# Sentiment Analysis On Movie Reviews

Yassir Arroud

# TABLE OF CONTENTS


01 INTRODUCTION

First things first! Lets explore our data & prepare it for modelisations

02 Neural Bow Model

Here We ‘re going to create our bag of words model

03 S.A Model

Thanks to Keras API, we will develop a MLP model for binary classifications

04 Predictions

Now that our model is ready, we can do some predictions

# INTRODUCTION

In this section, we're going to load the “**movie review data for sentiment analysis** ” (polarity.tar.gz, 3MB), load it , do some cleaning things and save our vocabulary in a text file  any for later use.

# CONTENTS OF THIS TEMPLATE

- Brief definition of our dataset
- Import libraires
- Load & Clean Data functions
- Vocabulary of the dataset
- Save prepared Data

# Librairies and data

- **Brief definition of our dataset**

The Movie Review Data is a collection of movie reviews retrieved from the imdb.com website in the early 2000s by Bo Pang and Lillian Lee. The reviews were collected and made available as part of their research on natural language processing. The reviews were originally released in 2002, but an updated and cleaned up version was released in 2004, referred to as v2.0. The dataset is comprised of 1,000 positive and 1,000 negative movie reviews drawn from an archive of the rec.arts.movies.reviews newsgroup hosted at IMDB. The authors refer to this dataset as the polarity dataset.

Our data contains 1000 positive and 1000 negative reviews all written before 2002, with a cap of 20 reviews per author (312 authors total) per category. We refer to this corpus as the polarity dataset.

# Librairies and data

- **Brief definition of our dataset**

We can download the dataset from here:  Movie Review Polarity Dataset (review

polarity.tar.gz, 3MB).  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz , after unzipping this file we got a repertory “txt sentoken” that contains 2 main folders “neg” & “pos” ; Reviews are stored
one per file with a naming convention from cv000 to cv999 for each of neg and pos.

So let’s get started with our notebook :

# Librairies and data

- **Import libraries and packages**

A good practice is to load all needed packages and libraries in the first cell of our notebook

```
import os
import re
import string
import nltk
import tensorflow
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from os import listdir
from nltk import word_tokenize
from nltk.corpus import stopwords
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
from collections import Counter
#
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.utils import plot_model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

Now, we create the functions « load_doc » and « clean_doc » that as expected allow us to read the file in mode reading and clean it as mentioned in the comments inside the code

# Librairies and data

- **Load and clean documents**

Now, we create the functions « load_doc » and « clean_doc » that as expected allow us to read the file in mode reading and clean it as mentioned in the comments inside the code

```
def load_doc(filename):
    file = open(filename, 'r')
    text = file.read()
    file.close()
    return text

# turn a doc into clean tokens
def clean_doc(doc):
    tokens = doc.split()
    # for char filtering, prepare regex
    re_punc = re.compile('[%s]'% re.escape(string.punctuation))
    # remove punctuation from each word
    tokens = [re_punc.sub('', w) for w in tokens]
    # remove  non alphabetic
    tokens = [word for word in tokens if word.isalpha()]
    # No stop words!
    stop_words = set(stopwords.words('english'))
    tokens = [w for w in tokens if not w in stop_words]
    # No short tokens!
    tokens = [word for word in tokens if len(word) > 1]
    return tokens
```

# Librairies and data

- **Vocabulary of the dataset**

The add_to_vocab() function takes as an argument a dictionary (often a counter) as a vocabulary where we're going to store the cleaned data of our file. Meanwhile, the process_docs() function repeats the process of add_to_voocab to all the files in a named directory ( In our case, we'll use it the neg & pos directories.

```
def doc_to_line(filename, vocab):
    # load the doc
    doc = load_doc(filename)
    # clean doc
    tokens = clean_doc(doc)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    return ''.join(tokens)

# load all docs in a directory
def process_docs(directory, vocab):
    lines = list()
    for filename in listdir(directory):
        if not filename.endswith(".txt"):
            next
        path = directory + '/'+ filename
        line = doc_to_line(path, vocab)
        lines.append(line)
        return lines
```

```
def add_to_vocab(filename, vocab):
    doc = load_doc(filename)
    tokens = clean_doc(doc)
    vocab.update(tokens)

def process_docs(directory, vocab):
    for filename in listdir(directory):
        if not filename.endswith(".txt"):
            next
        path = directory + '/'+ filename
        add_to_vocab(path, vocab)
```

# Librairies and data

Running the example creates a vocabulary with all documents in the dataset, including positive and negative reviews. We can see that there are a little over 46,000 unique words across all reviews and the top 3 words are film, one, and movie.

```
Entrée [10]: vocab = Counter()
             # add all docs to vocabulary
             process_docs('txt_sentoken/neg', vocab)
             process_docs('txt_sentoken/pos', vocab)

Entrée [11]: len(vocab)

  Out[11]: 46557

Entrée [12]: print(vocab.most_common(50))

[('film', 8860), ('one', 5521), ('movie', 5440), ('like', 3553), ('even', 2555), ('good', 2320), ('time', 2283), ('story',
2118), ('films', 2102), ('would', 2042), ('much', 2024), ('also', 1965), ('characters', 1947), ('get', 1921), ('character',
1906), ('two', 1825), ('first', 1768), ('see', 1730), ('well', 1694), ('way', 1668), ('make', 1590), ('really', 1563), ('li
ttle', 1491), ('life', 1472), ('plot', 1451), ('people', 1420), ('movies', 1416), ('could', 1395), ('bad', 1374), ('scene',
1373), ('never', 1364), ('best', 1301), ('new', 1277), ('many', 1268), ('doesnt', 1267), ('man', 1266), ('scenes', 1265),
('dont', 1210), ('know', 1207), ('hes', 1150), ('great', 1141), ('another', 1111), ('love', 1089), ('action', 1078), ('go',
1075), ('us', 1065), ('director', 1056), ('something', 1048), ('end', 1047), ('still', 1038)]
```

# Librairies and data

- **Save our prepared data**

Perhaps the least common words, those that only appear once across all reviews, are not predictive. Perhaps some of the most common words are not useful too. Generally, words that only appear once or a few times across 2,000 reviews are probably not predictive and can be removed from the vocabulary.

Entrée [13]:
```
threshold = 2
tokens = [k for k,v in vocab.items() if v >= threshold]
print(len(tokens))
```

```
27139
```

Entrée [14]:
```
""" I like to save the vocabulary as ASCII with one
word per line. Below defines a function called save_list() to save a list of items, in this case,
tokens to file, one per line."""
def save_list(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()
```

Entrée [15]:
```
save_list(tokens, "vocabulary.txt")
```

Now we got a text file named “vocabulary.txt” that contains our vocabulary !

# B-o-w Model

A popular technique for developing sentiment analysis models is to use a bag-of-words model that transforms documents into vectors where each word in the document is assigned a score. So let's develop a Neural Bag-of-Words !

# B-o-w Model

So let's develop a Neural Bag-of-Words !  A popular technique for developing sentiment analysis models is to use a bag-of-words model that transforms documents into vectors where each word in the document is assigned a score.

In this section, we will look at how we can convert each review into a representation that we can provide to a Multilayer Perceptron model. A bag-of-words model is a way of extracting features from text so the text input can be used with machine learning algorithms like neural networks. Each document, in this case a review, is converted into a vector representation. The number of items in the vector representing a document corresponds to the number of words in the vocabulary.

This section is divided into 2 steps:

1. Converting reviews to lines of tokens.

2. Encoding reviews with a bag-of-words model representation.

# B-o-w Model

## 1) from review text to line of tokens

Before we can convert reviews to vectors for modeling, we must first clean them up. This involves loading them, performing the cleaning operation developed above, filtering out words not in the chosen vocabulary, and converting the remaining tokens into a single string or line ready for encoding. So after we load our data and clean it

```
def process_docs(directory, vocab):
    lines = list()
    # walk through all files in the folder
    for filename in listdir(directory):
        # skip any reviews in the test set
        if filename.startswith('cv9'):
            continue
        # create the full path of the file to open
        path = directory + '/'+ filename
        # load and clean the doc
        line = doc_to_line(path, vocab)
        # add to list
        lines.append(line)
    return lines
```

```
# our training data is evolving buddy!!
def load_clean_data(vocab):
    neg_lines = process_docs('txt_sentoken/neg', vocab)
    pos_lines = process_docs('txt_sentoken/pos', vocab)
    docs = neg_lines + pos_lines
    # prepare labels for our training data
    labels = [ 0 for _ in range(len(neg_lines))] +  [1 for _ in range(len(pos_lines))]
    return docs, labels
```

# B-o-w Model

## 1) from review text to line of tokens

So our labels here are our response data, and the docs are our train data ( Xtrain)

```
]: # load the vocabulary
   vocab_filename = 'vocab.txt'
   vocab = load_doc(vocab_filename)
   vocab = set(vocab.split())
   vocab = set(vocab)
   # load all training reviews
   docs, labels = load_clean_data(vocab)
   # summarize what we have
   print(len(docs), len(labels))
```

```
1800 1800
```

```
]: #print(docs[:50])
   len(docs)
```

```
1800
```

# B-o-w Model

## 2) Movie Reviews to Bag-of-Words Vectors

```
# load and clean a dataset
def load_clean_data(vocab, is_train):
    # load documents
    neg_lines = process_docs('txt_sentoken/neg', vocab, is_train)
    pos_lines = process_docs('txt_sentoken/pos', vocab, is_train)
    docs = neg_lines + pos_lines
    # prepare labels
    labels = [0 for _ in range(len(neg))] + [1 for _ in range(len(pos))]
    return docs, labels

# fit a tokenizer
def create_tokenizer(lines):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(lines)
    return tokenizer
```

```
# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
```

# B-o-w Model

## 2) Movie Reviews to Bag-of-Words Vectors

the Tokenizer class is convenient and will easily transform documents into encoded vectors. First, the Tokenizer must be created, then fit on the text documents in the training dataset. In this case, these are the aggregation of the positive lines and negative lines arrays developed in the previous section

```
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
print(Xtrain.shape, Xtest.shape)
```

```
(1800, 25768) (200, 25768)
```

Running the example prints both the shape of the encoded training dataset and test dataset with 1,800 and 200 documents respectively, each with the same sized encoding vocabulary (vector length).

# S.A Model

In this section, we will develop Multilayer Perceptron (MLP) models to classify encoded documents as either positive or negative. The models will be simple feedforward network models with fully connected layers called Dense in the Keras deep learning library.

# CONTENTS OF THIS TEMPLATE

This section is divided into 3 sections :

1. First sentiment analysis model

2. Comparing word scoring modes

# S.A Model

## 1) First sentiment analysis model

The model will have an input layer that equals the number of words in the vocabulary, and in turn the length of the input document

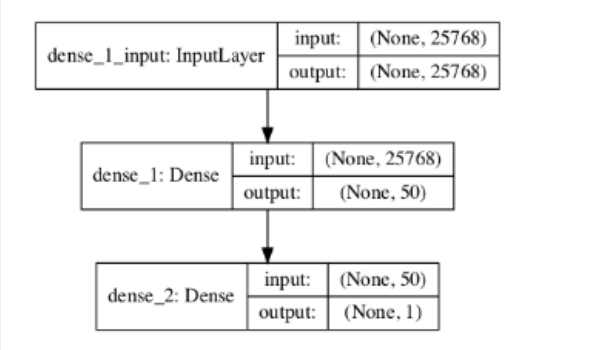
We will use a single hidden layer with 50 neurons and a rectified linear activation function. The output layer is a single neuron with a sigmoid activation function for predicting 0 for negative and 1 for positive reviews. The network will be trained using the efficient Adam implementation of gradient descent and the binary cross entropy loss function, suited to binary classification problem.

```
# define the model
def define_model(n_words):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # summarize defined model
    model.summary()
    plot_model(model, to_file='model.png', show_shapes=True)
    return model
```

# S.A Model

## 1) First sentiment analysis model

```
# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# create the tokenizer
tokenizer = create_tokenizer(train_docs)
# encode data
Xtrain = tokenizer.texts_to_matrix(train_docs, mode='freq')
Xtest = tokenizer.texts_to_matrix(test_docs, mode='freq')
# define the model
n_words = Xtest.shape[1]
model = define_model(n_words)
# fit network
model.fit(Xtrain, ytrain, epochs=10, verbose=2)
# evaluate
loss, acc = model.evaluate(Xtest, ytest, verbose=0)
#print('Test Accuracy: %f'% (acc*100))
```




| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense_1 (Dense) | (None, 50) | 1288450 |
| dense_2 (Dense) | (None, 1) | 51 |

Total params: 1,288,501
Trainable params: 1,288,501
Non-trainable params: 0

# S.A Model

## 1) First sentiment analysis model

We can see that the model easily fits the training data within the 10 epochs, achieving close to 100% accuracy. Evaluating the model on the test dataset, we can see that model does well, achieving an accuracy of above 87%, well within the ballpark of low-to-mid 80s seen in the original pape

```
...
Epoch 6/10
0s - loss: 0.5319 - acc: 0.9428
Epoch 7/10
0s - loss: 0.4839 - acc: 0.9506
Epoch 8/10
0s - loss: 0.4368 - acc: 0.9567
Epoch 9/10
0s - loss: 0.3927 - acc: 0.9611
Epoch 10/10
0s - loss: 0.3516 - acc: 0.9689

Test Accuracy: 87.000000
```

# S.A Model

## 2) Comparing Word Scoring Modes

The texts to matrix() function for the Tokenizer in the Keras API provides 4 different methods for scoring words; they are:

- binary Where words are marked as present (1) or absent (0).
- count Where the occurrence count for each word is marked as an integer.
- tfidf Where each word is scored based on their frequency, where words that are common across all documents are penalized.
- freq Where words are scored based on their frequency of occurrence within the document

We can evaluate the skill of the model developed in the previous section fit using each of the 4 supported word scoring modes. This first involves the development of a function to create an encoding of the loaded documents based on a chosen scoring model. The function creates the tokenizer, fits it on the training documents, then creates the train and test encodings using the chosen model. The function prepare data() implements this behavior given lists of train and test documents.

```
# prepare bag-of-words encoding of docs
def prepare_data(train_docs, test_docs, mode):
    # create the tokenizer
    tokenizer = Tokenizer()
    # fit the tokenizer on the documents
    tokenizer.fit_on_texts(train_docs)
    # encode training data set
    Xtrain = tokenizer.texts_to_matrix(train_docs, mode=mode)
    # encode training data set
    Xtest = tokenizer.texts_to_matrix(test_docs, mode=mode)
    return Xtrain, Xtest
```

# S.A Model

## 2) Comparing Word Scoring Modes

evaluate_mode() takes encoded documents and evaluates the MLP by training it on the train set and estimating skill on the test set 10 times and returns a list of the accuracy scores across all of these runs

```
# evaluate a neural network model
def evaluate_mode(Xtrain, ytrain, Xtest, ytest):
    scores = list()
    n_repeats = 30
    n_words = Xtest.shape[1]
    for i in range(n_repeats):
    # define network
    model = Sequential()
    model.add(Dense(50, input_shape=(n_words,), activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    # compile network
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    # fit network
    model.fit(Xtrain, ytrain, epochs=10, verbose=2)
    # evaluate
    loss, acc = model.evaluate(Xtest, ytest, verbose=0)
    scores.append(acc)
    print('%d accuracy: %s'% ((i+1), acc))
    return scores
```

# S.A Model

## 2) Comparing Word Scoring Modes

The texts to matrix() function for the Tokenizer in the Keras API provides 4 different methods for scoring words; they are:

- binary Where words are marked as present (1) or absent (0).

- count Where the occurrence count for each word is marked as an integer.

- tfidf Where each word is scored based on their frequency, where words that are common
across all documents are penalized.

- freq Where words are scored based on their frequency of occurrence within the document

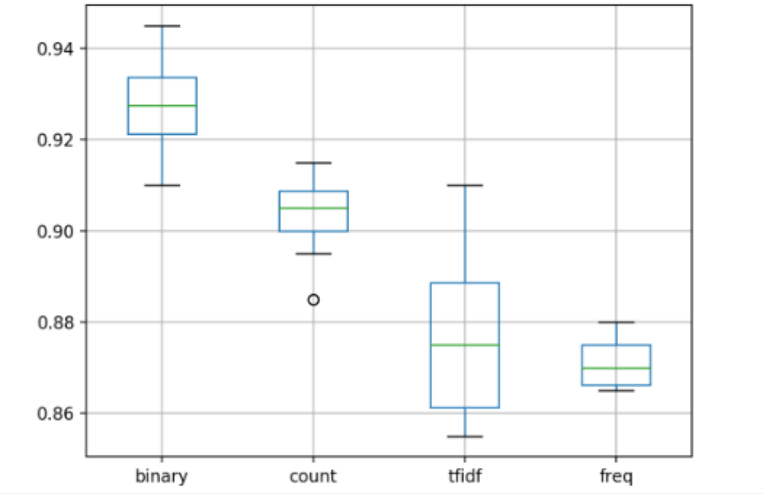
```
# load the vocabulary
vocab_filename = 'vocab.txt'
vocab = load_doc(vocab_filename)
vocab = set(vocab.split())
# load all reviews
train_docs, ytrain = load_clean_dataset(vocab, True)
test_docs, ytest = load_clean_dataset(vocab, False)
# run experiment
modes = ['binary', 'count', 'tfidf', 'freq']
results = DataFrame()
for mode in modes:
    # prepare data for mode
    Xtrain, Xtest = prepare_data(train_docs, test_docs, mode)
    # evaluate model on data for mode
    results[mode] = evaluate_mode(Xtrain, ytrain, Xtest, ytest)
    # summarize results
    print(results.describe())
    # plot results
    results.boxplot()
    plt.show()
```

# S.A Model

## 2) Comparing Word Scoring Modes

Execution the code above gave us the following results :
We can see that the mean score of both the count and binary methods appear to be better than freq and tfidf.

| | binary | count | tfidf | freq |
|---|---|---|---|---|
| count | 10.000000 | 10.000000 | 10.000000 | 10.000000 |
| mean | 0.927000 | 0.903500 | 0.876500 | 0.871000 |
| std | 0.011595 | 0.009144 | 0.017958 | 0.005164 |
| min | 0.910000 | 0.885000 | 0.855000 | 0.865000 |
| 25% | 0.921250 | 0.900000 | 0.861250 | 0.866250 |
| 50% | 0.927500 | 0.905000 | 0.875000 | 0.870000 |
| 75% | 0.933750 | 0.908750 | 0.888750 | 0.875000 |
| max | 0.945000 | 0.915000 | 0.910000 | 0.880000 |



# Predictions

Predicting Sentiment for New Reviews :
We can develop and use a final model to make predictions for new textual reviews. This is why we wanted the model in the first place. We will use the binary mode for scoring the bag-of-words model that was shown to give the best results in the previous section.

# Predictions

Predicting the sentiment of new reviews involves following the same steps used to prepare the test data. Specifically, loading the text, cleaning the document, filtering tokens by the chosen vocabulary, converting the remaining tokens to a line, encoding it using the Tokenizer, and making a prediction. We can make a prediction of a class value directly with the fit model by calling predict() that will return an integer of 0 for a negative review and 1 for a positive review.

All of these steps can be put into a new function called predict sentiment() that requires the review text, the vocabulary, the tokenizer, and the fit model and returns the predicted sentiment and an associated percentage or confidence-like output.

```
# classify a review as negative or positive
def predict_sentiment(review, vocab, tokenizer, model):
    # clean
    tokens = clean_doc(review)
    # filter by vocab
    tokens = [w for w in tokens if w in vocab]
    # convert to line
    line = ''.join(tokens)
    # encode
    encoded = tokenizer.texts_to_matrix([line], mode='binary')
    # predict sentiment
    yhat = model.predict(encoded, verbose=0)
    # retrieve predicted percentage and label
    percent_pos = yhat[0,0]
    if round(percent_pos) == 0:
        return (1-percent_pos), 'NEGATIVE'
    return percent_pos, 'POSITIVE'
```

# Predictions

We can now make predictions for new review texts. Below is an example with both a clearly positive and a clearly negative review using the simple MLP developed above with the frequency word scoring mode.

```
# test positive text
text = 'Best movie ever! It was great, I recommend it.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)'% (text, sentiment, percent*100))
# test negative text
text = 'This is a bad movie.'
percent, sentiment = predict_sentiment(text, vocab, tokenizer, model)
print('Review: [%s]\nSentiment: %s (%.3f%%)'% (text, sentiment, percent*10
```

# Predictions

We can now make predictions for new review texts. Below is an example with both a clearly positive and a clearly negative review using the simple MLP developed above with the frequency word scoring mode.

```
Review: [Best movie ever! It was great, I recommend it.]
Sentiment: POSITIVE (57.124%)
Review: [This is a bad movie.]
Sentiment: NEGATIVE (64.404%)
```

Ideally, we would fit the model on all available data (train and test) to create a final model and save the model and tokenizer to file so that they can be loaded and used in new software

Yassir Arroud