Sentimental analysis for marketing

Phase:04

SUBMITTED BY: Jeevitha C E

EMAIL ID: cdekambaram@gmail.com

The preprocessing technique for sentiment analysis of marketing was implemented in the previous phase. In this phase of development, we will continue the sentiment analysis solution by utilizing nlp techniques and generating insights.

With reference to the link below:

https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews?resource=download

Now we are going to employ NLP techniques:

i) Data splitting
ii) Tokenization
iii) LSTM model

Data splitting: When data is divided into two or more subgroups, this is known as data splitting. When using a two-part split, the data is usually evaluated or tested in part, while the model is trained in the other half. A crucial component of data science is data splitting, especially when building models with data.

Tokenization: Tokenization involves cutting the raw text into manageable pieces. Tokenization divides the original text into tokens, which are words or sentences. These tokens are useful for building the NLP model or comprehending the context. By examining the word order, tokenization assists in deciphering the text's meaning.

Data splitting and tokenization:

We start by splitting our DataFrame into a training and test lists. We use the train_test_split() function from the sklearn.model_selection module which allow to perform the splitting randomly with respect to the index of the DataFrame.

In [8]:

```
From sklearn.model_selection import train_test_split
```

Train_rev, test_rev, train_sent, test_sent = train_test_split(df[‘review’], df[‘sentiment’], test_size=0.1, random_state=42)

Print(‘\033[1m’ + ‘train_rev.shape:’ + ‘\033[0m’, train_rev.shape)

Print(‘\033[1m’ + ‘test_rev.shape:’ + ‘\033[0m’, test_rev.shape)

Print(‘\033[1m’ + ‘train_sent.shape:’ + ‘\033[0m’, train_sent.shape)

Print(‘\033[1m’ + ‘test_sent.shape:’ + ‘\033[0m’, test_sent.shape)

Train_rev.shape: (45000,)

Test_rev.shape: (5000,)

Train_sent.shape: (45000,)

Test_sent.shape: (5000,)

Next, we use the Tokenizer class from keras.preprocessing.text module to create a dictionary of the “dict_size” most frequent words present in the reviews (a unique integer is assigned to each word), and we print some of its attributes. The index of the Tokenizer is computed the same way no matter how many most frequent words we use later, see this post.

In [9]:

From keras.preprocessing.text import Tokenizer

Dict_size = 35000

Tokenizer = Tokenizer(num_words=dict_size)

Tokenizer.fit_on_texts(df[‘review’])

Print(‘\033[1m’ + ‘Dictionary size:’ + ‘\033[0m’, dict_size)

Print(‘\033[1m’ + ‘Length of the tokenizer index:’ + ‘\033[0m’, len(tokenizer.word_index))

Print(‘\033[1m’ + ‘Number of documents the tokenizer was trained on:’ + ‘\033[0m’, tokenizer.document_count, ‘\n’)

Print(‘\033[1m’ + ‘First 20 entries of the tokenizer index:’ + ‘\033[0m’)

Print(*list(tokenizer.word_index.items())[:20])

Dictionary size: 35000

Length of the tokenizer index: 125791

Number of documents the tokenizer was trained on: 50000

First 20 entries of the tokenizer index:

(‘movie’, 1) (‘film’, 2) (‘one’, 3) (‘like’, 4) (‘good’, 5) (‘time’, 6) (‘even’, 7) (‘would’, 8) (‘really’, 9) (‘story’, 10) (‘see’, 11) (‘well’, 12) (‘much’, 13) (‘get’, 14) (‘bad’, 15) (‘people’, 16) (‘great’, 17) (‘also’, 18) (‘first’, 19) (‘made’, 20)

We use the texts_to_sequences() function of the Tokenizer class to convert the training reviews and test reviews to lists of sequences of integers (tokens) “train_rev_tokens” and “test_rev_tokens”, and we store in the numpy array “seq_lengths” the lengths of the sequences included in “train_rev_tokens”.

In [10]:

Train_rev_tokens = tokenizer.texts_to_sequences(train_rev)

Test_rev_tokens = tokenizer.texts_to_sequences(test_rev)

Seq_lengths = np.array([len(sequence) for sequence in train_rev_tokens])

If the lengths of the sequences were normally distributed, then a given length could be considered small or large when outside the interval

Mean value of seq_lengths

±

2 standard deviations of seq_lengths,

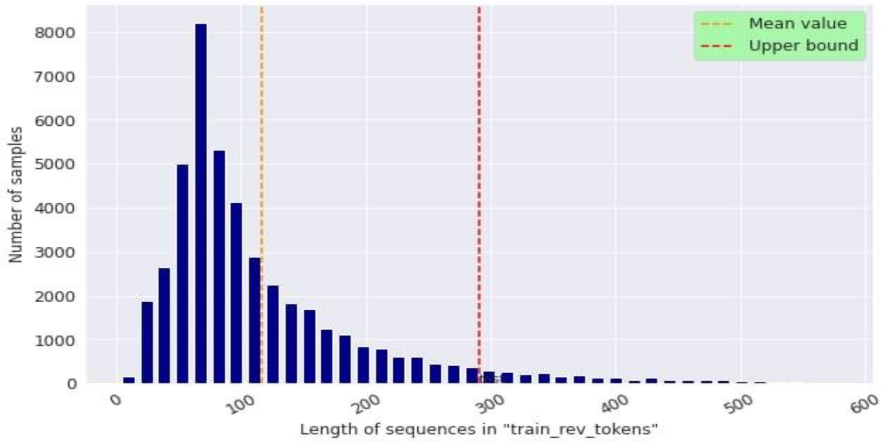
And lengths not belonging to this interval would only represent 5% of the elements of seq_lengths (see the 68–95–99.7 rule in statistics). Here, we follow this heuristics, and thus define an upper bound for the length of sequences accordingly.

(note that we use only the training set to define this upper bound, in order to avoid any data leakage or look-ahead bias)

In [11]:

```
# Storing in “upper_bound” our chosen upper bound for the length of sequences
# Computing the percentage of lengths smaller or equal than “upper_bound”

Upper_bound = int(np.mean(seq_lengths) + 2 * np.std(seq_lengths))
Percentage = stats.percentileofscore(seq_lengths, upper_bound)

Print(‘The value of upper_bound is %d and the percentage of sequences in “train_rev_tokens” \
Of length smaller or equal than upper_bound is %.2f%%.’ % (upper_bound, round(percentage,
2)))

# Histogram plot of the lengths of the sequences in “train_rev_tokens”

With sns.axes_style(“darkgrid”):

  _, hist = plt.subplots(figsize=(10,6))
  Hist.hist(seq_lengths[seq_lengths < 2*upper_bound], color=’darkblue’, bins=40, rwidth=0.7)
  Hist.axvline(np.mean(seq_lengths), color=’darkorange’, linestyle=’—‘, label=’Mean value’)
  Hist.axvline(upper_bound, color=’r’, linestyle=’—‘, label=’Upper bound’)

  Plt.xlabel(‘Length of sequences in “train_rev_tokens”’, size=’large’)
  Plt.ylabel(‘Number of samples’, size=’large’)
  Plt.text(upper_bound, 0, ‘test’)
  Plt.legend(fontsize=’large’, facecolor=’palegreen’)
  Plt.xticks(rotation=30)
  Plt.show()
```

The value of upper_bound is 291 and the percentage of sequences in “train_rev_tokens” of length smaller or equal than upper_bound is 94.56%



Using the pad_sequences() function from keras.preprocessing.sequence module, we transform "train_rev_tokens" and "test_rev_tokens" into 2D numpy arrays of shape (number of sequences, upper_bound). Sequences of length smaller (resp. larger) than "upper_bound" are extended (resp. truncated) to get a length equal to "upper_bound".

In [12]:

From keras.preprocessing.sequence import pad_sequences

Train_rev_pad = pad_sequences(train_rev_tokens, maxlen=upper_bound)

Test_rev_pad = pad_sequences(test_rev_tokens, maxlen=upper_bound)

Print('\033[1m' + 'train_rev_pad.shape:' + '\033[0m', train_rev_pad.shape)

Print('\033[1m' + 'test_rev_pad.shape:' + '\033[0m', test_rev_pad.shape, '\n')

Printing an example of review after padding

Idx_pad = random.randint(0, len(train_rev_pad)-1)

Print('\033[1m' + 'Review #%d after padding:' %idx_pad + '\033[0m' + '\n', train_rev_pad[idx_pad])

Train_rev_pad.shape: (45000, 291)

Test_rev_pad.shape: (5000, 291)

Review #2446 after padding:

```
[    0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0   297  2521     2    64  1389    17
  2593    11    83    29   108  2139   578   150  1748   406  2526  1567
     7  9163    64     5   330  1144     3     5   535   880  2529     2
  1072 31303   783     2  1240   187   308   202   308   270    56  2062
   216     2    81   240    40     2   173   666    37   332   375     2
    10     5  2593    17   899   995  4010  2593   286   690    10    28
   837    30    78   228  1489  2593 14333  1628 10076    96    40    28
    10   517  3188   113   333    25    40    25  1414  1690  2288    54
    83    29  1975  1578   795  1273 14132    21    69   330  1144     3
  5110   677     2]
```

LSTM: In the realm of deep learning, long short-term memory is an artificial recurrent neural network design. LSTM has feedback connections, in contrast to conventional feedforward neural networks. Unlike a conventional recurrent neural network, which retains all of the data, an LSM just stores the data in its short-term memory.

We start by importing some classes from Keras:

- The Sequential class from the keras.models API (to group a linear stack of layers into a model)
- The Embedding class from the keras.layers API (to turn positive integers (indexes) into dense vectors of fixed size)
- The LSTM class from the keras.layers API (to apply a long short-term memory layer to an input)
- The Dropout class from the keras.layers API (to apply dropout to an input .
- The Dense class from the keras.layers API (to apply a regular densely-connected NN layer to an input)

In [13]:

From keras.models import Sequential

From keras.layers import Embedding, LSTM, Dropout, Dense

In [14]:

Importing the "imageio.v3" library (for reading and writing images)

See https://imageio.readthedocs.io/en/stable/

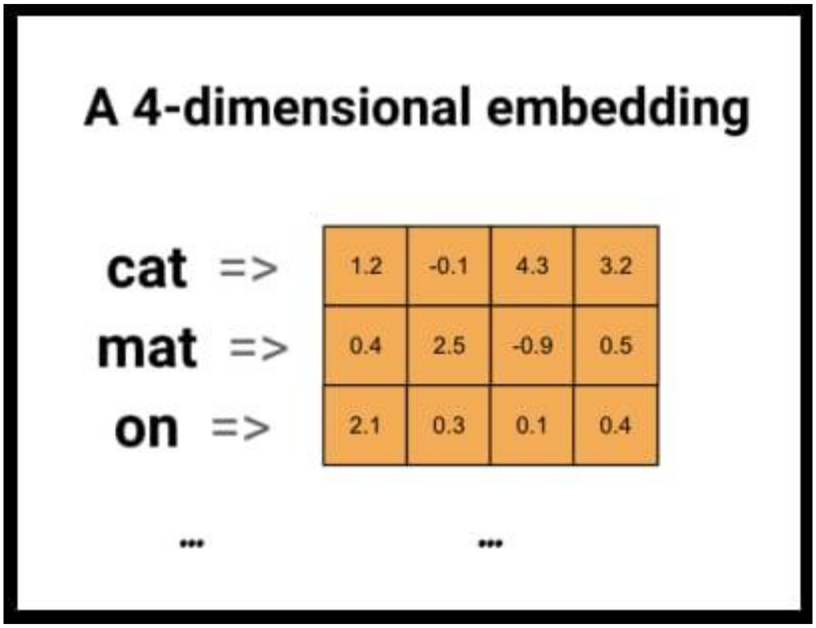
Import imageio.v3 as iio

Image = iio.imread(https://www.tensorflow.org/text/guide/images/embedding2.png)

Plt.figure(figsize = (7, 7))

Plt.imshow(image)

Plt.axis('off');



In the LSTM model, we set the following parameters:

- The output dimension of the Embedding layer (dimension of the vector space containing the word embeddings) is "output_dim"
- The number of units of the LSTM layer is "units_lstm"
- The dropout rate of the Dropout layer is "r"
- The activation function of the final Dense layer is sigmoid (this is a natural choice since the output of the model should be a number between 0, for negative reviews, and 1, for positive reviews)

In [15]:

Output_dim = 14

Units_lstm = 16

R = 0.8

Model = Sequential()

Model.add(Embedding(input_dim=dict_size, output_dim=output_dim, input_length=upper_bound))

Model.add(LSTM(units_lstm))

Model.add(Dropout®)

Model.add(Dense(1, activation='sigmoid'))

2022-07-04 08:40:29.473651: I tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool with default inter op setting: 2. Tune using inter_op_parallelism_threads for best performance.

We give a summary of the model using the summary method of the model class of Keras. The "None" value stands for the (not yet defined) value of the batch size.

In [16]:

Model.summary()

Model: "sequential"

Layer (type)	Output Shape	Param #
Embedding (Embedding)	(None, 291, 14)	490000
Lstm (LSTM)	(None, 16)	1984
Dropout (Dropout)	(None, 16)	0
Dense (Dense)	(None, 1)	17

Total params: 492,001

Trainable params: 492,001

Non-trainable params: 0

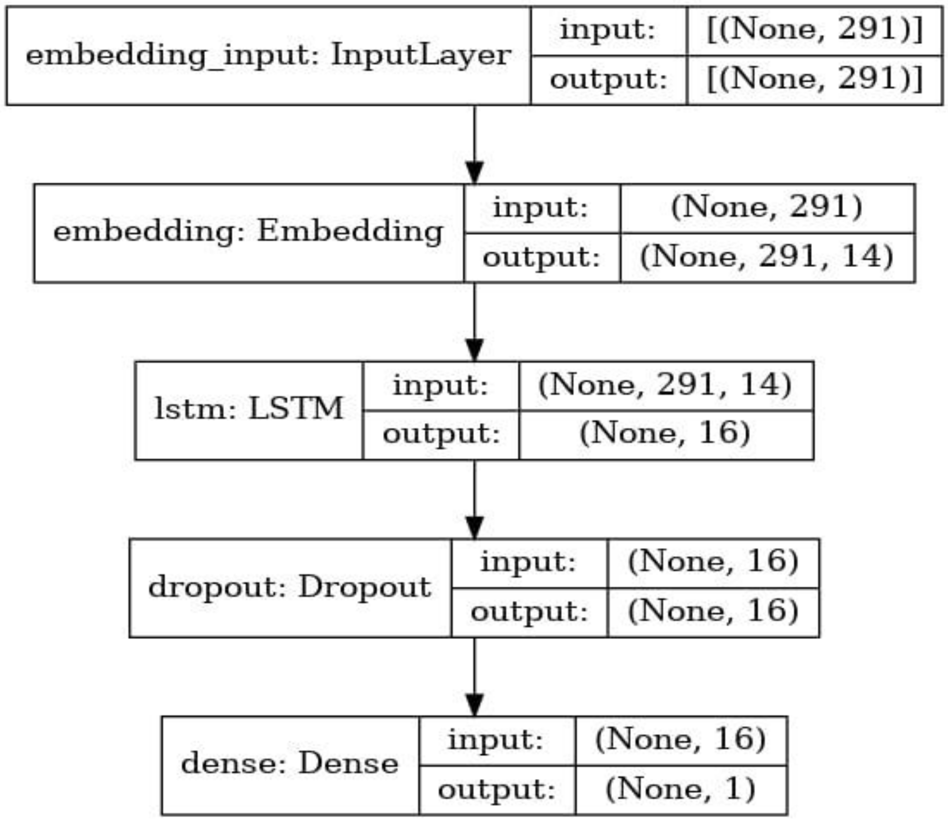
We import the plot_model function from the keras.utils.vis_utils module to plot a schema of the model.

In [17]:

From keras.utils.vis_utils import plot_model

Plot_model(model, show_shapes=True)

Out[17]:



We compile the model for training with the following parameters:

- Adam as optimizer to use during training process (a combination of gradient descent with momentum and RMSP)
- Binary cross-entropy (bce) between true labels and predicted labels as loss to minimise during training process
- Accuracy as metric to display during training process (how often predicted labels equal true labels)

In[18]:

Model.compile(optimizer=’adam’, loss=’bce’, metrics=’accuracy’)

We train the model with “train_rev_pad” as input array, “train_sent” as output array, validation split, batch size, number of epochs, and the option “shuffle=True” to shuffle the training data before each epoch. An epoch is a pass of the neural network over the entire training set and the batch size is the number of samples that are passed to the network at once. For each epoch, we thus have

Number of training steps

=

Length of training set – length of validation set

Batch size

.

In [19]:

Validation_split = 0.1

Batch_size = 384

Epochs = 3

Fitted = model.fit(train_rev_pad, train_sent, validation_split=validation_split,

Batch_size=batch_size, epochs=epochs, shuffle=True)

2022-07-04 08:40:31.429916: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)

Epoch 1/3

106/106 [==============================] – 32s 280ms/step – loss: 0.6357 – accuracy: 0.6617 – val_loss: 0.4595 – val_accuracy: 0.8480

Epoch 2/3

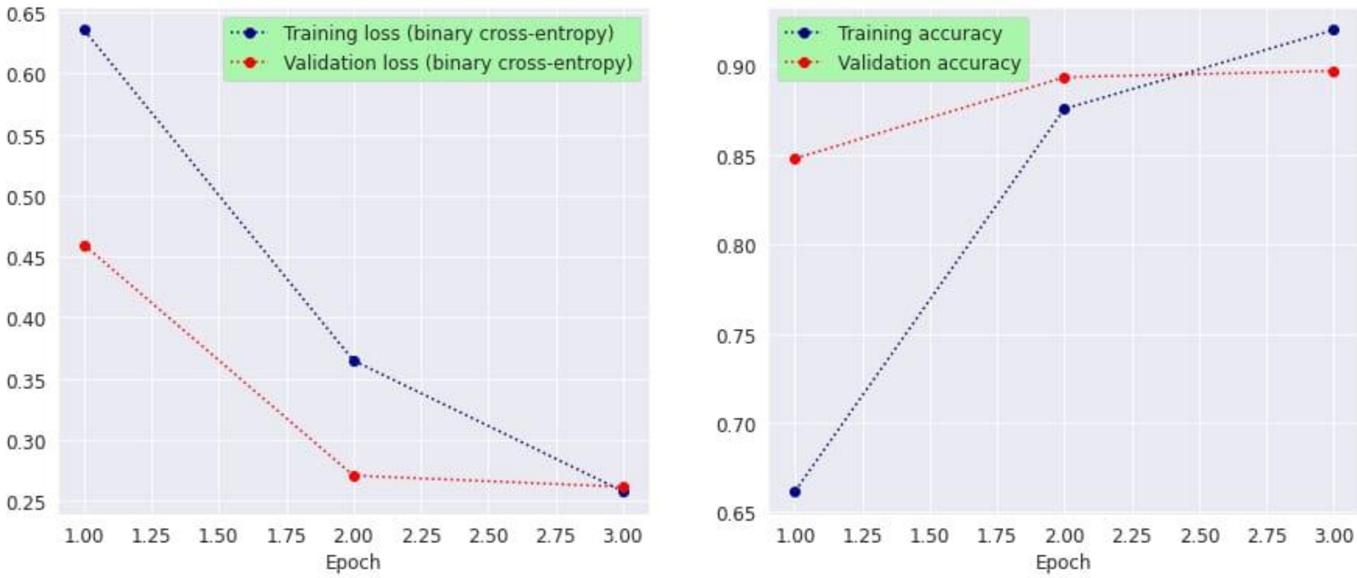
106/106 [==============================] – 29s 276ms/step – loss: 0.3650 – accuracy: 0.8758 – val_loss: 0.2707 – val_accuracy: 0.8936

Epoch 3/3

106/106 [==============================] – 29s 275ms/step – loss: 0.2579 – accuracy: 0.9200 – val_loss: 0.2615 – val_accuracy: 0.8971

In [20]:

```
# Storing in “ep_values” the values of the epochs

Ep_values = range(1, epochs+1)

# Plot of the training loss and validation loss (binary cross-entropy)

With sns.axes_style(“darkgrid”):

    _, (loss, acc) = plt.subplots(1, 2, figsize=(15, 6))
    Loss.plot(ep_values, fitted.history[‘loss’], color=’darkblue’, linestyle=’dotted’,
              Marker=’o’, label=’Training loss (binary cross-entropy)’)
    Loss.plot(ep_values, fitted.history[‘val_loss’], color=’r’, linestyle=’dotted’,
              Marker=’o’, label=’Validation loss (binary cross-entropy)’)
    Loss.set_xlabel(‘Epoch’, size=’large’)
    Loss.legend(fontsize=’large’, facecolor=’palegreen’)

    Acc.plot(ep_values, fitted.history[‘accuracy’], color=’darkblue’, linestyle=’dotted’,
             Marker=’o’, label=’Training accuracy’)
    Acc.plot(ep_values, fitted.history[‘val_accuracy’], color=’r’, linestyle=’dotted’,
             Marker=’o’, label=’Validation accuracy’)
    Acc.set_xlabel(‘Epoch’, size=’large’)
    Acc.legend(fontsize=’large’, facecolor=’palegreen’)

    Plt.show()
```



Results:

First, we evaluate the loss and accuracy of the trained model on the test set.

In [21]:

Result= model.evaluate(test_rev_pad, test_sent)

157/157 [==============================] – 2s 14ms/step – loss: 0.2522 – accuracy: 0.9028

Next, we use the confusion_matrix() function from the sklearn.metrics module to compute the confusion matrix for the predictions of the trained model, and we use the heatmap method from seaborn to plot the confusion matrix.

In [22]:

From sklearn.metrics import confusion_matrix

Predictions = np.round(model.predict(test_rev_pad))

Cf_matrix = confusion_matrix(test_sent, predictions)

In [23]:

Storing in “legends” the legends of each entry of the confusion matrix

Storing in “percentages” the percentages of each entry of the confusion matrix

Storing in “labels” the grouped values (legend + percentage) of each entry of the confusion matrix

Legends = [‘True negatives’, ‘False positives’, ‘False negatives’, ‘True positives’]

Percentages = [round(100*num, 2) for num in cf_matrix.flatten()/np.sum(cf_matrix)]

Labels = [f’{v1}\n\n{v2}%’ for v1, v2 in zip(legends, percentages)]

Labels = np.asarray(labels).reshape(2, 2)

Heatmap plot of the confusion matrix

Plt.figure(figsize = (7, 7))

Cm = sns.heatmap(cf_matrix, annot=labels, fmt=’’, cmap=’vlag’, annot_kws={‘fontsize’: ‘large’})

cm.set_xlabel(‘Predicted sentiments’, size=’large’)

cm.set_ylabel(‘Actual sentiments’, size=’large’)

cm.xaxis.set_ticklabels([‘Negative’, ‘Positive’])

cm.yaxis.set_ticklabels([‘Negative’, ‘Positive’]) plt show()

Finally, we test the trained model on a randomly chosen review from the test set. We display the

original review, the sentiment predicted by the model with its probability, and the actual (correct) sentiment.

In[24]:

Storing in DataFrame “df_original” the original reviews and sentiments

Df_original = pd.read_csv(‘../input/imdb-dataset-of-50k-movie-reviews/IMDB Dataset.csv’)

Choosing randomly a review and its sentiment in the test data

Idx_test = random.randint(0, len(test_sent)-1)

Idx_original = test_rev.index[idx_test]

(actual_rev, actual_sent) = df_original.iloc[idx_original]

Storing in “prediction_sent” the predicted sentiment of the chosen review

Storing in “probability” the probability of the predicted sentiment of the chosen review

Prediction = model.predict(test_rev_pad)[idx_test][0]

Prediction_sent = ‘positive’ if prediction >= 0.5 else ‘negative’

Probability = round(prediction if prediction >= 0.5 else 1-prediction, 2)

Printing the original review, its predicted sentiment and probability, and original sentiment

Print(‘\033[1m’ + ‘Review #%d:’ % idx_original + ‘\033[0m’ + ‘\n’, actual_rev, ‘\n’)

Print(‘\033[1m’ + ‘Predicted sentiment:’ + ‘\033[0m’, prediction_sent, ‘(with probability %.2f)’ % probability, ‘\n’)

Print(‘\033[1m’ + ‘Actual sentiment:’ + ‘\033[0m’, actual_sent)

Review #24912:

1) Bad acting.

2) For a bunch of castaways on an alien planet, it sure looked like home, especially with the houses and roads you can glimpse in the background.

3) Terrible plot with stupid caracters making idiotic decisions and blithely losing precious survival equipment and clothing left, right and center.

4) Cool 70’s scifi jumpsuits (possibly the only good thing about this movie)

5) Interesting ship at the beginning (this crew must have been watching Space 1999 a lot). Too bad it blows up so early. The escape ship also got sunk too fast. *sigh*

6) Anthropologists might find some aspects of the movie interesting in terms of primate group behavior.

Predicted sentiment: negative (with probability 0.98)

Actual sentiment: negative

Conclusion:

By this document we completed how a preprocessing methods, NLP techniques are implemented in sentiment analysis of marketing using IMDb movie reviews.