SMS Spam Detection Based on Character N-gram Models

Anjali Pankan, Matriculation Number: 11016821, Date of Submission: 15/02/2022, Task Number: 2, Instructor: Prof. Dr.-Ing. Milan Gnjatović, E-mail: anjali.pankan@gmail.com

Abstract

In this project, program for the SMS Spam Detection based on character n-gram models approach is explained. In order to implement the program, python programming language is used. For training and testing the model the program used UCI Machine Learning Repository SMS Spam Collection dataset. In order to use the dataset for n-gram model pre-processing is performed to clean the dataset. The dataset is then divided into training and test sets by taking 80% of dataset as training dataset and remaining as test dataset. The training procedure is performed by calculating the probability of all bigrams and trigrams in the training dataset. The testing procedure then determines the probability of all SMS messages in the test dataset. The program is found to produce good results for both bigram and trigram models.

Index Terms

N-gram Models, UCI SMS Spam Collection Dataset.

I. Introduction

The term machine learning was first put in place by Arthur Samuel in 1959. Machine learning is a popular field in computer science in which computers are equipped to learn from past data. There are many machine learning algorithms available to generate mathematical models to create predictions based on historical data or information. Machine learning has many applications such as image recognition, speech recognition, recommender system, email filtering etc.. In order to construct mathematical models, the machine learning algorithms uses sample historical data, which is referred to as training data [1]. As more and more information is provided to the model, it produces higher performance [2]. This helps making predictions and decisions without the need of explicit programming. A well known probabilistic language model in machine learning is the n-gram model, which is a simple model that can help make quick prediction of the next item in a sequence based on some number of previous items [3].

One of the commonly used machine learning datasets is the UCI SMS spam collection dataset. It is a large dataset consisting of a collection of both spam and normal (ham) messages [4]. The main aim of this project is to classify SMS messages to ham or spam using n-gram approach by utilizing UCI SMS spam collection dataset.

II. N-gram Models

An N-gram model is a probabilistic language model which predicts the occurrence of a word in a sequence based on the occurrence of its N - 1 previous words [5]. N-gram models express the linguistic intuition that the order of words in a sequence is not random [6]. They are largely used in applications such as natural language processing, speech recognition etc. The N-gram model not only gives the probability of occurrence of a word preceding a sequence of words, but also gives the probability of a given set of words [6].

III. Data pre-processing

In this section, the data pre-processing for the SMS Spam detection program using character n-gram approach is explained. In order to train and test the SMS Spam detection program, the UCI SMS spam collection dataset has been used [4]. The acronym UCI stands for UC Irvine [7]. The UCI SMS spam collection dataset of ham and spam is available as tab-separated values (tsv) format. It is composed of a single text file dataset containing both spam and ham, which the program is used for training and testing.

In order to do the pre-processing the program used an alphabet of symbols of which majority of them are occurring in the dataset. The alphabet includes lowercase English alphabet characters such as: 'a', 'b', 'c' until 'z' and digits from 0 to 9. The alphabet also contains special symbols like space(' '), '/', '?', '!' and '.' along with special symbols 'α', 'β' and 'γ' to represent start, end and unknown character for a sequence respectively. Listing 1 shows the alphabet used in the project.

Listing 1: Alphabet used in the project

```
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm','n', 'o','p', 'q',
    'r', 's', 't', 'u', 'v', 'w', 'x','y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8',
    '9', ' ', '/', '?', '!', '.', 'α', 'β', 'γ']
```

The dataset consist of 5572 sample SMS messages, in which some of them are spam messages where as others are normal (ham) messages. The number of ham messages are high compared to the spam messages. For loading the UCI SMS spam collection dataset into the program, "Pandas" library is being used. The Pandas is a python library, which is used for data analysis. Listing 2 illustrates importing of Pandas library into the program and loading of UCI SMS spam collection dataset using it's read_csv() function. The read_csv() function takes as an argument the path to the dataset file to be loaded as well as other optional parameters. In this project, the program expects the dataset to be kept in the same directory as that of the program. Additional parameters such as "sep" and "header" are also given with values *"tab"* and $None$ to indicate that the data is tab separated and there is no header for the dataset. It is important to note that the read_csv() function loads data in a columnar fashion. That means, the loaded data can be accessed as column rather than rows. For example, data[0] gives the first column of dataset which contains all 5572 SMS types whose actual messages are available in the second column.

Listing 2: Code for loading UCI SMS spam collection dataset using Pandas library

```
import pandas as pd

# Load datasets.
data = pd.read_csv("./SMSSpamCollection", sep='\t', header=None)
```

Before using the dataset for training and testing, it needs to be processed properly. The pre-processing consists of firstly changing all uppercase letter to lowercase letters. Then all unknown symbols present in the SMS messages of dataset, which are not declared in the alphabet, are replaced with the unknown character of alphabet ('γ'). Finally, for bigram model, the start ('α') and end ('β') character are placed at the start and end of each SMS message of the dataset. Please note that for trigram model two start characters needs to be placed at the beginning and two end characters needs to be places at the end of the messages. Listing 3 represents the code for pre-processing the dataset for bigram model.

Listing 3: Code for pre-processing the UCI SMS spam collection dataset for bigram model

```
# preprocess the dataset
def data_preprocessing(sms):
  # convert to lowercase.
  sms = sms.lower()

  # replace all unknow character to 'γ', where 'γ' is
  # taken as replacement for unknown character.
  sms = "".join([ c if c in alphabet else 'γ' for c in sms ])

  # append start ('α') and end ('β') symbols.
  sms = 'α' + sms + 'β'

  return sms

for i in range(total_num_sms):
  data[1][i] = data_preprocessing(data[1][i])
```

Since there are no separate training and test dataset available, the dataset is divided into two parts in which initial 80% of messages are used for training and remaining 20% are used for testing. This is realized by calculating minimum and maximum index values for training and test data rows. This calculation is shown in Listing 4.

Listing 4: Code for calculating min and max indices of training and test data rows

```
# determine min and max indices of train data.
train_min_index = 0
train_max_index = int((total_num_sms * 80) / 100)

# determine min and max indices of test data.
test_min_index = train_max_index + 1
test_max_index = total_num_sms
```

IV. Training procedure

Once the dataset is divided into training and test sets, it is possible to use the training set for determining n-gram model. In this project both bigram and trigram models are evaluated. As already seen in the pre-processing section, the program used 80% of the dataset for training the model. That means a total of 4458 SMS messages are used for training. The first column (data[0]) contains the classes of the SMS messages. The program uses python library called NumPy to manipulate dataset

arrays. This library is very useful for processing array type objects in python. The number of classes (num_classes) is obtained by finding length of the array returned by NumPy's built in function unique() on the data[0] column. Which returns all the unique elements in the data[0] column. The dataset contain only two classes which are "ham" and "spam". Listing 5 shows num_classes calculation.

Listing 5: Code for determining number of classes

```
import numpy as np

# Determine number of classes.
num_classes = len(np.unique(data[0]))
```

For the character bigram, the model is determined by finding conditional probabilities of each character bigram present in the training dataset for both ham and spam classes. The probability of occurrence of character c_m immediately after character c_{m-1} is given by the Equation 1, where $C(c_{m-1}c_m)$ is the number of occurrences of character bigram $c_{m-1}c_m$ and $C(c_{m-1}\sqcup)$ is the number of occurrences of bigram in which c_{m-1} is the first word, and second word is arbitrary [6].

$$P(c_m|c_{m-1}) = \frac{C(c_{m-1}c_m)}{C(c_{m-1}\sqcup)} \tag{1}$$

From the Equation 1 it is clear that the model needs to find the occurrence of each character of the alphabet in the training set for both ham and spam. Listing 6 shows the calculation of count for each character of the alphabet in the training dataset for ham and spam. Here, two python dictionaries are used to save counts of each character. Initially, the dictionary values has been set to zero for each character. Then for each SMS message in the training dataset the messages are processed one-by-one and the corresponding dictionary values for the characters are increased based on the SMS type in the corresponding dictionary. At the end, each character count for ham and spam are obtained.

Listing 6: Code for determining number of alphabet characters in the traing set

```
alphabet_count_spam = dict()
alphabet_count_ham = dict()

# count alphabet characters in ham and spam training dataset.
def count_trainingset_alphabet_spam_ham():
  # first set items in alphabet to 0.
  for char in alphabet:
    alphabet_count_spam[char] = 0
    alphabet_count_ham[char] = 0

  # count each item of alphabet in training data.
  for i in range(train_min_index, train_max_index):
    sms = data[1][i]
    if data[0][i] == "spam":
      for c in sms:
        alphabet_count_spam[c] = alphabet_count_spam[c] + 1
    else:
      for c in sms:
        alphabet_count_ham[c] = alphabet_count_ham[c] + 1

count_trainingset_alphabet_spam_ham()
```

From Equation 1 it can be also noted that for the bigram model the character bigram count needs to be also determined. Listing 7 shows the calculation of count of each bigram in the training dataset for both ham and spam messages. Here also two python dictionaries are created to hold bigram counts. The function then initialize this dictionaries to zero for all possible bigrams. It is to be noted that for an alphabet of size n there will be $n{\times}n$ bigrams possible. In this project the alphabet contains 44 characters and so there will be 1,936 possible bigrams. For each SMS message the bigram is determined and the corresponding count in the respective dictionary is incremented. In a similar way character trigram counts can be determined and stored in a dictionary. Please note that for an alphabet of size n there will be $n{\times}n{\times}n$ possible trigrams.

TABLE I: Probability of some random character bigrams in the training set

	ham	spam
αg	0.028243	0.028044
bj	0.001043	0.002497
mt	0.003617	0.012805
gw	0.000444	0.001378
ig	0.032764	0.016304
6b	0.005952	0.000815

TABLE II: Probability of some random character trigrams in the training set

	ham	spam
3pγ	0.022727	0.021739
q/?	0.022727	0.022727
2cm	0.021739	0.02
12r	0.016666	0.003205
b1o	0.022727	0.021739
x04	0.022727	0.022727

Listing 7: Code for determining number of character bigram in the traing set

```
bigram_count_spam = dict()
bigram_count_ham = dict()

# count character bigram in ham and spam training dataset.
def count_trainingset_bigram_spam_ham():
  # first set bigram count to 0.
  for first_char in alphabet:
    for second_char in alphabet:
      bigram = first_char + second_char
      bigram_count_spam[bigram] = 0
      bigram_count_ham[bigram] = 0

  # count each bigram in training data.
  for i in range(train_min_index, train_max_index):
    sms = data[1][i]
    if data[0][i] == "spam":
      for char in range(len(sms) - 1):
        bigram = sms[char] + sms[char + 1]
        bigram_count_spam[bigram] = bigram_count_spam[bigram] + 1
    else:
      for char in range(len(sms) - 1):
        bigram = sms[char] + sms[char + 1]
        bigram_count_ham[bigram] = bigram_count_ham[bigram] + 1

count_trainingset_bigram_spam_ham()
```

An important factor in determining bigram count is that some of the bigrams might never appear in the training dataset. In such situations it is required to do smoothing of the calculated probability. In this project Add-k smoothing is used, where $0 < k < 1$. Hence, for bigram model the probability calculation can be rewritten as shown in Equation 2, where V is the alphabet [6]. For this project the k value used is 0.5.

$$P(c_m|c_{m-1}) = \frac{C(c_{m-1}c_m) + k}{C(c_{m-1}) + k|V|} \tag{2}$$

For character trigram model the probability can be expressed as in Equation 3.

$$P(c_m|c_{m-2}c_{m-1}) = \frac{C(c_{m-2}c_{m-1}c_m) + k}{C(c_{m-2}c_{m-1}) + k|V|} \tag{3}$$

Listing 8 shows the probability calculation for each bigram in the training dataset for both ham and spam messages. Here again two dictionaries are taken to store probabilities of each bigrams for ham and spam messages.

Table I and II respectively shows probabilities evaluated for some of the bigrams and trigrams from the training dataset.

Listing 8: Code for calculating probabilities of each bigram

```
# taking k as 0.5 for finding prediction.
```

```
k = 0.5

prob_bigram_ham = dict()
prob_bigram_spam = dict()

# determine probabilities of ech bigrams
def prob_bigram_spam_ham():
  for bigram, count in bigram_count_spam.items():
    bigram_first_char = bigram[0]
    prob_bigram_spam[bigram] = (count + k) / (alphabet_count_spam[bigram_first_char] + (k *
        len(alphabet)))

  for bigram, count in bigram_count_ham.items():
    bigram_first_char = bigram[0]
    prob_bigram_ham[bigram] = (count + k) / (alphabet_count_ham[bigram_first_char] + (k*
        len(alphabet)))

prob_bigram_spam_ham()
```

V. Testing procedure

Testing the model created during the training procedure is carried out using the test dataset identified during the data pre-processing section. The test dataset consists of around 1114 SMS messages. Like the training dataset the test dataset contains data[0] column the class of the SMS message (ham or spam) and column data[1] contains the actual SMS message. For testing the model, each test image data excluding the first column value is read row by row and passed to the predict function.

The probability of a character sequence c_1, c_2, ..., c_m is given by the chain rule of probability as given in Equation 4, where $< s >$ and $< /s >$ are special symbols appended at the beginning and end of the sequence [6].

$$\begin{aligned} P(< s > w_1 w_2 ... w_m < /s >) = & P(w_1 | < s >) \\ & P(w_2 | < s > w_1) \\ & P(w_3 | < s > w_1 w_2) \\ & ... \\ & P(w_m | < s > w_1 w_2 ... w_{m1}) \\ & P(< /s > | < s > w_1 w_2 ... w_{m1} w_m) \end{aligned} \tag{4}$$

By applying the Markovian assumption for bigram, the above equation can be written as shown in Equation 5 [6].

$$P(< s > w_1 w_2 ... w_m < /s >) \approx P(w_1 | < s >) P(w_2 | w_1) P(w_3 | w_2) ... P(w_m | w_{m-1}) P(< /s > | w_m) \tag{5}$$

Given the bigram probabilities estimated during the training procedure for ham and spam, it should be possible to use Equation 5 to determine probability of a test SMS message. A test SMS message is assigned to one of the classes (ham or spam) if the calculated probability is higher compared to other class.

A problem with multiplication of a sequence of probabilities is that it can lead to arithmetic underflow [8]. In order to eliminate this issue, the log probability can be used. Equation 4 shows the log probability representation of the above equation.

$$\begin{aligned} log\, P(< s > w_1 w_2 ... w_m < /s >) \approx & log\, P(w_1 | < s >) + log\, P(w_2 | w_1) + log\, P(w_3 | w_2) + ... \\ & ... + log\, P(w_m | w_{m-1}) + log\, P(< /s > | w_m) \end{aligned} \tag{6}$$

Listing 9 shows the calculation of log probability of bigram model in function log_probability() where x represents the test SMS message and "spam_ham" says whether the probability calculation is to be done against spam or ham model.

Listing 9: Code for calculating log probabilty of test SMS message for bigram model

```
# determine log probability of a test SMS message.
# used log to eliminate arithmatic undeflow.
def log_probability(x, spam_ham):
  log_prob = 0
  if spam_ham == "spam":
    for char in range(len(x) - 1):
      bigram = x[char] + x[char + 1]
      log_prob = log_prob + math.log(prob_bigram_spam[bigram])
  else:
    for char in range(len(x) - 1):
```

```
      bigram = x[char] + x[char + 1]
      log_prob = log_prob + math.log(prob_bigram_ham[bigram])

  return log_prob
```

Finally, the predict() function shown in Listing 10 returns the class that produces the maximum probability.

Listing 10: Code for predicting test SMS message for ham or spam

```
# predict the test SMS message for ham or spam.
def predict(x):
  spam_predict = log_probability(x, "spam")
  ham_predict = log_probability(x, "ham")

  if spam_predict >= ham_predict:
    return "spam"
  else:
    return "ham"

# predict test SMS message in each row in the test dataset.
test_data_count = test_max_index - test_min_index
predicted_classes = np.empty((test_data_count), dtype="<U10")
for n in range(test_min_index, test_max_index):
  index = n - test_min_index
  predicted_classes[index] = predict(data[1][n])
```

The predicted values are stored in an array called ”predicted_classes”. This array has a size equal to the number of rows in the test dataset and the predicted class values are stored at the same index location as that of the test data.

VI. Evaluation

Once the ”predicted_classes” array is available it should be possible to evaluate the prediction compared to the actual class values. From the testing dataset, the array containing actual class values can be obtained by taking column data[0] for the test portion. The accuracy of prediction is calculated by comparing the two arrays and counting the places where the two arrays are equal, after that dividing it with the total test instance count, as shown in Listing 11. For the bigram model the accuracy obtained is 0.970377 (97.03%) whereas for the trigram model the accuracy obtained is 0.9883 (98.83%).

Listing 11: Code for determining accuracy

```
actual_classes = np.array(data[0])[test_min_index:test_max_index]

# determine accuracy of prediction compared to the actual values.
accuracy = 0
for i in range(len(predicted_classes)):
  if predicted_classes[i] == actual_classes[i]:
    accuracy = accuracy + 1

accuracy = accuracy / len(predicted_classes)
```

From the predicted and actual class arrays it is also possible to evaluate Precision and Recall values. The Precision represents the fraction of instances that systems assigned to class $c \in C$ that actually belongs to class c, as shown in Equation 7 [8].

$$P = \frac{truepositive}{truepositive + falsepositive} \tag{7}$$

Similarly, the Recall represents the fraction of instances that belong to class $c \in C$ that were actually classified as c, as shown in Equation 8 [8].

$$P = \frac{truepositive}{truepositive + falsenegative} \tag{8}$$

Using this information, the Precision and Recall values of each digit class is calculated and formed a Confusion Matrix using the function confusion_matrix(). Listing 12 shows the confusion_matrix() function.

TABLE III: Confusion Matrix for bigram

		Predicted classes	
		spam	ham
Actual classes	spam	139	27
	ham	6	942

TABLE IV: Confusion Matrix for trigram

		Predicted classes	
		spam	ham
Actual classes	spam	139	7
	ham	6	962

Listing 12: Code for calculating Confusion Matrix

```
# determine the confusion matrix.
def confusion_matrix(predicted_classes, actual_classes):
  cm = np.zeros((num_classes, num_classes), dtype=np.int32)

  for i in range(len(predicted_classes)):
    if predicted_classes[i] == "spam":
      predicted_class = 0
    else:
      predicted_class = 1

    if actual_classes[i] == "spam":
      actual_class = 0
    else:
      actual_class = 1

    cm[predicted_class][actual_class] = cm[predicted_class][actual_class] + 1

  return cm
```

For the test dataset, the programs produced Confusion Matrices for both bigram and trigram models, which has values as shown in Tables III and IV. From the Confusion Matrix it is possible to calculate the Macroaverage Precision and Recall using the Equations 9 and 10. Listing 13 shows the code for finding Macroaverage Precision and Recall with functions p_macroaverage() and r_macroaverage() respectively.

$$P_{macroaverage} = \frac{1}{|C|} \sum_{c \in C} P_c \tag{9}$$

$$R_{macroaverage} = \frac{1}{|C|} \sum_{c \in C} R_c \tag{10}$$

Listing 13: Code for calculating Macroaverage Precision and Recall

```
# determine precision macroaverage.
def p_macroaverage():
  p_macro = np.zeros((num_classes), dtype=np.float64)

  for i in range(num_classes):
    sum_t = 0
    frac = confusion_matrix[i][i]
    for j in range(num_classes):
      sum_t = sum_t + confusion_matrix[j][i]

    p_macro[i] = frac / sum_t

  return np.sum(p_macro) / num_classes

# determine recall macroaverage.
def r_macroaverage():
  r_macro = np.zeros((num_classes), dtype=np.float64)

  for i in range(num_classes):
```

```
predicted classes of test SMS messages
['ham' 'ham' 'spam' ..., 'ham' 'ham' 'ham']
actual classes of test SMS messages
['ham' 'ham' 'spam' ..., 'ham' 'ham' 'ham']
accuracy:  0.9703770197486535
[[139  27]
 [  6 942]]
p_macroaverage:  0.965378456283
r_macroaverage:  0.915510141833
F1-score:  0.939783215571
```

Fig. 1: Evaluation result part of program output for bigram model

```
predicted classes of test SMS messages
['ham' 'ham' 'spam' ... 'ham' 'ham' 'ham']
actual classes of test SMS messages
['ham' 'ham' 'spam' ... 'ham' 'ham' 'ham']
accuracy:  0.9883303411131059
[[139   7]
 [  6 962]]
p_macroaverage:  0.9756983737233551
r_macroaverage:  0.972928223706555
F1-score:  0.9743113297048005
```

Fig. 2: Evaluation result part of program output for trigram model

```
    sum_t = 0
    frac = confusion_matrix[i][i]
    for j in range(num_classes):
      sum_t = sum_t + confusion_matrix[i][j]

    r_macro[i] = frac / sum_t

  return np.sum(r_macro) / num_classes
```

In case of the UCI SMS spam collection dataset, for bigram model, the program calculated Macroaverage Precision as 0.965378456283 and Macroaverage Recall as 0.915510141833. For the trigram model it calculated Macroaverage Precision as 0.975698373723 and Macroaverage Recall as 0.972928223707. Finally, the F1-score, which is the harmonic mean of the above two values are calculated using the Equation 11. The F1-score calculated by the programs against the test dataset is 0.939783215571 and 0.974311329705 for bigram and trigram models respectively. The evaluation result of the program execution is shown in Figures 1 and 2.

$$F_{1,macroaverage} = \frac{2\,P_{macroaverage} R_{macroaverage}}{P_{macroaverage} + R_{macroaverage}} \tag{11}$$

Finally, the model is tested against a randomly taken spam and a ham message. The messages are listed as below:

- spam_msg = ”Save up to Rs 150 this Valentine’s Day ! ”
- ham_msg = ”We recommend that you change your password immediately to keep your account secure.”

The program correctly predicts the message class for the above messages as below:

- message: ” Save up to Rs 150 this Valentine’s Day ! ” is : spam
- message: ” We recommend that you change your password immediately to keep your account secure. ” is : ham

VII. Conclusion

In this project, programs for the SMS Spam Detection based on character n-gram models approach is successfully realized. The programs are written using python programming language. The project contains two programs to evaluate against bigram and trigram. For a given alphabet, the programs achieved accuracies 97.03% and 98.83% for bigram and trigram respectively. The programs could also predict a randomly taken SMS message as spam or ham successfully.

REFERENCES

[1] *Machine-learning*. github.com. [Online]. Available: https://github.com/AnjaliPankan/Machine-learning
[2] *Machine Learning Tutorial*. javatpoint.com. [Online]. Available: https://www.javatpoint.com/machine-learning
[3] *n-gram*. wikipedia.org. [Online]. Available: https://en.wikipedia.org/wiki/N-gram
[4] *SMS Spam Collection Data Set*. UCI Machine Learning Repository. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/sms+spam+collection
[5] *An Introduction to N-grams: What Are They and Why Do We Need Them?*. xrds.acm.org. [Online]. Available: https://blog.xrds.acm.org/2017/10/introduction-n-grams-need/
[6] *N-gram Models*. Milan Gnjatović. [Online]. Available: http://gnjatovic.info/misc/ngram.models.pdf
[7] *UCI Machine Learning Repository*. re3data.org. [Online]. Available: https://www.re3data.org/repository/r3d100010960
[8] *Naive Bayesian Classification*. Milan Gnjatović. [Online]. Available: http://gnjatovic.info/misc/naive.bayesian.classification.pdf