (list, np.ndarray)):
            return [self.id2word[i] for i in idx]

```

```python
        return self.id2word[idx]

    def __len__(self):
        return len(self.list_vocab)

    def get_vocab(self):
        return self.word2id


def json_print(data):
    return json.dumps(
        data,
        sort_keys=True,
        indent=4
    )



def timeit(
    Func:Callable
):
    """ Calculate time spend of the function

    Usage:
        >>  @timeit
        >>  def func(x):
        >>      return x
    """
    def _timeStamp(*args, **kwargs):
        since = time.time()
        result = Func(*args, **kwargs)
        time_elapsed = time.time() - since

        if time_elapsed > 60:
            print('Time Consumed for {}: {:.0f}m {:.0f}s'.format(Func.__name__, time_elapsed // 60, time_elapsed % 60))
        else:
            print(f'Time Consumed for {Func.__name__}: {round((time_elapsed), 4)} s')
        return result
    return _timeStamp

def random_seed(
    Func:Callable,
    seed:int = 42
):
    """
    Decorator random seed.

    Usage:
        >>  @random_seed
        >>  def func(*args):
        >>      return [arg for arg in args]
    """
    def _random_seed(*args, **kwargs):
        np.random.seed(seed)
        random.seed(seed)
        result = Func(
            *args,
            **kwargs
```

```python
        )
        return result
    return _random_seed

# %%
from __future__ import (
    print_function,
    division
)

import matplotlib.pyplot as plt
import stopwordsiso as swiso
import seaborn as sns
import pandas as pd
import numpy as np
import cleantext
import warnings
import random
import string
import pickle
import spacy
import json
import nltk
import time
import abc
import os
import re
import sys
sys.path.append('./src')

from spellchecker import SpellChecker
from stop_words import get_stop_words
from collections import OrderedDict
from nltk.stem import PorterStemmer
from collections import Counter
from nltk.corpus import stopwords
from wordcloud import WordCloud


from textblob import (
    TextBlob,
    Word
)

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)

from utils import (
    Classifier,
    Pipeline,
    Vocabulary,
    json_print,
    timeit,
    random_seed
```

```python
)
from supervised import (
    KNeighborsClassifier,
    MultiNominalNaiveBayes,
    BernaulliNaiveBayes
)
from feature import (
    BackwardElimination,
    #mutual_information
)



# %%
cols = [
    'date',
    'id',
    'text',
    'query_string',
    'user',
    'sentiment',
]

df = pd.read_csv(
    '../data/training.1600000.processed.noemoticon.csv',
    encoding=  'latin1',
    names = [
        'sentiment',
        'id',
        'date',
        'query_string',
        'user',
        'text'
    ]
)

df = df[cols]

# %%
df.head()

# %%
df.info()

# %%
df.set_index('id', inplace=True)
df.drop(
    columns= [
        'date', 'query_string'
    ],
    axis=1,
    inplace=True
)

# %%
df.head(7)

# %%
df['sentiment'].value_counts(
```

```
    normalize = True
    ).plot.bar(
        title = 'Sentiment Distribution',
        xlabel = ['Neg', 'Pos']
)

# %%
print(f"Max length of the tweet {df.text.str.len().max()}")
print(f"Min length of the tweet {df.text.str.len().min()}")
print(f"Average length of the tweet {df.text.str.len().mean()}")
print(f"Median length of the tweet {df.text.str.len().median()}")

# %%
len_stats = pd.DataFrame(df.text.str.len().describe())
len_stats.col = 'tweet_len_stats'
len_stats

# %%
df['sentiment'] = df['sentiment'].replace(
    {
        4:1
    }
)

df['sentiment'].sample(5)

# %%
class Cleaner:
    def __init__(self, operations = []):
        self.operations = operations

    def __call__(self, text):
        for operation in self.operations:
            text = operation(text)
        return text

def lower_case(text):
    return text.lower()

def remove_mentiones_and_hashtag(text):
    return re.sub('@[^ ]+|#[^ ]+', '', text)

def remove_punctuation(text):
    return text.replace('[^A-Za-z0-9 ]', "")

def remove_punctuations_layer_2(text):
    punct = re.compile(r'[^\w\s]')

    return punct.sub(r'',text)

def remove_stop_words(text):
    #stop_words = get_stop_words()
    return " ".join(word for word in text.split() if word not in stop_words)


def remove_emoji(text):
    """ Reference : https://gist.github.com/slowkow/7a7f61f495e3dbb7e3d767f97bd7304b"""
    emoji_pattern = re.compile("["
                        u"\U0001F600-\U0001F64F"  # emoticons
```

```python
                          u"\U0001F300-\U0001F5FF"  # symbols & pictographs
                          u"\U0001F680-\U0001F6FF"  # transport & map symbols
                          u"\U0001F1E0-\U0001F1FF"  # flags
                          u"\U00002702-\U000027B0"
                          u"\U000024C2-\U0001F251"
                          "]+",
                          flags=re.UNICODE)
    return emoji_pattern.sub(r'', text)


def remove_HTML(text):
    tag = re.compile(r'<.*?>')

    return tag.sub(r'',text)

def remove_URL(text):
    url = re.compile(r'https?://\S+|www\.\S+')

    return url.sub(r'',text)

def get_stop_words_(all_languages:bool = False):
    stop_words_1 = stopwords.words('english')
    stop_words_2 = get_stop_words('english')
    stop_words_3 = list(swiso.stopwords('en'))


    stop_words = stop_words_1 + stop_words_2 + stop_words_3

    if all_languages:
        stop_words_4 = list(
            swiso.stopwords(
                swiso.langs()
            )
        )
        stop_words = stop_words + stop_words_4

    stop_words = set(stop_words)

    return stop_words

def stemmization(text):
    st = PorterStemmer()
    return " ".join([st.stem(word) for word in text.split()])

def lemmatization(text):
    return " ".join([Word(word).lemmatize() for word in text.split()])

def spell_check(text):
    return str(
        TextBlob(text).correct()
    )
def spell_check_layer_2(text):
    spell = SpellChecker()
    return spell.correction(text)


def spacy_lemmatization(text):
    nlp = spacy.load("en_core_web_sm")
    return " ".join([word.lemma_ for word in nlp(text)])
```

```

def final_cleaner(text):
    return cleantext.clean(text, all= True)

stop_words = get_stop_words_()

# %%
%%time
clean_text(
    df.text.iloc[0],
    operations = [
        remove_HTML,
        remove_URL,
        lower_case,
        remove_mentiones_and_hashtag,
        remove_punctuation,
        remove_punctuations_layer_2,
        remove_emoji,
        remove_stop_words,
        #spell_check,
        #spell_check_layer_2,
        stemmization,
        lemmatization,
        #spacy_lemmatization,
        final_cleaner
    ]
)

# %%
df['clean_text'] = df['text'].apply(
    Cleaner(
        operations = [
            remove_HTML,
            remove_URL,
            lower_case,
            remove_mentiones_and_hashtag,
            remove_punctuation,
            remove_punctuations_layer_2,
            remove_emoji,
            remove_stop_words,
            #spell_check,
            #spell_check_layer_2,
            stemmization,
            lemmatization,
            #spacy_lemmatization,
            final_cleaner
        ]
    )
)

df.to_csv('data/cleaned_preprocessed_data.csv')

# %%
df = pd.read_csv('../data/cleaned_preprocessed_data.csv')

cols = [
    'id',
    'text',
    'clean_text',
```

```
    'user',
    'sentiment',
]

df = df[cols]

df['clean_text'] = df['clean_text'].str.replace(",", '')
df['clean_text'] = df['clean_text'].str.replace(".", '')
df['clean_text'] = df['clean_text'].str.strip()

df.drop(
    ['id'],
    axis = 1,
    inplace = True
)

# %%
neg_tweets = df.loc[df['sentiment']==0]
pos_tweets = df.loc[df['sentiment']==1]

# %%
neg_string = neg_tweets['clean_text'].str.cat(sep=' ')
pos_string = pos_tweets['clean_text'].str.cat(sep=' ')

# %%
plt.figure(figsize=(12,10))
wordcloud_neg = WordCloud(max_font_size=200,
↪  background_color="white").generate(neg_string)
plt.imshow(wordcloud_neg, interpolation="bilinear")
plt.axis('off')
plt.title("Tweets with Negative sentiment")

# %%
plt.figure(figsize=(12,10))
wordcloud_pos = WordCloud(max_font_size=200,
↪  background_color="white").generate(neg_string)
plt.imshow(wordcloud_pos, interpolation="bilinear")
plt.axis('off')
plt.title("Tweets with Positive sentiment")

# %%
def remove_unknown_(text):
    s = SpellChecker()
    unknown_words = s.unknown(text.split())
    return " ".join(word for word in text.split() if word not in unknown_words)


# %%
df['unknown_removed_cleaned'] = df['clean_text'].apply(remove_unknown_)

# %%
df.to_csv('data/cleaned_preprocessed_unknown_removed_data.csv')


# %% [markdown]
# # Imports and Read Data

# %%
from __future__ import (
```

```
    print_function,
    division
)

import matplotlib.pyplot as plt
import stopwordsiso as swiso
import seaborn as sns
import pandas as pd
import numpy as np
import cleantext
import warnings
import random
import string
import pickle
import spacy
import json
import nltk
import time
import abc
import os
import re
import sys
sys.path.append('./src')

from spellchecker import SpellChecker
from stop_words import get_stop_words
from collections import OrderedDict
from nltk.stem import PorterStemmer
from collections import Counter
from nltk.corpus import stopwords
from wordcloud import WordCloud


from textblob import (
    TextBlob,
    Word
)

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)

from utils import (
    Classifier,
    Pipeline,
    json_print,
    timeit,
    random_seed,
    #save_obj
)
from supervised import (
    KNeighborsClassifier,
    MultiNominalNaiveBayes,
    BernaulliNaiveBayes
)
```

```python
from feature import (
    BackwardElimination,
    #mutual_information
)

# %%
df = pd.read_csv('../data/cleaned_preprocessed_data.csv')

# %% [markdown]
# # Post-preprocessing

# %%
cols = [
    'id',
    'text',
    'clean_text',
    'user',
    'sentiment',
]

df = df[cols]

df['clean_text'] = df['clean_text'].str.replace(",", '')
df['clean_text'] = df['clean_text'].str.replace(".", '')
df['clean_text'] = df['clean_text'].str.strip()

df.drop(
    ['id'],
    axis = 1,
    inplace = True
)


df['doc_count'] = df['clean_text'].apply(lambda t: len(
    str(t).split()
    )
)

df.drop(
    df[df['doc_count'] <= 2].index,
    inplace = True
)

# %%
df.head()

# %%
df.isna().sum()

# %% [markdown]
# # Construction of Corpus

# %%
df_vocab = df['clean_text'].str.split(expand=True).stack().value_counts().reset_index()
df_vocab.columns = [
    'word',
    'frequency'
]
df_vocab.head(10)
```

```

# %%
len(df_vocab)

# %%
word_freq = Counter(
    df['clean_text'].str.cat(
        sep = ' '
    ).split()
)

print(word_freq.most_common(10))

# %% [markdown]
# # Post-processing with Word Count Data

# %%
dict_freq = dict(word_freq)
freq_threshold = 1000

# %%
df['clean_freq_removed_text'] = df['clean_text'].apply(
    lambda text : " ".join(
            [
                word for word in text.split() if dict_freq[word] > freq_threshold
        ]
    )
)

# %%
df['doc_count_clean_freq_removed_text'] = df['clean_freq_removed_text'].apply(
    lambda t: len(
        str(t).split()
    )
)

df.drop(
    df[df['doc_count_clean_freq_removed_text'] <= 2].index,
    inplace = True
)

# %%
df_vocab_greq_removed =
↪  df['clean_freq_removed_text'].str.split(expand=True).stack().value_counts().reset_index()
df_vocab_greq_removed.columns = [
    'word',
    'frequency'
]

# %%
class Vocabulary:
    def __init__(
        self,
        vocab_dict,

    ):
        super(Vocabulary, self).__init__()

        assert 'word' in vocab_dict.keys() and 'frequency' in vocab_dict.keys()
```

```python

        self.vocab_dict = vocab_dict

        self.id2word = vocab_dict['word']
        self.frequency = vocab_dict['frequency']

        self.word2id = {
            word: i for i, word in enumerate(self.id2word)
        }

    def __getitem__(self, idx):

        if isinstance(idx, (list, np.ndarray)):
            return [self.id2word[i] for i in idx]

        return self.id2word[idx]

    def __str__(self):
        return json_print(self.id2word)

    def __repr__(self):
        return json_print(self.id2word)


    def __len__(self):
        return len(self.word2id)

    def get_vocab(self):
        return self.word2id

    def get_frequency_dict(self):
        return {self.id2word[i] : freq for i, freq in self.frequency.items()}

    def save(self, filename: str) -> None:
        self.save_obj(
            self.__dict__,
            filename
        )

    def load(self, filename: str) -> None:
        self.__dict__ = self.load_obj(filename)

    def save_obj(
        self,
        obj:object,
        path:str = None
    ) -> None:
        """ Saves Python Object as pickle"""
        with open(path + '.pkl', 'wb') as f:
            pickle.dump(obj, f, pickle.HIGHEST_PROTOCOL)


    def load_obj(
        self,
        path:str = None
    ) -> object:
        """ Loads Python Object from pickle"""
        with open(path + '.pkl', 'rb') as f:
            return pickle.load(f)
```

```python


vocab = Vocabulary(
    df_vocab.to_dict()
)


vocab_freq_removed = Vocabulary(
    df_vocab_greq_removed.to_dict()
)




# %%
vocab.save('data/vocabulary')
vocab_freq_removed.save('data/vocabulary_freq_removed')

# %%
df.to_parquet('data/final_training_data.parquet')


# %%
"""
  INFORMATION:

    Course     : EEE485/585
    Name       : Can Kocagil
    ID         : 21602218
    E-mail     : can.kocagil@ug.bilkent.edu.tr
    Assignment : Progress Report

"""

from __future__ import (
    print_function,
    division
)

import matplotlib.pyplot as plt
import stopwordsiso as swiso
import seaborn as sns
import pandas as pd
import numpy as np
import cleantext
import warnings
import random
import string
import pickle
import spacy
import json
import nltk
import time
import abc
import os
import re
import sys
sys.path.append('./src')

```

```python
from sklearn.utils.validation import (
    check_X_y,
    check_array
)

from sklearn.metrics import (
    confusion_matrix,
    precision_score
)

from sklearn.feature_extraction.text import (
    CountVectorizer,
    TfidfVectorizer
)

from sklearn.model_selection import train_test_split

from typing import (
    Callable,
    Iterable,
    List,
    Union,
    Tuple,
)

from utils import (
    Pipeline,
    Classifier,
    json_print,
    timeit,
    random_seed
)
from supervised import (
    KNeighborsClassifier,
    MultiNominalNaiveBayes,
    BernaulliNaiveBayes
)
from feature import (
    BackwardElimination,
    #mutual_information
)

# %%
df = pd.read_parquet('../data/final_training_data.parquet')

df = df.sample(10000)

# %% [markdown]
# # Feature Extraction

# %%
doc = df['clean_freq_removed_text']

vectorizer=  TfidfVectorizer(
    #max_features = 10000,
    ngram_range = (1, 1),
    analyzer='word',
    stop_words='english',
    norm='l2'
```

```python
)

vectorizer.fit(doc)

features = vectorizer.transform(doc)

# %%
features.shape

# %% [markdown]
# # Train and Test Split

# %%
concat_doc_count = False

if concat_doc_count:
    X = np.concatenate(
            [
                features.toarray(),
                df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
        ],
        axis = 1
    )

else:
    X = features.toarray()

y = df['sentiment']

# %%
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size = 0.22,
    random_state = 21
)

# %%
X_train  = pd.DataFrame(X_train)
X_test  = pd.DataFrame(X_test)
y_train = pd.DataFrame(y_train)
y_train.columns = ['class']

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# %% [markdown]
# # Modelling

# %% [markdown]
# ## Multinominal Naive Bayes

# %%
doc = df['clean_text']

vectorizer = CountVectorizer(
    max_features = 1000,
    ngram_range = (1, 1)
```

```
)
vectorizer.fit(doc)

features = vectorizer.transform(doc)

concat_doc_count = False

if concat_doc_count:
    X = np.concatenate(
            [
                features.toarray(),
                df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
        ],
        axis = 1
    )

else:
    X = features.toarray()

y = df['sentiment']


X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size = 0.22,
    random_state = 21
)

X_train  = pd.DataFrame(X_train)
X_test  = pd.DataFrame(X_test)
y_train = pd.DataFrame(y_train)
y_train.columns = ['class']
y_train = y_train.reset_index().drop('index', 1)

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# %%
class MultiNominalNaiveBayes(Classifier):
    def __init__(self, alpha=0.0001):
        super(MultiNominalNaiveBayes, self).__init__()
        self.alpha = alpha

        self._hyperparams['alpha'] = self.alpha
        self._name = 'MultiNominal NaiveBayes Classifier'

    @timeit
    def fit(
        self,
        X_train:pd.DataFrame,
        y_train: pd.DataFrame,
        **fit_params
    ):

        m, n = X_train.shape
        self.classes = np.unique(y_train)
        n_classes = len(self.classes)
```

```

        if not isinstance(X_train, pd.DataFrame):
            X_train = pd.DataFrame(X_train)

        self.priors = y_train.value_counts(normalize = True).values
        self.counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
        likelihoods = self.counts.T / self.counts.sum(1).values.reshape(-1,  n_classes) + self.alpha
        self.likelihoods = likelihoods.values #.T
        self.log_priors = np.log(self.priors)

        return self

    @timeit
    def predict(self, X_test):

        if isinstance(X_test, pd.DataFrame):
            X_test = X_test.values

        self.log_likelihoods = X_test @ np.log(self.likelihoods) #(np.log(self.likelihoods) @ X_test.T).T
        self.posteriors = self.log_likelihoods + self.log_priors

        return self.classes[
            self.posteriors.argmax(1)
        ]

    def posteriors(self):
        return self.posteriors

# %%
m = MultiNominalNaiveBayes().fit(X_train.values, y_train)
(m.predict(X_test) == y_test.values).mean()

# %%
pipe = Pipeline([
        ('classifier',
        MultiNominalNaiveBayes(
        )
    )
])


pipe.fit(
    X_train = X_train,
    y_train = y_train
)


score = pipe.score(
    X_test = X_test.values,
    y_test = y_test
)

predictions = pipe.predict(
    X_test = X_test.values,
)

conf_matrix = pipe.model.confusion_matrix(
```

```python
    predictions.reshape(-1, ),
    y_test.values.reshape(-1, ),
)

pipe.model.visualize_confusion_matrix(conf_matrix)

print(f"Accuracy Score: {score['accuracy']}")
print(f"Confusion Matrix: \n {conf_matrix}")
print(f"The number of parameters to be estimated: {pipe.model.priors.size + pipe.model.likelihoods.size - 1}")


# %% [markdown]
# ## Bernoulli Naive Bayes

# %%
class BernaulliNaiveBayes(Classifier):
    def __init__(self, alpha = 0.001):
        super(BernaulliNaiveBayes, self).__init__()
        self.alpha = alpha

        self._hyperparams['alpha'] = self.alpha
        self._name = 'Bernaulli NaiveBayes Classifier'

    @timeit
    def fit(self, X_train, y_train, **fit_params):
        self.classes = np.unique(y_train)

        n_classes = len(self.classes)

        if not isinstance(X_train, pd.DataFrame):
            X_train = pd.DataFrame(X_train)

        self.priors = y_train.value_counts(normalize = True).values
        self.log_priors = np.log(self.priors)

        counts = pd.concat([X_train, y_train], 1).groupby('class').agg('sum')
        likelihoods = counts.T / counts.sum(1).values.reshape(-1,  n_classes) + self.alpha
        self.likelihoods = likelihoods.T.values

        return self

    @timeit
    def predict(self, X_test):

        self.posteriors = np.array(
            [
                (
                    (np.log(self.likelihoods) * x) + (np.log(1 - self.likelihoods) * np.abs(x - 1))
                ).sum(axis = 1) + self.log_priors for x in X_test
            ]

        )

        return self.classes[
            self.posteriors.argmax(1)
        ]
```

```python

# %%
pipe = Pipeline([
        ('classifier',
        BernaulliNaiveBayes(
            alpha=0.00019
        )
    )
])


pipe.fit(
    X_train = np.where(X_train >= 1, 1, 0),
    y_train = y_train
)

score = pipe.score(
    X_test = np.where(X_test >= 1, 1, 0),
    y_test = y_test
)

conf_matrix = pipe.model.confusion_matrix(
    predictions,
    y_test.values
)

print(f"Accuracy Score: {score['accuracy']}")
print(f"Confusion Matrix: \n {conf_matrix}")

# %%
def mutual_information(
    x1:np.ndarray,
    x2:np.ndarray
) -> np.float:

    jh = np.histogram2d(
        x1,
        x2,
        bins = (
            256, 256
        )
    )[0]

    jh = jh + 1e-15

    sh = np.sum(jh)
    jh = jh / sh

    y1 = np.sum(
        jh,
        axis=0
    ).reshape(
        ( -1, jh.shape[0])
    )

    y2 = np.sum(
        jh,
        axis=1
    ).reshape(
```

```
        (jh.shape[1], -1)
    )


    return  (
        np.sum(jh * np.log(jh)) - np.sum(y1 * np.log(y1)) - np.sum(y2 * np.log(y2))
    )


# %%
mutual_infos = []
feature_dimension = X_train.shape[1]

for feature in range(feature_dimension):

    mutual_infos.append(
        mutual_information(
            X_train.values[:, feature].reshape(-1, ),
            y_train.values.reshape(-1, )
        )
    )

mutual_infos = np.array(mutual_infos)


# %%
sorted_mutual_infos = np.flip(
    mutual_infos.argsort(
        axis = 0
    )
)

cache = []
selected_features = []


for iteration in range(0, 1000, 100):

    features = sorted_mutual_infos[iteration: iteration + 100]

    selected_features.extend(
        features
    )


    pipe = Pipeline([
            ('classifier',
            BernaulliNaiveBayes(
                alpha=0.00019
            )
        )
    ])

    since = time.time()

    pipe.fit(
        X_train = np.where(X_train >= 1, 1, 0)[:, selected_features],
        y_train = y_train,
    )
```

```
    time_passed = time.time() - since

    score = pipe.score(
        X_test = np.where(X_test >= 1, 1, 0)[:, selected_features],
        y_test = y_test.values
    )

    cache.append(
        {
            'Iter num'          : iteration,
            'Features'          : features,
            'Feature Dim'       : len(selected_features),
            'Fitting Time (s)' : round(time_passed, 4),
            'Accuracy'          : score['accuracy']
        }

    )

df_cache = pd.DataFrame(cache).sort_values('Accuracy', ascending = False)
df_cache['Iter num'] = (df_cache['Iter num'] / 100).astype(int)

# %%
df_cache

# %% [markdown]
# ## k-Neirest Neigbhor

# %%
doc = df['clean_freq_removed_text']

vectorizer=  TfidfVectorizer(
    max_features = 10000,
    ngram_range = (1, 1),
    analyzer='word',
    stop_words='english',
    norm='l2'
)

vectorizer.fit(doc)

features = vectorizer.transform(doc)

concat_doc_count = False

if concat_doc_count:
    X = np.concatenate(
            [
                features.toarray(),
                df['doc_count_clean_freq_removed_text'].values.reshape(-1, 1)
        ],
        axis = 1
    )

else:
    X = features.toarray()

y = df['sentiment']

```

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size = 0.22,
    random_state = 21
)

X_train  = pd.DataFrame(X_train)
X_test  = pd.DataFrame(X_test)
y_train = pd.DataFrame(y_train)
y_train.columns = ['class']

print(f"X_train shape: {X_train.shape}")
print(f"X_test shape: {X_test.shape}")
print(f"y_train shape: {y_train.shape}")
print(f"y_test shape: {y_test.shape}")

# %%
distances = [
    'euclidean',
    'manhattan',
    'cosine'
]

k_neighbors = 7

cache_model = []

for distance in distances:
    pipe = Pipeline([
        ('classifier',
        KNeighborsClassifier(
            k_neighbors = k_neighbors,
            distance_metric = distance
            )
        )
    ])

    pipe.fit(
        X_train = X_train.values,
        y_train = y_train.values
    )

    score = pipe.score(
        X_test = X_test.values,
        y_test = y_test.values
    )


    cache_model.append(
        (
            pipe.model,
            pipe.model.distance_metric,
            score['accuracy']
        )
    )

result_df = pd.DataFrame(
    cache_model,
    columns = [
```

```python
        'Model',
        'Distance',
        'Accuracy'
    ]
)

result_df

# %% [markdown]
# # Logistic Regression

# %%
class LogisticRegression(Classifier):
    def __init__(self):
        super().__init__()

    def init_params(self,
        input_shape:int,
        output_shape:int = 1
    ):
        self.__random_seed()

        #assert self.X_train.shape[1] == self.X_test.shape[1], 'Improper feature dimension!'

        W_high = self.__init_xavier(input_shape, output_shape)
        W_low  = - W_high
        W_size = (input_shape, output_shape)
        B_size = (1, output_shape)

        self.W = np.random.uniform(
            W_low,
            W_high,
            size = W_size
        )

        self.b = np.random.uniform(
            W_low,
            W_high,
            size = B_size
        )


    def __random_seed(self, seed = 32):
        """ Random seed for reproducebility """
        random.seed(seed)
        np.random.seed(seed)

    def __init_xavier(self, L_pre, L_post):
        """ 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))]  """
        return np.sqrt(6/(L_pre + L_post))

    def __train_config(self,
        lr:float,
        batch_size:int,
        epochs:int,
    ):
```

```
        self.lr = lr
        self.batch_size = batch_size
        self.epochs = epochs

    def sigmoid(self,X, grad = False):
        """ Computing sigmoid and it's gradient w.r.t. it's input """
        sig = 1/(1 + np.exp(-X))

        return sig * (1-sig) if grad else sig

    def __forward(self, X):

        Z = (X @ self.W)  + self.b
        A = self.sigmoid(Z)

        return {
            "Z": Z,
            "A": A
        }


    def __SGD(self, grads):
        self.W -= self.lr * grads['W']
        self.b -= self.lr * grads['b']


    def matrix_back_prop(self, outs, X, Y):
        """ Matrix form backward propagation """
        m = self.batch_size

        Z = outs['Z']
        A = outs['A']

        dZ = (A-Y) * self.sigmoid(Z, grad = True)
        dW = (1 / m) * (X.T @ dZ)
        db = (1 / m) * np.sum(dZ, axis=0, keepdims=True)

        assert self.W.shape == dW.shape, f'Error in weight shapes!, {dW.shape} does not match with {self.W.shape}'
        assert self.b.shape == db.shape, f'Error in bias shapes!, {db.shape} does not match with {self.b.shape}'

        grads = {}
        grads['W'] = dW
        grads['b'] = db

        return grads


    def backward(self,
        outs,
        X,
        Y
    ):
        return self.matrix_back_prop(
            outs,
            X,
            Y
        )
```

```


    def BinaryCrossEntropyLoss(self, pred, label):
        m = pred.shape[0]
        preds = np.clip(pred, 1e-16, 1 - 1e-16)
        loss = np.sum(-label * np.log(preds + 1e-20) - (1 - label) * np.log(1 - preds + 1e-20))
        return loss / m

    def eval(self, x, y, knob:float = 0.5):
        predictions = self.__forward(x)
        predictions = predictions['A']
        predictions[predictions>=knob] = 1
        predictions[predictions< knob] = 0
        acc_score = self.accuracy(predictions, y)

        return acc_score

    def __accuracy(self,pred,label):
        return np.sum(pred == label) / pred.shape[0]


    @timeit
    def fit(
        self,
        X_train,
        y_train,
        X_test,
        y_test,
        lr:float = 1e-2,
        batch_size:int = 32,
        epochs:int = 100,
        verbose = True
    ):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        self.init_params(
            input_shape = X_train.shape[1]
            )

        self.__train_config(
            lr,
            batch_size,
            epochs,
        )

        self.history = {}

        self.history['train'] = {
            'loss': [],
            'acc' : []
        }

        self.history['val'] =   {
            'loss': [],
```

```
            'acc' : []
        }

        m = self.batch_size

        self.sample_size_train = X_train.shape[0]

        for epoch in range(self.epochs):

            perm = np.random.permutation(self.sample_size_train)

            for i in range(self.sample_size_train // m):


                shuffled_index = perm[i*m: (i+1)*m]

                X_feed = X_train[shuffled_index]
                y_feed = y_train[shuffled_index]

                outs = self.__forward(X_feed)
                grads = self.backward(
                    outs,
                    X_feed,
                    y_feed
                )
                self.__SGD(grads)

            loss_train = self.BinaryCrossEntropyLoss(
                self.__forward(X_train)['A'],
                y_train
            )

            acc_train = self.eval(
                X_train,
                y_train
            )

            self.history['train']['loss'].append(loss_train)
            self.history['train']['acc'].append(acc_train)



            loss_val = self.BinaryCrossEntropyLoss(
                self.__forward(X_test)['A'],
                y_test
            )

            acc_val  = self.eval(
                X_test,
                y_test
            )

            self.history['val']['loss'].append(loss_val)
            self.history['val']['acc'].append(acc_val)

            if verbose:
                print(f"[{epoch}/{self.epochs}] ------> Training : BCE: {loss_train} and
                ↪  Acc: {acc_train}")
```

```
                print(f"[{epoch}/{self.epochs}] ------> Testing  : BCE: {loss_val}   and Acc: {acc_val}")


    def __str__(self):
        model = LogisticRegression().__class__.__name__
        model += f' with hyperparameters (learning rate,batch_size,epochs) = ({self.lr},{self.batch_size},{self.epochs})'
        num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] * self.b.shape[1]
        model += f'\n There are {num_params} number of traniable parameters'
        return model

    def __repr__(self):
        model = LogisticRegression().__class__.__name__
        model += f' with hyperparameters (learning rate,batch_size,epochs) = ({self.lr},{self.batch_size},{self.epochs})'
        num_params = self.W.shape[0] * self.W.shape[1] + self.b.shape[0] * self.b.shape[1]
        model += f'\n There are {num_params} number of traniable parameters'
        return model

    def plot_history(self):

        fig,axs = plt.subplots(1,2,figsize = (24,8))
        axs[0].plot(self.history['train']['loss'],color = 'orange',label = 'Training')
        axs[0].plot(self.history['val']['loss'], label = 'Validation')
        axs[0].set_xlabel('Epochs')
        axs[0].set_ylabel('BCE Loss')
        axs[0].set_title(f'Binary Cross Entropy Loss Over Iterations with Learning Rate $\eta$ = {self.lr}')
        axs[0].legend(loc="upper right")
        axs[0].grid()

        axs[1].plot(self.history['train']['acc'],color ='orange',label = 'Training')
        axs[1].plot(self.history['val']['acc'], label = 'Validation')
        axs[1].set_xlabel('Epochs')
        axs[1].set_ylabel('Accuracy')
        maxs = round(max(self.history['train']['acc']),3), round(max(self.history['val']['acc']),3)
        axs[1].set_title(f'Accuracy Over Iterations with Learning Rate $\eta$ = {self.lr} \n Best Accuracy in (Training,Validation) = {maxs} ')
        axs[1].legend(loc="lower right")
        axs[1].grid()

# %%
train_config = dict(
    lr          = 1e-3,
    batch_size  = 64,
    epochs      = 1000,
    verbose = False,
)

model = LogisticRegression()
model.fit(
    X_train.values,
    y_train.values.reshape(-1, 1),
    X_test.values,
    y_test.values.reshape(-1, 1),
```

```
    **train_config
)
model.plot_history()
print(model)

# %%
train_config = dict(
    lr          = 1e-3,
    batch_size  = 64,
    epochs      = 100000,
    verbose = False,
)

model = LogisticRegression()
model.fit(
    X_train.values,
    y_train.values.reshape(-1, 1),
    X_test.values,
    y_test.values.reshape(-1, 1),
    **train_config
)
model.plot_history()
print(model)

# %%
train_config = dict(
    lr          = 9e-3,
    batch_size  = 128,
    epochs      = 10000,
    verbose = False,
)

model = LogisticRegression()
model.fit(
    X_train.values,
    y_train.values.reshape(-1, 1),
    X_test.values,
    y_test.values.reshape(-1, 1),
    **train_config
)
model.plot_history()
print(model)

# %% [markdown]
# # MLP (Multi-Layer Perceptron)

# %%
class MLP(Classifier):

    def __init__(self,
        input_size = X_train.shape,
        batch_size = 19 ,
        n_neurons = 76 ,
        mean = 0,
        std = 1,
        lr = 1e-1,
        distribution = 'Xavier'
    ):

```

```

        np.random.seed(15)
        self.lr = lr
        self.mse_train = {}
        self.mce_train = {}
        self.mse_test = {}
        self.mce_test = {}

        self.sample_size = input_size[0]
        self.feature_size = input_size[1]
        self.batch_size = batch_size
        self.n_neurons = n_neurons
        self.mean, self.std = mean, std

        self.dist = distribution


        self.n_update = round((self.sample_size/self.batch_size))

        self.W1_size = self.feature_size,self.n_neurons
        self.W2_size = self.n_neurons,1

        self.B1_size = 1,self.n_neurons
        self.B2_size = 1, 1

        self.B1 = np.random.normal(loc = self.mean, scale = self.std, size = (self.B1_size)) * 0.01
        self.B2 = np.random.normal(loc = self.mean, scale = self.std, size = (self.B2_size)) * 0.01

        self.he_scale1 = np.sqrt(2/self.feature_size)
        self.he_scale2 = np.sqrt(2/self.n_neurons)
        self.xavier_scale1 = np.sqrt(2/(self.feature_size+self.n_neurons))
        self.xavier_scale2 = np.sqrt(2/(self.n_neurons+1))

        if (self.dist == 'Zero') :
            self.W1 = np.zeros((self.W1_size))
            self.W2 = np.zeros((self.W2_size))

        elif (self.dist == 'Gauss'):
            self.W1 = np.random.normal(loc = self.mean, scale = self.std, size = (self.W1_size))* 0.01
            self.W2 = np.random.normal(loc = self.mean, scale = self.std, size = (self.W2_size))* 0.01

        elif (self.dist == 'He'):
            self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) * self.he_scale1
            self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) * self.he_scale2

        elif (self.dist == 'Xavier'):

            self.W1 = np.random.randn(self.W1_size[0],self.W1_size[1]) * self.xavier_scale1
            self.W2 = np.random.randn(self.W2_size[0],self.W2_size[1]) * self.xavier_scale2



    def forward(self,X):
```

```
        Z1 = (X @ self.W1)  + self.B1
        A1 = np.tanh(Z1)
        Z2 = (A1 @ self.W2) + self.B2
        A2 = np.tanh(Z2)

        return {
            "Z1": Z1,
            "A1": A1,
            "Z2": Z2,
            "A2": A2
        }


    def tanh(self,X):
        return (np.exp(X) - np.exp(-X))/(np.exp(X) + np.exp(-X))

    def tanh_der(self,X):
        return  1-(np.tanh(X)**2)

    def backward(self,outs, X, Y):
        m = (self.batch_size)

        Z1 = outs['Z1']
        A1 = outs['A1']
        Z2 = outs['Z2']
        A2 = outs['A2']

        dZ2 = (A2-Y)* self.tanh_der(Z2)
        dW2 = (1/m) * (A1.T @ dZ2)
        dB2 = (1/m) * np.sum(dZ2, axis=0, keepdims=True)

        dZ1 = (dZ2 @ self.W2.T) * self.tanh_der(Z1)
        dW1 = (1/m) * (X.T @ dZ1)
        dB1 = (1/m) * np.sum(dZ1, axis=0, keepdims=True)


        return {
            "dW1": dW1,
            "dW2": dW2,
            "dB1": dB1,
            "dB2": dB2
        }

    def Loss(self,pred, y_true, knob = 0):

        mse = np.square(pred-y_true).mean()

        pred[pred>=knob]=1
        pred[pred<knob]=-1

        mce = (pred == y_true).mean()

        return {
            'MSE':mse,
            'MCE':mce
        }
```

```
    def SGD(self,grads):
        self.W1 -= self.lr * grads['dW1']
        self.W2 -= self.lr * grads['dW2']
        self.B1 -= self.lr * grads['dB1']
        self.B2 -= self.lr * grads['dB2']

    def fit(self,X,Y,X_test,y_test,epochs = 300,verbose=True):
        """
        Given the traning dataset,their labels and number of epochs
        fitting the model, and measure the performance
        by validating training dataset.
        """

        m = self.batch_size

        for epoch in range(epochs):
            perm = np.random.permutation(self.sample_size)

            for i in range(self.n_update):


                batch_start = i * m
                batch_finish = (i+1) * m
                index = perm[batch_start:batch_finish]

                X_feed = X[index]
                y_feed = Y[index]


                outs = self.forward(X_feed)
                loss = self.Loss(
                    outs['A2'],
                    y_feed
                )

                outs_test = self.forward(X_test)
                loss_test = self.Loss(
                    outs_test['A2'],
                    y_test
                )

                grads = self.backward(
                    outs,
                    X_feed,
                    y_feed
                )

                self.SGD(grads)

            self.mse_train[f"Epoch:{epoch}"] = loss['MSE']
            self.mce_train[f"Epoch:{epoch}"] = loss['MCE']
            self.mse_test[f"Epoch:{epoch}"] = loss_test['MSE']
            self.mce_test[f"Epoch:{epoch}"] = loss_test['MCE']

            if verbose:
                print(f"[{epoch}/{epochs}] ------> Training :MSE: {loss['MSE']} and MCE:
                ↪  {loss['MCE']}")
                print(f"[{epoch}/{epochs}] ------> Testing :MSE: {loss_test['MSE']} and
                ↪  MCE: {loss_test['MCE']}")
```

```python

    def history(self):
        return {
            'Train_MSE' : self.mse_train,
            'Train_MCE' : self.mce_train,
            'Test_MSE'  : self.mse_test,
            'Test_MCE'  : self.mce_test
        }

# %%
initialize = 'Xavier'
input_size = X_train.shape
batch_size = 64
hidden_neurons = 256
epochs = 100

model = MLP(
    input_size,
    batch_size,
    hidden_neurons,
    lr=1e-3,
    distribution = 'Gauss'
)

# %%
model.fit(
    X_train.values,
    y_train.values.reshape(-1, 1),
    X_test.values,
    y_test.values.reshape(-1, 1),
    epochs
)

# %%
history = model.history()

# %%
plt.rcParams['figure.figsize'] = (9,6)
plt.plot(history['Train_MCE'].values())
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Training')
plt.show()

plt.plot(history['Train_MSE'].values(),color = 'green')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Squared Error')
plt.title('MSE versus Epoch in Training')
plt.show()

plt.plot(history['Test_MCE'].values(),color = 'orange')
plt.xlabel('# of Epoch')
plt.ylabel('Mean Classification Error')
plt.title('MCE versus Epoch in Validation')
plt.show()


plt.plot(history['Test_MSE'].values(),color = 'blue')
plt.xlabel('# of Epoch')
```

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