Palavi Rajgude

301892969, CS164

REPORT - SENTIMENT ANALYSIS USING LOGISTIC REGRESSION AND NAIVES BAYES

GOALS OF THE PROJECT:

Perform a Sentiment Analysis on tweets with the help of Logistic Regression and Naives Bayes which classified the tweet as negative or positive.

TOOLS AND ALGORITHMS USED:

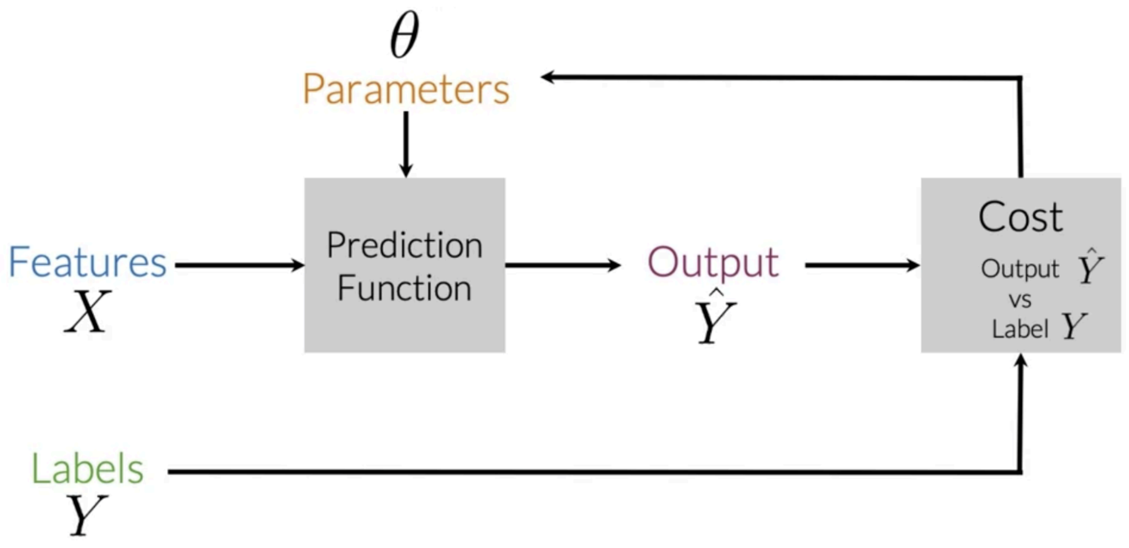
1. I have made use of the nltk library `nltk that natural language processing library used for NLP related tasks and also provides alot of NLP related data. You can check this website here:` https://www.nltk.org/

```
import nltk
nltk.download('twitter_samples')
nltk.download('stopwords')
import numpy as np
import pandas as pd
```

2. Google Colab: I have used google colab for executing all my tasks and executing my notebook and also utilizing its abilities like the GPU it provides for faster processing of my data.
3. CHATGPT: I have used chatgpt to help with coding such as suggesting alternative ways to get a code that is not working, and getting maximum benefit from libraries such as numpy and pandas. Discovering new methods to apply and use libraries.
4. Gemini: Getting an error explanation for google colab whenever I was stuck, it provided instant feedback.
5. Stack overflow: It was not possible to get all the answers from chatgpt and gemini all the time. Sometimes I had to refer to the documentation and also stack overflow for more ways to tackle a problem.

INTRODUCTION:

In supervised learning, we have input and output and this input goes into our prediction function and we get the predicted value and compare it with the true value. The difference in accuracy is what we call cost which will help us update our parameters. Below is a picture that summarizes well.



To perform sentiment analysis on a tweet, we first have to represent the text (i.e. "I am happy because I am learning NLP ") as features, then train logistic regression classifier, and then can use it to classify the text. (as simple as that)
Note in this case, you either classify 1 for a positive sentiment or 0 for a negative sentiment.

Given a tweet, we can represent it as a vector of dimension V, V is your vocabulary size. If we had the tweet "I am happy because I am learning NLP", then we would put a 1 in the corresponding index for any word in the tweet, and a 0 otherwise.

As *V* gets larger, the vector becomes more sparse, we end up having many more features and end up training θ*, V* parameters. This could result in larger training time, and large prediction time.

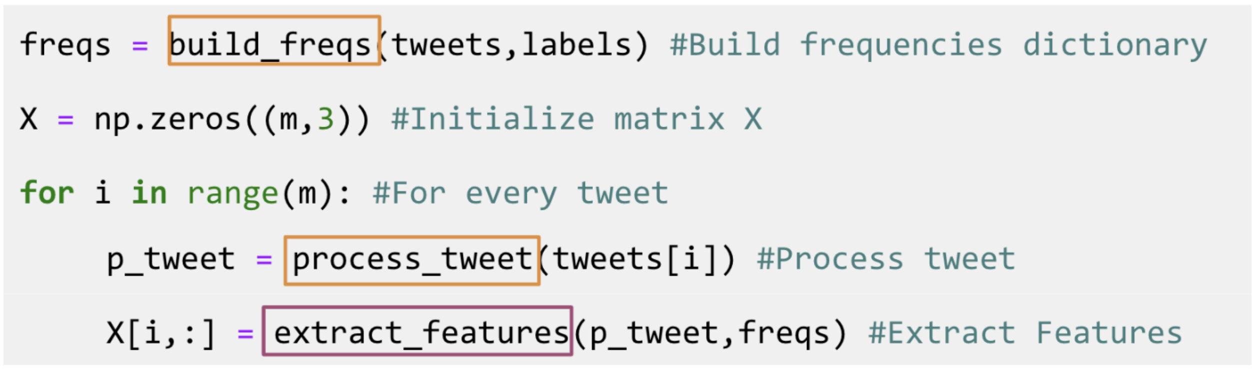
PREPROCESSING OF DATA:

When preprocessing, you have to perform the following:
1. Eliminate handles and URLs
2. Tokenize the string into words.
3. Remove stop words like "and, is, a, on, etc."
4. Stemming- or convert every word to its stem. Like a dancer, dancing, dancing, becomes 'danc'. You can use a porter stemmer to take care of this.
5. Convert all your words to lowercase.

For example the following tweet "@YMourri and @AndrewYNg are tuning a GREAT AI model at https://deeplearning.ai!!!" after preprocessing becomes
Putting it all together:

Over all , start with a given text, we perform preprocessing, then we do feature extraction to convert text into numerical representation

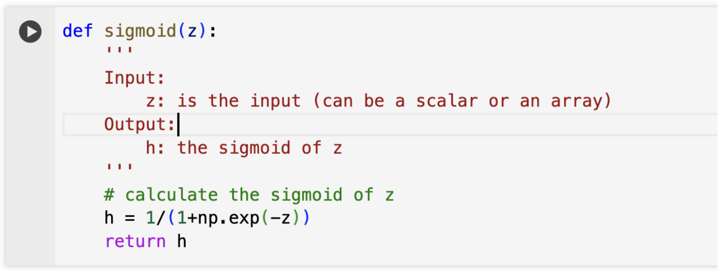
When implementing it with code, it becomes as follows:



```
freqs = build_freqs(tweets,labels) #Build frequencies dictionary

X = np.zeros((m,3)) #Initialize matrix X

for i in range(m): #For every tweet

    p_tweet = process_tweet(tweets[i]) #Process tweet

    X[i,:] = extract_features(p_tweet,freqs) #Extract Features
```

LOGISTIC REGRESSION:

Logistic regression makes use of the sigmoid function which outputs a probability between 0 and 1. The sigmoid function with some weight parameter θ and some input $x(i)$ is defined as follows.



```
def sigmoid(z):
    '''
    Input:
        z: is the input (can be a scalar or an array)
    Output:
        h: the sigmoid of z
    '''
    # calculate the sigmoid of z
    h = 1/(1+np.exp(-z))
    return h
```

Now given a tween transform it into a vector and run it through your sigmoid function to get a prediction as follows:
Tweet:
@bestai @GUIDAR are tuning a great ai model ——--> [tun, ai, great, model] ——-->
θ = [0.00003
0.00150
-0.00120]

FEATURE EXTRACTION:

Given a corpus with positive and negative tweets as follows:

Positive tweets	Negative tweets
I am happy because I am learning NLP	I am sad, I am not learning NLP
I am happy	I am sad

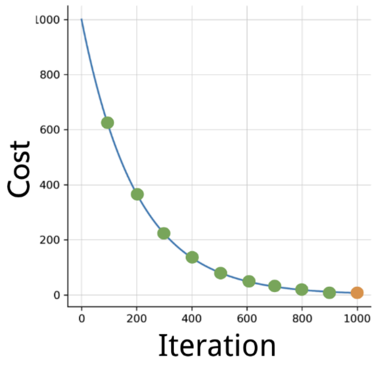
We have to encode each tweet as a vector. Previously, this vector was of dimension V. Now, will represent it with a vector of dimension 3. To do so, have to create a dictionary to map the word, and the class it appeared in (positive or negative) to the number of times that word appeared in its corresponding class.

Vocabulary	PosFreq (1)	NegFreq (0)
I	3	3
am	3	3
happy	2	0
because	1	0
learning	1	1
NLP	1	1
sad	0	2
not	0	1

We call this dictionary `freqs`. In the table above, we can see how words like happy and sad tend to take clear sides, while other words like "I, am" tend to be more neutral. Given this dictionary and the tweet, "I am sad, I am not learning NLP", you can create a vector corresponding to the feature.

GRADIENT DESCENT:

We initialize our parameter θ, that we can use in our sigmoid, then compute the gradient that we will use to update θ, and then calculate the cost. Keep doing so until it is good enough. Usually you keep training until the cost converges. If we were to plot the number of iterations versus the cost, we would see something like this, the cost should keep decreasing and below is the code used for that.



```
def computeCost(x, y, theta):
    m = len(y)
    z = np.dot(x, theta)
    h = sigmoid(z)
    epsilon = 1e-5  # small value to avoid log(0)
    cost = - (1/m) * (np.dot(y.T, np.log(h + epsilon)) + np.dot((1 - y).T,
np.log(1 - h + epsilon)))
    return cost

def gradientDescent(x, y, theta, alpha, num_iters):
    '''
    Input:
        x: matrix of features which is (m, n+1)
        y: corresponding labels of the input matrix x, dimensions (m, 1)
        theta: weight vector of dimension (n+1, 1)
        alpha: learning rate
        num_iters: number of iterations you want to train your model for
    Output:
        J: the final cost
        theta: your final weight vector
    '''
    # get 'm', the number of rows in matrix x
    m = x.shape[0]


    for i in range(num_iters):
        # get z, the dot product of x and theta
        z = np.dot(x, theta)

        # get the sigmoid of z
        h = sigmoid(z)

        # calculate the cost function
        J = computeCost(x, y, theta)

        # update the weights theta
        gradient = (1/m) * np.dot(x.T, (h - y))
        theta = theta - alpha * gradient
        #print('theta', theta)

        # Optionally print the cost every 100 iterations to monitor progress

    # Calculate the final cost
    J = computeCost(x, y, theta)

    return J.item(), theta
```

Finally, we are going to follow the following steps:

- Learn how to extract features for logistic regression given some text
- Implement logistic regression from scratch
- Apply logistic regression on a natural language processing task
- Test using your logistic regression
- Perform error analysis

RESULTS FOR LOGISTIC REGRESSION:

1. I trained the model with theta [6e-08, 0.0005382, -0.0005583] after training it for 700 iterations and costs 0.22512220 by applying gradient descent and reducing cost.
2. I predicted many tweets according to the logistic regression model and it helped me get the results as follows.
    a. I am happy -> 0.519275
    b. I am bad -> 0.494347
    c. this movie should have been great. -> 0.515980
    d. great -> 0.516065
    e. great great -> 0.532097
    f. great great great -> 0.548063
    g. great great great great -> 0.563930
3. Accuracy of the model is as given as: Logistic regression model's accuracy = 0.9950

NAIVES BAYES:

In naives bayes, to calculate a probability of a certain event happening, take the count of that specific event and divide by the sum of all events. Furthermore, the sum of all probabilities has to equal 1.

$$P(\text{Positive} \mid \text{“happy”}) = \frac{P(\text{Positive} \cap \text{“happy”})}{P(\text{“happy”})}$$

Conditional probabilities help us reduce the sample search space and following is the bayes rule.

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}.$$

To build a classifier, we will first start by creating conditional probabilities then once we have the probabilities, can compute the likelihood score. A score greater than 1 indicates that the class is positive, otherwise it is negative.

We usually compute the probability of a word given in a class. However, if a word does not appear in the training, then it automatically gets a probability of 0, to fix this we add smoothing Note that we added a 1 in the numerator, and since there are V words to normalize, we add V in the denominator.

$$P\left(\mathrm{w_i} \mid \text{class}\right) = \frac{\text{freq}(\mathrm{w_i}, \text{ class }) + 1}{(\mathrm{N_{class}} + \mathrm{V})}$$

To compute the log likelihood, we need to get the ratios and use them to compute a score that will allow us to decide whether a tweet is positive or negative. The higher the ratio, the more positive the word is. The first component is called the log prior and the second component is the log likelihood.

Finally we have the following steps:

To train naïve Bayes classifier, have to perform the following steps:
1) Get or annotate a dataset with positive and negative tweets
2) Preprocess the tweets: process_tweet(tweet) ➞ [w1, w2, w3, ...]: Lowercase, Remove punctuation, urls, names, remove stop words, stemming, tokenize sentences

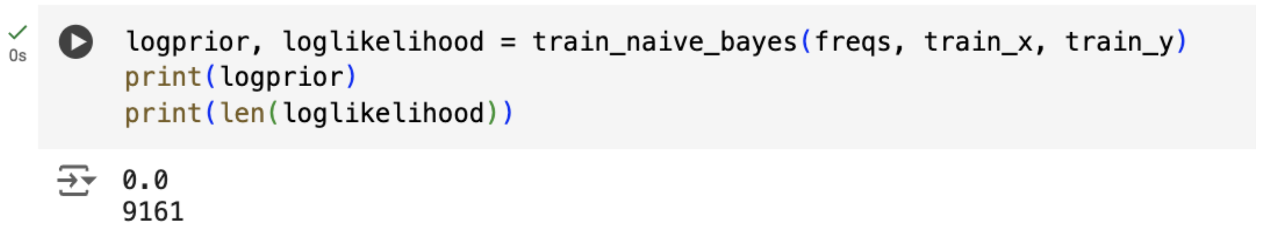
3) Compute freq(w, class): `{('happi', 1): 1, ('trick', 0): 1, ('sad', 0): 1, ('tire', 0): 2}`
4) Get P(w|pos),P(w|neg) P(w|pos),P(w|neg)
5) Get probability of each word'
6) Get log prior

Final Code (adding log likelihood for all functions):

```
def naive_bayes_predict(tweet, logprior, loglikelihood):
    '''
    Input:
        tweet: a string
        logprior: a number
        loglikelihood: a dictionary of words mapping to numbers
    Output:
        p: the sum of all the logliklihoods of each word in the tweet (if
found in the dictionary) + logprior (a number)

    '''
    #print(loglikelihood)

    # process the tweet to get a list of words
    word_l = process_tweet(tweet)

    # initialize probability to zero
    p = 0

    # add the logprior
    p += logprior

    for word in word_l:

        # check if the word exists in the loglikelihood dictionary
        if word in loglikelihood:
            # add the log likelihood of that word to the probability
            p += loglikelihood.get(word)



    return p
```

RESULTS FOR NAIVES BAYES:

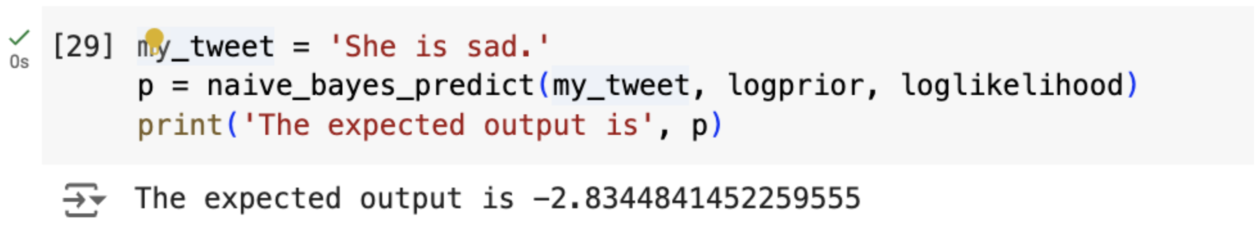
1. The result for log prior and likelihood is below for the training set.



```
logprior, loglikelihood = train_naive_bayes(freqs, train_x, train_y)
print(logprior)
print(len(loglikelihood))
```

```
0.0
9161
```

2. Prediction of a tweet through the model.



```
[29] my_tweet = 'She is sad.'
     p = naive_bayes_predict(my_tweet, logprior, loglikelihood)
     print('The expected output is', p)
```

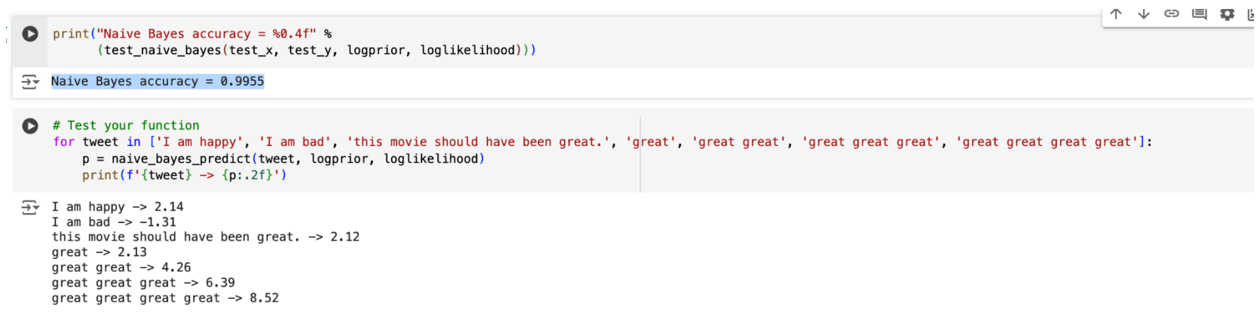
```
The expected output is -2.8344841452259555
```

3. Accuracy of the model from the notebook below.

```
print("Naive Bayes accuracy = %0.4f" %

       (test_naive_bayes(test_x, test_y, logprior, loglikelihood)))
```

Naive Bayes accuracy = 0.9955

4. Below is the accuracy and prediction from our model.



```
print("Naive Bayes accuracy = %0.4f" %
      (test_naive_bayes(test_x, test_y, logprior, loglikelihood)))
```

```
Naive Bayes accuracy = 0.9955
```

```
# Test your function
for tweet in ['I am happy', 'I am bad', 'this movie should have been great.', 'great', 'great great', 'great great great', 'great great great great']:
    p = naive_bayes_predict(tweet, logprior, loglikelihood)
    print(f'{tweet} -> {p:.2f}')
```

```
I am happy -> 2.14
I am bad -> -1.31
this movie should have been great. -> 2.12
great -> 2.13
great great -> 4.26
great great great -> 6.39
great great great great -> 8.52
```

FUTURE WORK:

1. Natural language processing is a big area of interest and exploring this subject will never be enough and the models that I have presented are very basic models but a good intuition of how it got started.
2. Further exploration for me on this topic to use pre-trained models and perhaps build models of my own.
3. Also, use some advanced deep learning models and concepts and go in depth about this topic of NLP.
4. I would like to develop and extend this understanding further to neural networks and try with different datasets, what are the issues and problems faced there.

IDEAS:

1. There are many applications of Naive Bayes including:
    - Author identification
    - Spam filtering
    - Information retrieval
    - Word disambiguation
2. Logistic regression is best suited for classification based projects in determining whether its yes/no type question, true or false, binary classification.

—-- Thank you! ----------------------------------—--------------