Spring 2022

Deep Learning – CS 4277

Spam Detector

Final Report

Yassir Benabdallah

ybenabda@students.kennesaw.edu, y.benabdallah@aui.ma

Supervised by:

Dr. Hafiz Khan

Table of Contents

I. Abstract

Email is a virtual way to express mails sent or received by individuals. Most of the official information is communicated through this mean with the use of various electronic devices, such as mobile phones, laptops, tablets, etc. This formal exchange of information makes the tidiness of inboxes in terms of the credibility of emails an important variable within this context. Unfortunately, it is not always the case; the massive increase of spam fills our inboxes with irrelevant emails, which causes serious threat on our social networks. Therefore, it has become imperative to build algorithms that detect such misuse of technology.

Throughout this project, the aim is to build different algorithms that help in detecting spam messages and emails using mathematical and coding techniques inspired from deep learning and probability concepts. The diversity of these algorithms would allow us to compare them to each other, and hence end up with best accuracy. Four main techniques are involved in this paper: Naïve Bayes, Logistic Regression, Recurrent Neuron Network and Long Short-Term Memory. Accuracy values have ranged from 95% to 98% depending on the technique used, which shows the outperformance of some methods over others.

II. Introduction

Following the work environment mentioned in the abstract, a spam mail detection system is proposed to classify a given email as ham or spam email. This filtering operation focuses on the content of the message, such that the algorithm would take care of identifying the terms that are frequently repeated in spam emails and calculate the probability that the existence of such terms in an email makes it spam.

Communication through emails has increased in multiple forms in the last decade leading to a new area or platform for junk promotions from disreputable marketers and spammers. With the wide spread of websites with different purposes, people tend to innocently share their credentials in order to benefit from temporary or imaginary services. Nowadays, more than 333 billion emails are sent and received every day in the rate of 3.5 million emails per second, and there are more than four billion email users worldwide[1]. With these astronomical numbers, spam emails are considered as one of the most dangerous threats posed to email users. The same source indicates that 85% of all emails are spam mails which usually take the format of links that lead users to websites with malware or phishing attacks that can access and disrupt the receiver's computer. With that being said, an effective spam filtering technology is a significant contribution to the sustainability of the cyberspace and to our society.

III. Related Work

A lot of work has been invested in solving this problem. Jain, Manisha and Agarwal evaluated spam detection in social media based on RNN and CNN, and the ended to the conclusion where deep learning has great ability to perform such a classification, as these methods can learn features by themselves unlike the traditional methods of features extraction[2]. Ishtiaq, Donghai and Choong approached this problem using Naïve-Bayes classifier and Apriori algorithm[3]. Almeida, Hidgalo and Yamakami offered and encoded SMS spam collection treated with

Support Vector Machine (SVM), which outperformed other evaluated classifiers based on the largest dataset at that time[4]. The same dataset is used in this project. Chandra and Khatri proposed a new method utilizing Recurrent Neural Network (RNN) and Long Short Term Memory (LSTM) using Keras models and Tensorflow backend to detect Spam and Ham with an accuracy that reached 98%[5]. One of the best achievements in this topic is Google Gmail's System. It predicts weather a received message is spam or ham based on results from statistical filters and user feedback. If the message satisfies certain criteria, then it is whitelisted; otherwise, it goes through another process of anti-spam filter for making the final decision. Google relies more on the user feedback though, such that it is always working updating its quality guidelines and feedback forms to be understood by its users. Thanks to this, Google is observing spam emails with 99.9% accuracy[6].

IV. Methodology and Experimental Evaluations

1. Dataset Description

For this project, I am using a set of SMS messages that are divided into two parts are labeled as Spam or Ham. It consists of 5572 samples, such that 747 are spam and 4825 are ham, which are enough to build and analyze our model. Please refer to the attached CSV file for more details[7]

2. Naïve-Bayes

a. Methodology

The algorithm is supposed to be based on the Naïve Bayes classifier, which is derived from Bayes Theorem, one of the most famous theorems in probability and statistics.

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

What is special about it is that it assumes that all features of the model are independent. Within the context of spam filtering, words in messages are independent from each other regardless of the context they are put in. Moreover, we can get to know the probability of a certain email being spam or ham based on the set of words that compose it given that they belong to spam or ham email, which is provided in the dataset.

Mathematically speaking, our task is to find the probability that a certain message or email is spam given the words the form it. We can denote this by the following expression.

$$P(Spam|w_1, w_2, w_3, \dots.)$$

In other words, for a message that is composed of words $(w_1, w_2, w_3, \dots.)$ to be spam, it is proportional to the probability to detect the spam multiplied by a product of probabilities for these words in the message to belong a spam message. In fact, we calculate the probability of finding a word $w_i$in a already known spam message. The following formula summarizes this explanation.

$$P(Spam|w_1, w_2, w_3, \dots.) \propto P(Spam) * \prod_{i=1}^{n} P(w_i|Spam)$$

In order to find each word probabilities (denoted as $P(w_i|Spam)$) we calculate the frequency of this word in the messages that are labeled as *Spam*[8].

b. Experimental Evaluation

First, we ignore the role of punctuation, and we assume that they have no role in spam detection, so get rid of them. Then, as mentioned in the Methodology part, messages' words need to independent; therefore, we split the text into separate words.

```
df['v2'] = df['v2'].str.replace('\W+', ' ').str.replace('\s+', ' ').str.strip()

df['v2'] = df['v2'].str.lower()

df['v2'] = df['v2'].str.split()
```

```
C:\Users\pc\AppData\Local\Temp\ipykernel_16968\1105765197.py:1: FutureWarning: The default value of regex will change from True
to False in a future version.
  df['v2'] = df['v2'].str.replace('\W+', ' ').str.replace('\s+', ' ').str.strip()
```

Figure 1: Text Split into Separate Words

Dataset now can be divided into train and test data.

```
train = df.sample(frac=0.8,random_state=1).reset_index(drop=True)

test = df.drop(train.index).reset_index(drop=True)

train = train.reset_index(drop=True)
```

Figure 2: Data Split into Train and Test Data

Then, we count the number of separate words in each test, so that we will be able to calculate the word probabilities. And, finally, we attach to each message all the words in the dataset along with the frequency of each word in that text.

```
vocab = list(set(train['v2'].sum()))
word_counts_per_sms = pd.DataFrame([
    [row[1].count(word) for word in vocab]
    for _, row in train.iterrows()], columns=vocab)
train = pd.concat([train.reset_index(), word_counts_per_sms], axis=1).iloc[:,1:]
```

```
train.head(5)
```

										...								
		good, evenings, to, u, s]																
2	ham	[my, sort, code, is, and, acc, no, is, the, ba...	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
3	ham	[sorry, i, din, lock, my, keypad]	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0
4	spam	[hi, babe, its, chloe, how, r, u, i, was, smas...	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0

5 rows × 7768 columns

Figure 3: Word Count in each Test of Data Training

We define then some metrics that help us in calculating the probability of a certain message being ham or spam, and hence in classifying it:

- **ProbHam:** this variable would return the probability of a message to be ham
- **NumSpam:** this variable would return the number of words in a spam message.
- **NumHam:** this variable would return the number of words in a ham message.
- **vocabSize:** this variable would return the number of special words in the dataset.
- **NaN:** fixed value that refers to the cases where a word in the message does not exist in the dataset.

```
ProbSpam = train['v1'].value_counts()['spam'] / train.shape[0]

ProbHam = train['v1'].value_counts()['ham'] / train.shape[0]

NumSpam = train.loc[train['v1'] == 'spam', 'v2'].apply(len).sum()

NumHam = train.loc[train['v1'] == 'ham','v2'].apply(len).sum()

vocabSize = len(train.columns) - 3

NaN = 1
```

Figure 4: Main Variables in Spam Classification

Now, we can define the two main methods that would allow us to find out the probability of a certain word to belong to spam or ham messages.

```
def spamProb(word):
    if word in train.columns:
        return (train.loc[train['v1'] == 'spam', word].sum() + NaN) / (NumSpam + NaN*vocabSize)
    else:
        return 1

def hamProb(word):
    if word in train.columns:
        return (train.loc[train['v1'] == 'ham', word].sum() + NaN) / (NumHam + NaN*vocabSize)
    else:
        return 1
```

Figure 5: Main Methods in Spam Classification

Finally, we can build a classification function. Based on the results we got from calculating the probability that a certain word is spam, we implement the product of the probabilities of each word being spam or ham. Then, if spam product is greater than the ham product, our message is spam; otherwise, it is ham[9].

```
def classify(message):

    Probspam = ProbSpam
    Probham = ProbHam

    for word in message:
        Probspam *= spamProb(word)
        Probham *= hamProb(word)
    if Probspam > Probham:
        return 'spam'
    elif Probham > Probspam:
        return 'ham'
    else:
        return 'needs human classification'
```

Figure 6: Classification Function

3. Logistic Regression

a. Methodology

The logistic regression model is considered as one of the best models that handle classification problems, especially when it comes to binary classification.

The idea is to build a linear representation of the dataset by associating each input (word) with a value. These values are then multiplied to a corresponding weight and bias to form a linear function as shown in the following formula:

$$f(x_1, x_2, x_3, \dots, x_n) = w_1.x_1 + w_2.x_2 + \cdots + w_n.x_n + \varepsilon$$

The output of this function is then fed to a logistic function (or an activation function), which usually sigmoid function, the comes to play to represent a specific curve (in the case of sigmoid function, it is S shaped). The output tends towards 1 (Spam) as Z goes to infinity, while it goes to 0 (Not Spam) as Z goes to negative infinity. Therefore, a range of output is being produced, which makes the binary classification more accurate[10].

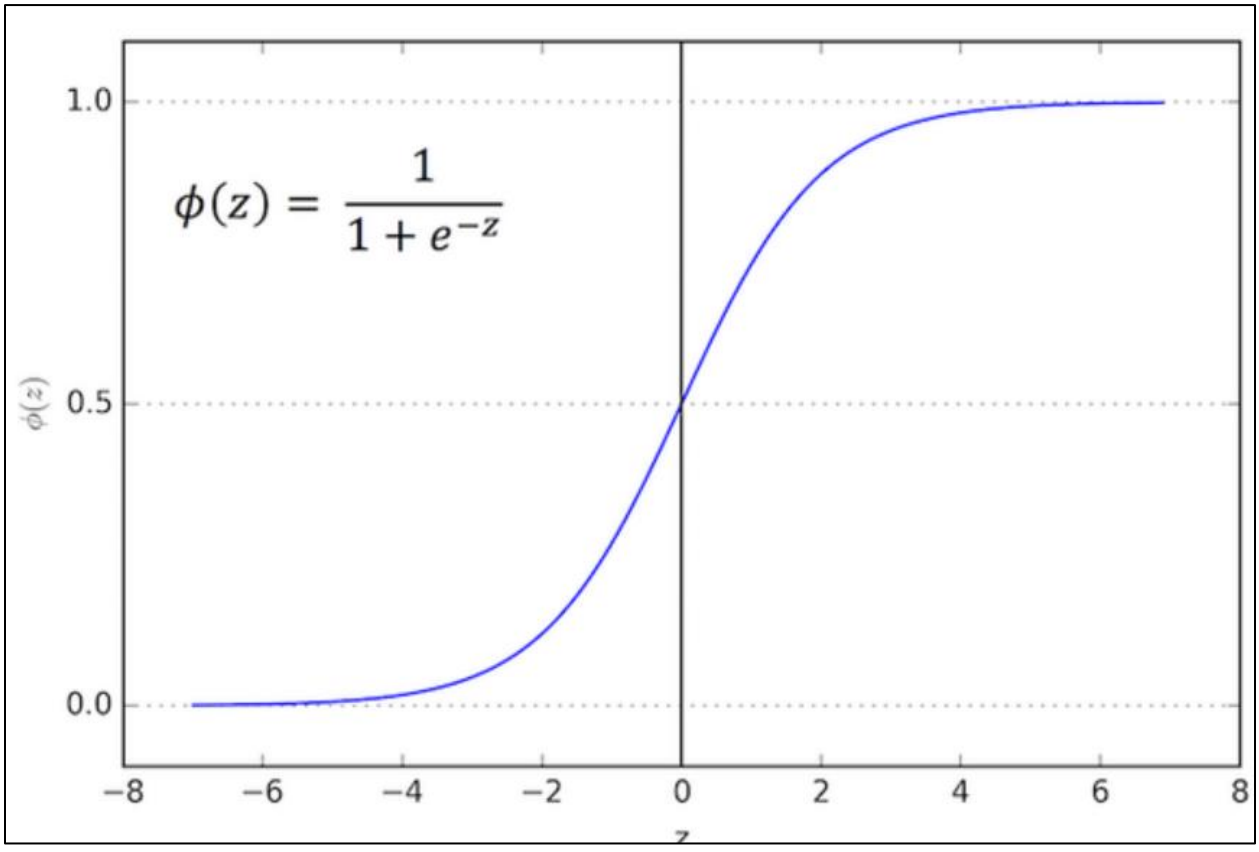

Figure 7: Representation of The Curve of Logistic Regression with Sigmoid Function[11]

b. Experimental Evaluation

The implementation of logistic regression in terms of code is easier compared to naïve-bayes, as there are already pre-built functions that perform the tasks. After the dataset, we preprocess it by converting all characters to lower case and get rid of the special characters, such as punctuations. This would help us in lowering the number of calculations, and hence get quick results[7].

```
stopWords = set(stopwords.words('english'))

def cleanTxt(text):
    text = text.lower()
    text = re.sub(r'^0-9a-zA-Z', ' ', text)
    text = re.sub(r'\s+', ' ', text)
    text = " ".join(word for word in text.split() if word not in stopWords)
    return text
```

Figure 8: Data Preprocessing

Again, the data is split into train and split data, which will be then associated with the logistic regression model.

```
def classify(model, X, y):
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=42, shuffle=True, stratify=y)
    pipeline_model = Pipeline([('vect', CountVectorizer()),
                               ('tfidf', TfidfTransformer()),
                               ('clf', model)])
    pipeline_model.fit(X_train, y_train)

    print("Accuracy = ", pipeline_model.score(X_test, y_test)*100)

    y_pred = pipeline_model.predict(X_test)
    print(classification_report(y_test, y_pred))
```

Figure 9: Splitting and Training of the Dataset

4. Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM)

a. Methodology

RNN is a developed version of feedforward neural networks, as its nodes are connected to each other like a directed graph. Unlike FNN where data only flows forward, information in RNN flows back and forth among nodes; therefore, values and models in each layer gets updated according to the past events. Recursion denotes the use of stacks as memory mechanisms that allow the process of sequences of inputs. We can visualize how it really works using the following recursive formula:

$$S_t = F_w(S_{t-1}, X_t)$$

Where S_t: new state of the RNN at time t
S_{t-1}: state of RNN at time $t-1$
X_t: input at time t

However, there is this downside of RNN that is Vanishing Gradient Problem. In the case of long sequences, it will be difficult to hold information from earlier steps to later ones, which causes the loss of the earlier information. Therefore, many of the first states will not be informative.

LSTM, being a special type of RNN, comes to solve the problem of vanishing gradient problem. LSTM is composed of three gates:

- Forget Gate: gives us an idea on whether we should keep the information from the previous timestamp or forget it.
- Input Gate: gives a value that measures the importance of the new information carried by the input.
- Output Gate: determines the value of the next state.

The use of all these gates in addition to some intermediary states makes RNN able to learn from long-term dependencies.

b. Experimental Evaluation

The experiment is performed according to the following steps[12]:

1. We load the data and split it into training and testing datasets.
2. We tokenize our dataset. Meaning, each word is assigned a unique value. This step is about receiving and processing each token at a particular timestep. The following tokens are then used to prepare a vocabulary, which is the set of unique words used in our dataset.

```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)
```

Figure 10: Dataset Tokenization

3. We sort then the tokens into sequences. As you might know, there is this problem of sequential computing, which enforces LSTMs to deal only with ordered sequences one at a time. Therefore, we need to ensure that the tokens for each word follow the correct order as dictated by each sentence. The last line in *figure 10* refers to this step.
4. We now use padding in order to make all sequences in a batch to fit a given standard length. This is important, as we can find one sentence that is longer than another one; therefore, we put all of them in the same length.

```
data = pad_sequences(sequences, maxlen=maxlen)
```

Figure 12: Sequences Padding

5. Finally, we build and train across 10 epochs.

```
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history_rnn = model.fit(texts_train, y_train, epochs=10, batch_size=60, validation_split=0.2)
```

Figure 13: Training the RNN Model

```
model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
history_ltsm = model.fit(texts_train, y_train, epochs=10, batch_size=60, validation_split=0.2)
```

Figure 14: Training the LSTM Model

5. Results

The results showed that the accuracy value increases as we move towards complex methods. The following table summarizes the results of this paper.

Method	**Accuracy**
Naïve-Bayes	96.55%
Logistic Regression	96.84%
RNN	98.43%
LSTM	98.88%

V. Conclusion

In this paper, we researched about spam detection using four methods: Naïve-Bayes, Logistic Regression, Recurrent Neural Network and Long Short-Term Memory. We put more emphasis on the comparison among these methods in terms of accuracy. The results concluded that the accuracy increases as move towards complex methods. The more we do this, the more we the correlation between words increase; in Naïve-bayes, we are completely ignoring the correlation between words from the same email, while in LSTM, words are fed into sequences which establishes a correlation among them.

Future work can include larger datasets and introduction of other methods, such as SVM and stemming and lemmatization in addition to developing the four main techniques of this paper.

VI. References:

[1] Swindells, M. (2022, April 5). *How many emails are sent per day in 2022? [new data]*. EarthWeb. Retrieved May 1, 2022, from https://earthweb.com/how-many-emails-are-sent-per-day/

[2] Jain, G., Manisha & Agarwal, B. (2016). An Overview of RNN and CNN Techniques for Spam Detection in Social Media. Retrieved May 1, 2022, from IJARCSSE, 6, 126--132.

[3] Ahmed, Ishtiaq; Guan, Donghai; Chung, Tae Choong. (2014) Sms classification based on naive bayes classifier and apriori algorithm frequent itemset, Retrieved May 1, 2022, International Journal of Machine Learning and Computing; Singapore

[4] Almeida, Tiago A and Hidalgo, Jose Maria G and Yamakami, Akebo. (2011) Contributions to the study of SMS spam filtering: new collection and results, Retrieved May 1, 2022, Proceedings of the 11th ACM symposium on Document engineering, ACM

[5] A. Chandra and S. K. Khatri, "Spam SMS Filtering using Recurrent Neural Network and Long Short Term Memory," Retrieved May 1, 2022, from 2019 4th International Conference on Information Systems and Computer Networks (ISCON), 2019, pp. 118-122, doi: 10.1109/ISCON47742.2019.9036269.

[6] Gary, I. (2020, July 3). How spam reports are used at Google. Google. Retrieved May 1, 2022, from https://developers.google.com/search/blog/2020/07/how-spam-reports-are-used-at-google

[7] Mohitgupta-Omg. (2018, March 15). *Kaggle-SMS-SPAM-Collection-dataset-/spam.csv at master · Mohitgupta-OMG/kaggle-SMS-SPAM-collection-dataset-*. GitHub. Retrieved May 1, 2022, from https://github.com/mohitgupta-omg/Kaggle-SMS-Spam-Collection-Dataset-/blob/master/spam.csv

[8] Motwani, S. (2020, August 8). *Email spam filtering using naive Bayes classifier*. Springboard Blog Email Spam Filtering Using Naive Bayes Classifier Comments. Retrieved May 1, 2022, from https://in.springboard.com/blog/email-spam-filtering-using-naive-bayes-classifier/

[9] Horbonos, P. (n.d.). GitHub Blog. GitHub. Retrieved May 1, 2022, from https://gist.github.com/Midvel

[10] Saeed, M. (2021, August 17). *A gentle introduction to sigmoid function*. Machine Learning Mastery. Retrieved May 1, 2022, from https://machinelearningmastery.com/a-gentle-introduction-to-sigmoid-function/

[11] Sharma, N. (2021, August 3). *Spam detection with logistic regression*. Medium. Retrieved May 1, 2022, from https://towardsdatascience.com/spam-detection-with-logistic-regression-23e3709e522

[12] Batsuuri, J. (2021, March 21). *Language processing with recurrent models*. Medium. Retrieved May 1, 2022, from https://medium.com/computronium/language-processing-with-recurrent-models-4b5b53c03f1