**Text Prediction using Natural Language Processing**

**Executive Summary**

Natural language processing (NLP) is a rapidly growing field of study with an expanding number of applications. One task in NLP is text prediction. In a text prediction task, a sentence, or series of words, is presented to a prediction model. The model’s task is to then present the word or words with the highest probability of following the initial series to the user. For this project, a corpus of 3 documents is evaluated and used to train a text prediction model. In this report, summary and exploratory corpus statistics are presented along with a high-level road map to model development.

**Summary Statistics**

**Document Properties**

The training corpus provided for this project is comprised of 3 text documents. The documents are from three different sources to vary sentence structure and capture a wide range of uses of the English language. The documents are taken one each from the following sources: blogs (899288 lines of text), news websites (1010242 lines of text), and twitter (2360148 lines of text).

| **Document** | **Number of Lines** | **Number of Unique Words** |
| --- | --- | --- |
| blogs | 899288 | 832520 |
| news | 1010242 | 661750 |
| twitter | 2360148 | 1000223 |

We can see that the twitter document has the most lines and also the most unique words in the document. The line number is likely due to the 140 character cap for tweets. The high number of unique words could be due to misspellings and abbreviations which are used to fit messages into this 140 character cap. These words will likely be uncommon across the entire document or the entire corpus and will need to be dealt with before model training is started.

**Word Frequency Distribution**

A picture containing graphical user interface

Description automatically generatedThe plot above shows that 95% of the word instances in these three documents comes from less than 10% of the unique words that are observed in each of the three documents. This will have an important impact on the model development phase. Below, some of the most common words in the corpus are presented.

Chart, histogram

Description automatically generated

The plot above lends some key insights into the distribution of words in the corpus. Some words appear more frequently in some of the documents than they do in others. This information may be useful during model tuning to reweight an ngram’s probability.

**Model Development Plan**

A text prediction model will be built using the documents described above. The development process will consist of the following phases.

**Text Preprocessing**

As the above data suggests, the entire dataset need not be used in order to capture most word uses in this corpus. Preprocessing of the text will include: \* Stem of all words to reduce the number of forms each word appears in the corpus \* Add part-of-speech tags to all words \* Replace infrequenty words with an infrequent word tag () that includes part-of-speech

**Model Building**

Lists of n-grams can be built from the cleaned data. Unigrams up to four-grams will be created initially. The necessity of four-grams vs. trigrams will be evaluated in the model training process. Backoff logic will be evaluated as a means of handling previously unseen n-grams. The model will use smoothing to handle previously unseen words. Different smoothing methods will be evaluated during training. To train the model, Markov chains will be used to evaluate the probabilities of sentences. Probabilities will be stored as log Probability tables for the each of the different n-grams to avoid underflow.

**Application Development**

To allow users to interact with the model, a simple text prediction application will be built on top of model using Shiny. Users will submit a word or series of words to server which will process the input and return the 3 most likely next words.

* [**Start Here**](javascript:void(0);)
  + [Machine Learning](https://courses.analyticsvidhya.com/courses/Machine-Learning-Certification-Course-for-Beginners?utm_source=blog_navbar&utm_medium=start_here_button)
  + [Deep Learning](https://courses.analyticsvidhya.com/courses/getting-started-with-neural-networks?utm_source=blog_navbar&utm_medium=start_here_button)
  + [NLP](https://courses.analyticsvidhya.com/courses/Intro-to-NLP?utm_source=blog_navbar&utm_medium=start_here_button)
* [Articles](javascript:void(0);)
  + [Guides](https://www.analyticsvidhya.com/blog/category/guide/?utm_source=blog_navbar&utm_medium=machine_learning_button)
  + [Machine Learning](https://www.analyticsvidhya.com/blog/category/machine-learning/?utm_source=blog_navbar&utm_medium=machine_learning_button)
  + [Deep Learning](https://www.analyticsvidhya.com/blog/category/deep-learning/?utm_source=blog_navbar&utm_medium=deep_learning_button)
  + [NLP](https://www.analyticsvidhya.com/blog/category/nlp/?utm_source=blog_navbar&utm_medium=_button)
  + [Computer Vision](https://www.analyticsvidhya.com/blog/category/computer-vision/?utm_source=blog_navbar&utm_medium=article_button)
  + [Data Visualization](https://www.analyticsvidhya.com/blog/category/data-visualization/?utm_source=blog_navbar&utm_medium=_button)
  + [Careers](https://www.analyticsvidhya.com/blog/category/career/?utm_source=blog_navbar&utm_medium=career_button)
  + [More](javascript:void(0);)

[Write an Article](https://datahack.analyticsvidhya.com/contest/data-science-blogathon-19/?utm_source=blog_navbar&utm_medium=blogathon_button)

* [Courses](javascript:void(0);)
  + [Certified AI & ML BlackBelt Plus](https://blackbelt.analyticsvidhya.com/plus?utm_source=blog_navbar&utm_medium=blackbelt_button)
  + [Ascend Pro](https://ascendpro.analyticsvidhya.com/?utm_source=blog_navbar&utm_medium=ascend_pro_button)
  + [Data Science Immersive Bootcamp](https://bootcamp.analyticsvidhya.com/?utm_source=blog_navbar&utm_medium=bootcamp_button)

[All Courses](https://courses.analyticsvidhya.com/collections?utm_source=blog_navbar&utm_medium=all_courses_button)

* [Blogathon](https://datahack.analyticsvidhya.com/contest/data-science-blogathon-19/?utm_source=blog_navbar&utm_medium=blogathon_button)
* [Sign in](https://id.analyticsvidhya.com/auth/login/?next=https://www.analyticsvidhya.com/blog/2021/11/a-guide-to-automated-deep-machine-learning-for-natural-language-processing-text-prediction/)
* [Join Now](https://id.analyticsvidhya.com/auth/signup/?next=https://www.analyticsvidhya.com/blog/2021/11/a-guide-to-automated-deep-machine-learning-for-natural-language-processing-text-prediction/)

SAVE 30,000 INR on our JOB GUARANTEE PROGRAM | EARLY BIRD OFFER LIVE

[BOOK YOUR SEAT!](https://bootcamp.analyticsvidhya.com/?utm_source=blog_India&utm_medium=flashstrip#FeeStructure)

×

1. [Home](https://www.analyticsvidhya.com/blog/)

# A Guide to Automated Deep/Machine Learning for Natural Language Processing: Text Prediction

[**rendyk**](https://www.analyticsvidhya.com/blog/author/rendyk/)**— November 12, 2021**

[Advanced](https://www.analyticsvidhya.com/blog/category/advanced/) [Deep Learning](https://www.analyticsvidhya.com/blog/category/deep-learning/) [Guide](https://www.analyticsvidhya.com/blog/category/guide/) [Machine Learning](https://www.analyticsvidhya.com/blog/category/machine-learning/) [NLP](https://www.analyticsvidhya.com/blog/category/nlp/) [Programming](https://www.analyticsvidhya.com/blog/category/programming/) [Python](https://www.analyticsvidhya.com/blog/category/python-2/)

This article was published as a part of the [*Data Science Blogathon*](https://datahack.analyticsvidhya.com/contest/data-science-blogathon-14/)

This article starts by discussing the fundamentals of Natural Language Processing (NLP) and later demonstrates using Automated Machine Learning (AutoML) to build models to predict the sentiment of text data. Other applications of NLP are for translation, speech recognition, chatbot, *etc.* You may be thinking that this article is general because there are many NLP tutorials and sentiment analyses on the internet. But, this article tries to show something different. It will demonstrate the use of AutoKeras as an AutoML to generate Deep Learning to predict text, especially sentiment rating and emotion. But before that, let’s briefly discuss basic NLP because it supports text sentiment prediction.

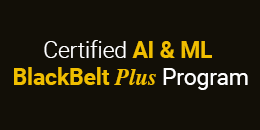
This article will cover the following topics:

1. Regular expression
2. Word tokenization
3. Named Entity Recognition
4. Stemming and lemmatization
5. Word cloud
6. Bag-of-words (BoW)
7. Term Frequency — Inverse Document Frequency (TF-IDF)
8. Sentiment analysis
9. Text Regression (Automated Machine Learning and Deep Learning)
10. Text Classification (Automated Deep Learning)

NLP aims to make the sense of text data. The examples of text data commonly analyzed in Data Science are reviews of products, posts from social media, documents, *etc.* Unlike numerical data, text data cannot be analyzed with descriptive statistics. If we have a list of product prices data containing 1000 numbers, we can understand the overall prices data by examining the average, median, standard deviation, boxplot, and other technics. We do not have to read all the numbers to understand them.

Now, if we have thousands of texts reviewing products from an e-commerce online store, how do we know what the reviews are saying without reading them all. With NLP, those text reviews can be interpreted into satisfaction rating, emotion, *etc.* This is called sentiment analysis. Machine Learning models are created to predict the sentiment of the text.

Before getting into the sentiment analysis, this article will start from the very basics of NLP, such as regular expression, word tokenization, until Bag-of-Words and how they contribute to the sentiment analysis. [Here](https://www.kaggle.com/rendyk/e-commerce-nlp-and-sentiment-analysis-part-1) is the Python notebook supporting this article.



##### Become a Full-Stack Data Scientist

Power Ahead in your AI ML Career | No Pre-requisites Required[Download Brochure](https://blackbelt.analyticsvidhya.com/plus?utm_source=blog&utm_medium=Bannerbetweenarticles&utm_campaign=BlackBelt)

## Regular Expression

Regular Expression (RegEx) is a pattern to match, search, find, or split one or more sentences or words. The following code is an example of using 6 RegEx lines to match the same sentence. The RegEx w+, d+, s, [a-z]+, [A-Z]+, (w+|d+) has the code to search for word, digit, space, small-cap alphabet, big-cap alphabet, and digit respectively. The re.match will return the respective text with a certain pattern for the first word of the text. In this case, it is the word “The”.

import re

text = 'The monkeys are eating 7 bananas on the tree!'

print(re.match('w+', text)) # word

print(re.match('d+', text)) # digit

print(re.match('s', text)) # space

print(re.match('[a-z]+', text)) # alphabet

print(re.match('[A-Z]+', text)) # big caps alphabet

print(re.match('(w+|d+)', text)) # word or digit

**Output**:

None

None

None

Notice that the second, third, and fourth returns “None” as the word “The” does not start with a digit, space, and small alphabet.

The RegEx re.search searches the first text according to the pattern. Unlike re.match which only checks the first word in the text, re.search can identify the words after the first word in the text.

print(re.search('w+', text)) # word

print(re.search('d+', text)) # digit

print(re.search('s', text)) # space

print(re.search('[a-z]+', text)) # small caps alphabet

print(re.search('[a-z]', text)) # small caps alphabet

print(re.search('[A-Z]+', text)) # big caps alphabet

print(re.search('(w+|d+)', text)) # word or digit

Output

<re.Match object; span=(0, 3), match='The'>

<re.Match object; span=(23, 24), match='7'>

<re.Match object; span=(3, 4), match=' '>

<re.Match object; span=(1, 3), match='he'>

<re.Match object; span=(1, 2), match='h'>

<re.Match object; span=(0, 1), match='T'>

<re.Match object; span=(0, 3), match='The'>

Instead of only matching the beginning of the text, re.findall finds all of the RegEx pattern in the text and store them in a list.

print(re.findall('w+', text)) # word

print(re.findall('d+', text)) # digit

print(re.findall('s', text)) # space

print(re.findall('[a-z]+', text)) # small caps alphabet

print(re.findall('[a-z]', text)) # small caps alphabet

print(re.findall('[A-Z]+', text)) # big caps alphabet

print(re.findall('(w+|d+)', text)) # word or digit

**Output**:

['The', 'monkeys', 'are', 'eating', '7', 'bananas', 'on', 'the', 'tree']

['7']

[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

['he', 'monkeys', 'are', 'eating', 'bananas', 'on', 'the', 'tree']

['h', 'e', 'm', 'o', 'n', 'k', 'e', 'y', 's', 'a', 'r', 'e', 'e', 'a', 't', 'i', 'n', 'g', 'b', 'a', 'n', 'a', 'n', 'a', 's', 'o', 'n', 't', 'h', 'e', 't', 'r', 'e', 'e']

['T']

['The', 'monkeys', 'are', 'eating', '7', 'bananas', 'on', 'the', 'tree']

re.split searches for the RegEx pattern in the whole text and splits the text into list of strings based on the RegEx pattern.

print(re.split('w+', text)) # word

print(re.split('d+', text)) # digit

print(re.split('s', text)) # space

print(re.split('[a-z]+', text)) # small caps alphabet

print(re.split('[A-Z]+', text)) # big caps alphabet

print(re.split('(w+|d+)', text)) # word or digit

**Output**:

['', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '!']

['The monkeys are eating ', ' bananas on the tree!']

['The', 'monkeys', 'are', 'eating', '7', 'bananas', 'on', 'the', 'tree!']

['T', ' ', ' ', ' ', ' 7 ', ' ', ' ', ' ', '!']

['', 'he monkeys are eating 7 bananas on the tree!']

['', 'The', ' ', 'monkeys', ' ', 'are', ' ', 'eating', ' ', '7', ' ', 'bananas', ' ', 'on', ' ', 'the', ' ', 'tree', '!']

## Word Tokenization

There are still many RegEx patterns to explore, but this article will continue to word tokenization. Word tokenization splits sentences into words. The below code shows how it is done. Can you tell which RegEx code above can do the same thing?

import nltk

from nltk.tokenize import word\_tokenize, sent\_tokenize

word\_tokenize(text)

**Output**:

['The', 'monkeys', 'are', 'eating', '7', 'bananas', 'on', 'the', 'tree', '!']

Besides word tokenization, we can also perform sentence tokenization.

text2 = 'The monkeys are eating 7 bananas on the tree! The tree will only have 5 bananas left later. One monkey is jumping to another tree.'

sent\_tokenize(text2)

Output:

['The monkeys are eating 7 bananas on the tree!',

'The tree will only have 5 bananas left later.',

'One monkey is jumping to another tree.']

## Named Entity Recognition (NER)

After the word tokenization, we can apply NER to it. NER identifies which entity each word is. Please find the example below of how pos\_tag from nltk is used to perform NER.

text\_tag = word\_tokenize(text2)

nltk.pos\_tag(text\_tag)

**Output:**

[('The', 'DT'),

('monkeys', 'NNS'),

('are', 'VBP'),

('eating', 'VBG'),

('7', 'CD'),

('bananas', 'NNS'),

('on', 'IN'),

('the', 'DT'),

('tree', 'NN'),

('!', '.'),

('The', 'DT'),

('tree', 'NN'),

('will', 'MD'),

('only', 'RB'),

('have', 'VB'),

('5', 'CD'),

('bananas', 'NNS'),

('left', 'VBD'),

('later', 'RB'),

('.', '.'),

('One', 'CD'),

('monkey', 'NN'),

('is', 'VBZ'),

('jumping', 'VBG'),

('to', 'TO'),

('another', 'DT'),

('tree', 'NN'),

('.', '.')]

Where, DT=determiner, NNS=plural noun, VBP= verb for non-3rd person singular present, VBG= gerund/present participle verb, CD= cardinal digit, IN= preposition/subordinating conjunction, NN=noun, MD=modal verb, RB=adverb, VB=base form verb, and so on.

Another way to perform NER is by using spacy package. Observe the following example of how spacy identifies each word lemmatization, PoS, tag, dep, shape, whether it is an alphabet, and whether it is a stop word. We will discuss lemmatization later. PoS and tag define the part of speech, such as a determiner, noun, auxiliary verb, number, etc. “dep” shows the word dependencies. “shape” shows the word letters in X and x for big capital and small capital letter respectively. “is\_alphabet” and “is\_stop\_words” identify whether the word is an alphabet or stop word respectively.

import spacy

spa = spacy.load("en\_core\_web\_sm")

spa\_text = spa(text)

print('text' + 't' + 'lemmatized' + 't' + 'PoS' + 't' + 'tag' + 't' + 'dep' + 't' +

'shape' + 't' + 'is\_alphabet' + 't' + 'is\_stop\_words')

for word in spa\_text:

print(word.text + 't' + word.lemma\_ + 'tt' + word.pos\_ + 't' + word.tag\_ + 't' + word.dep\_ + 't' + word.shape\_ + 't' + str(word.is\_alpha) + 'tt' + str(word.is\_stop))

Output:

text lemmatized PoS tag dep shape is\_alphabet is\_stop\_words

The the DET DT det Xxx True True

monkeys monkey NOUN NNS nsubj xxxx True False

are be AUX VBP aux xxx True True

eating eat VERB VBG ROOT xxxx True False

7 7 NUM CD nummod d False False

bananas banana NOUN NNS dobj xxxx True False

on on ADP IN prep xx True True

the the DET DT det xxx True True

tree tree NOUN NN pobj xxxx True False

! ! PUNCT . punct ! False False

## Stemming and Lemmatization

Stemming and Lemmatization return a word to its simpler root form. Both stemming and lemmatization are similar to each other. To understand the difference, observe the following code. Here, we apply stemming and lemmatization to the word “studies” and they will return different outputs. Stemming returns “studi” as the root form of “studies”. Lemmatization returns “study” as the root form of “studies”. The root form returned by lemmatization has a meaning. The root form of stemming sometimes does not have a meaning. The word “studi” from stemming does not have a meaning. Stemming cannot change the letter “i” from the word “studies”.

# Stemming

from nltk.stem import PorterStemmer

print(PorterStemmer().stem('studies'))

# Lemmatization

from nltk.stem import WordNetLemmatizer

print(WordNetLemmatizer().lemmatize('studies'))

Output:

studi

study

## Word Cloud

For this exercise, we are going to use [women’s clothing reviews](https://www.kaggle.com/nicapotato/womens-ecommerce-clothing-reviews) from an e-commerce dataset. The dataset provides the text reviews and the rating score from 1 to 5. We are now trying to understand what the 23,486 reviews were saying. If the reviews are in numbers, we can use descriptive statistics to see the data distribution. But, the reviews are in text form. How to quickly get a summary of the 23,486 reviews text? Reading them one by one is not an efficient solution.

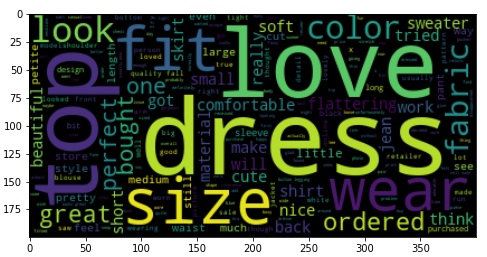
A simple way is to plot the word cloud. Word cloud displays the commonly found words from the whole dataset. A larger font size means more frequently found.

visual\_rev = WordCloud().generate(' '.join(data['Review Text']))

plt.figure(figsize=(8,8))

plt.imshow(visual\_rev, interpolation='bilinear')

plt.show()

Fig. 1 word cloud (source: [image by author](https://www.kaggle.com/rendyk/e-commerce-nlp-and-sentiment-analysis-part-1))

From the word cloud, we can notice that the reviews are talking about dress, love, size, top, wear, and so on as they are the most commonly found words. To display the exact frequency number of each word, we can use Counter(). Let’s demonstrate it using the variable “text2”.

from collections import Counter

Counter(word\_tokenize(text2))

**Output**:

Counter({'The': 2,

'monkeys': 1,

'are': 1,

'eating': 1,

'7': 1,

'bananas': 2,

'on': 1,

'the': 1,

'tree': 3,

'!': 1,

'will': 1,

'only': 1,

'have': 1,

'5': 1,

'left': 1,

'later': 1,

'.': 2,

'One': 1,

'monkey': 1,

'is': 1,

'jumping': 1,

'to': 1,

'another': 1})

The following code does the same thing, but it calls only the top 3 most common words.

Counter(word\_tokenize(text2)).most\_common(3)

Output:

[('tree', 3), ('The', 2), ('bananas', 2)]

## Bag-of-Words (BoW)

Bag-of-Words does a similar thing. It returns a table with features consisting of the words in the reviews. The row contains the word frequency. The following code applies BoW to the women’s clothing review dataset. It will create a data frame with tokenized words as the features.

In the CountVectoricer, I set max\_features to be 100 to limit the number of features. “max\_df” and “min\_df” determine the maximum and minimum appearance percentage of the tokenized words in the documents. The selected tokenized words should appear in more than 10% and less than 95% of the documents. This is to deselect words that appear too rarely and too frequently. “ngram\_range” of (1,2) is set to tokenize 1 word and 2 consecutive words (2-word sequence or bi-gram). This is important to detect two-word sequences, like “black mamba”, “land cover”, “not happy”, “not good”, *etc*. Notice that lemmatizer and regular expression are also used.

*# Filter rows with column*

data = dataset.loc[dataset['Review Text'].notnull(),:]

*# Apply uni- and bigram vectorizer*

class **lemmatizer**(object):

def \_\_init\_\_(self):

self.wnl = WordNetLemmatizer()

def \_\_call\_\_(self, df):

return [self.wnl.lemmatize(word) for word **in** word\_tokenize(df)]

vectorizer = CountVectorizer(max\_features=100, max\_df=0.95, min\_df=0.1, ngram\_range=(1,2),

tokenizer=lemmatizer(), lowercase=True, stop\_words='english',

token\_pattern = r'w+')

vectorizer.fit(data['Review Text'])

count\_vector = vectorizer.transform(data['Review Text'])

*# Transform into data frame*

bow = count\_vector.toarray()

bow = pd.DataFrame(bow, columns=vectorizer.get\_feature\_names())

bow.head()

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ! | beautiful | … | ordered | perfect | really | run | size | small | soft | wa | wear | work |
| 0 | 0 | 0 | … | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | … | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | … | 1 | 0 | 1 | 0 | 1 | 3 | 0 | 3 | 0 | 1 |
| 3 | 2 | 0 | … | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 4 | 3 | 0 | … | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| … | … | … | … | … | … | … | … | … | … | … | … | … | … |

### Term Frequency — Inverse Document Frequency (TF-IDF)

Similar to BoW, TF-IDF also creates a data frame with features of tokenized words. But, it tries to scale up the rare terms and scale down the frequent terms. This is useful, for example, the word “the” may appear many times, but it is not what we expect as it does not have a sentiment tendency. The values of TF-IDF are generated in 3 steps. Step 1, calculate the TF = count of term/number of words. For example, let’s apply these 3 sentences:

1. The monkeys are small. The ducks are also small.

2. The comedians are hungry. The comedians then go to eat.

3. The comedians have small monkeys.

|  |  |  |  |
| --- | --- | --- | --- |
| **Term** | **TF 1** | **TF 2** | **TF 3** |
| the | 2/9 | 2/10 | 1/5 |
| monkeys | 1/9 | 0/10 | 1/5 |
| are | 2/9 | 1/10 | 0/5 |
| small | 2/9 | 0/10 | 1/5 |
| ducks | 1/9 | 0/10 | 0/5 |
| comedians | 0/9 | 2/10 | 1/5 |
| hungry | 0/9 | 1/10 | 0/5 |
| … | … | … | … |

The first text contains 9 words. It has 2 words of “the”. So, the word “the” in TF 1 is 2/9.

Step 2, IDF = log (number of documents/number of documents with the term)

|  |  |
| --- | --- |
| **Term** | **IDF** |
| the | log(3/3) |
| monkeys | log(3/2) |
| are | log(3/2) |
| small | log(3/2) |
| ducks | log(3/1) |
| comedians | log(3/2) |
| hungry | log(3/1) |
| … | … |

The word “monkeys” appears 2 times in 3 documents. So, the IDF is log(3/2).

Step 3, calculate the TF-IDF = TF \* IDF

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Term** | **TF 1** | **TF 2** | **TF 3** | **IDF** | **TF-IDF 1** | **TF-IDF 2** | **TF-IDF 3** |
| the | 2/9 | 2/10 | 1/5 | log(3/3) | 0.000 | 0.000 | 0.000 |
| monkeys | 1/9 | 0/10 | 1/5 | log(3/2) | 0.020 | 0.000 | 0.035 |
| are | 2/9 | 1/10 | 0/5 | log(3/2) | 0.039 | 0.018 | 0.000 |
| small | 2/9 | 0/10 | 1/5 | log(3/2) | 0.039 | 0.000 | 0.035 |
| ducks | 1/9 | 0/10 | 0/5 | log(3/1) | 0.053 | 0.000 | 0.000 |
| comedians | 0/9 | 2/10 | 1/5 | log(3/2) | 0.000 | 0.035 | 0.035 |
| hungry | 0/9 | 1/10 | 0/5 | log(3/1) | 0.000 | 0.048 | 0.000 |
| … | … | … | … | … | … | … | … |

Here is the output table.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Term** | **the** | **monkeys** | **are** | **small** | **ducks** | **comedians** | **hungry** | **…** |
| 1 | 0.000 | 0.020 | 0.039 | 0.039 | 0.053 | 0.000 | 0.000 | … |
| 2 | 0.000 | 0.000 | 0.018 | 0.000 | 0.000 | 0.035 | 0.048 | … |
| 3 | 0.000 | 0.035 | 0.000 | 0.035 | 0.000 | 0.035 | 0.000 | … |

Now, let’s compare it with the BoW data frame below. The word “the” in BoW data frame has the values of 2, 2, and 1 for text 1, 2, and 3 respectively. On the other hand, the same word has 0 value for all of the 3 texts in the TF-IDF data frame. This is because the word “The” appears in all 3 texts. Hence, the IDF is zero (log(3/3)). The words “monkeys” and “ducks” appear once in the first text, but “ducks” has a higher value (0.053) compared to the “monkeys” value (0.020) in the TF-IDF data frame. The word “ducks” appears less in all documents than “monkeys” does, so it gives more highlight to the word “duck”.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Text** | **the** | **monkeys** | **are** | **small** | **ducks** | **comedians** | **hungry** | **…** |
| **1** | 2 | 1 | 2 | 2 | 1 | 0 | 0 | … |
| **2** | 2 | 0 | 1 | 0 | 0 | 2 | 1 | … |
| **3** | 1 | 1 | 0 | 1 | 0 | 1 | 0 | … |

Here is the code to apply TF-IDF to the women’s clothing dataset.

from sklearn.feature\_extraction.text import TfidfVectorizer

tfidf = TfidfVectorizer(max\_features=100)

tfidf.fit(data['Review Text'])

tfidf\_data = tfidf.transform(data['Review Text'])

tfidf\_data = pd.DataFrame(tfidf\_data.toarray(), columns=tfidf.get\_feature\_names())

tfidf\_data.head()

Output:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | all | also | am | an | and | are | … | will | with | work | would | you |
| 0 | 0.000000 | 0.0 | 0.000000 | 0.0 | 0.602589 | 0.0 | … | 0.0 | 0.000000 | 0.000000 | 0.000000 | 0.0 |
| 1 | 0.000000 | 0.0 | 0.148418 | 0.0 | 0.133039 | 0.0 | … | 0.0 | 0.000000 | 0.000000 | 0.307109 | 0.0 |
| 2 | 0.000000 | 0.0 | 0.000000 | 0.0 | 0.163130 | 0.0 | … | 0.0 | 0.000000 | 0.163481 | 0.000000 | 0.0 |
| 3 | 0.000000 | 0.0 | 0.000000 | 0.0 | 0.142825 | 0.0 | … | 0.0 | 0.000000 | 0.000000 | 0.000000 | 0.0 |
| 4 | 0.228419 | 0.0 | 0.000000 | 0.0 | 0.082899 | 0.0 | … | 0.0 | 0.273455 | 0.000000 | 0.000000 | 0.0 |

What is the use of text converted into BoW or TF-IDF data frame? It is very important if we want to apply Machine Learning to text data. Machine Learning does not understand text, so the text must be converted into a numeric data frame. One of the common use of Machine Learning for text prediction is sentiment analysis. Sentiment analysis can predict the sentiment of the review text. Instead of reading the reviews one by one, sentiment analysis can convert the text into how satisfied the reviews sound.

## Sentiment Analysis

Sentiment analysis can be run by using TextBlob or training a Machine Learning model. TextBlob does not require training. It can tell the polarity and subjectivity of the reviews. The polarity ranges from 1 to -1 expressing positive sentiment to negative sentiment. Here is the code to apply sentiment analysis to the text2 = ‘The monkeys are eating 7 bananas on the tree! The tree will only have 5 bananas left later. One monkey is jumping to another tree.’

from textblob import TextBlob

TextBlob(text2).sentiment

Output:

Sentiment(polarity=-0.0125, subjectivity=0.25)

Now, let’s see how it works on the women’s clothing dataset.

# Applying text blob sentiment

def polarity(t):

a = TextBlob(t).sentiment

return a[0]

def subjectivity(t):

a = TextBlob(t).sentiment

return a[1]

data['polarity'] = data.apply(lambda t: polarity(t['Review Text']), axis=1)

data['subjectivity'] = data.apply(lambda t: subjectivity(t['Review Text']), axis=1)

data.head()

Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Review Text | Rating | polarity | subjectivity |
| 0 | Absolutely wonderful – silky and sexy and comf… | 4 | 0.633333 | 0.933333 |
| 1 | Love this dress! it’s sooo pretty. i happene… | 5 | 0.339583 | 0.725000 |
| 2 | I had such high hopes for this dress and reall… | 3 | 0.073675 | 0.356294 |
| 3 | I love, love, love this jumpsuit. it’s fun, fl… | 5 | 0.550000 | 0.625000 |
| 4 | This shirt is very flattering to all due to th… | 5 | 0.512891 | 0.568750 |

The table above displays the first 5 rows of the review text polarity. Examine how the words in “Review Text” return the “polarity”. In the same dataset, the satisfaction rating is also given by each of the reviewers in the feature “Rating”. The “Rating” ranges from 1 to 5. The figure below visualizes the polarity distribution of each rating class. Examine that a higher rating tends to have more positive polarity.

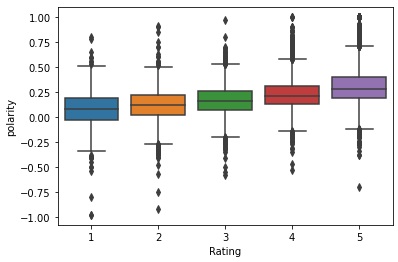


Fig. 2 Polarity distribution of each rating class (source: [image by author](https://www.kaggle.com/rendyk/e-commerce-nlp-and-sentiment-analysis-part-1))

## Text Regression (Automated Machine Learning and Deep Learning)

The sentiment analysis provided by TextBlob is a pre-trained model. Users can directly apply it to text. But, what if we want to train our new model to analyze text? What if we want to build a model to detect sentiment ranging from 1 to 5, instead of the text positivity and negativity rate (like using TextBlob)?

Using TextBlob has the advantage that users do not have to train a large number of data. On the other hand, creating a new Machine Learning model for text analysis requires extra time and resources. But, the created new model will be much more customized to a specific topic based on a training dataset. For example, training women’s clothing review dataset will result in an NLP model that is more specific in predicting the sentiment.

In Machine Learning for text analysis or NLP, there are text regression and text classification. Text regression aims to analyze text with continuous or ordinal output. Predicting the “rating” of women’s clothing reviews dataset requires text regression analysis because the output ranges from 1 to 5. The Text Regression notebook is available [here](https://www.kaggle.com/rendyk/e-commerce-nlp-and-sentiment-analysis-part-2).

Just like Machine Learning for structured data, Machine Learning for NLP also requires data frames containing engineered features and label. The engineered features for predictors can be generated using BoW or TF-IDF. The 2 data frames have been made before. According to how the data frames are generated, let’s call them BoW data frame and TF-IDF data frame.

To train a Machine Learning model for text analysis, the technic is the same as training it for structured data. We can use linear regression, decision tree, gradient boosting tree, *etc.* The code below uses AutoSklearn to train NLP Machine Learning with BoW data frame. AutoSklearn is an automated Machine Learning that can perform feature engineering, model selection, and hyperparameter-tuning automatically. Users can skip those processes and get a model in a specified time allocation. The AutoSklearn below is set to find an optimal model in 3 minutes. To find out more about Automated Machine Learning, visit my previous [article](https://www.analyticsvidhya.com/blog/2021/09/10-automated-machine-learning-for-supervised-learning-part-2/).

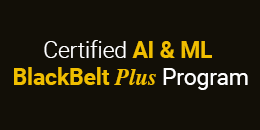
*# Create the model*

sklearn = AutoSklearnRegressor(time\_left\_for\_this\_task=3\*60, per\_run\_time\_limit=60, n\_jobs=-1)

*# Fit the training data*

sklearn.fit(X\_train, y\_train)

The output prediction of the text regression is a continuous number ranging from 1 to 5. The objective is to predict the review rating ranging from 1 to 5 in integer. So, the predicted values must be rounded to the closest integer between 1 to 5.



##### Become a Full-Stack Data Scientist

Power Ahead in your AI ML Career | No Pre-requisites Required[Download Brochure](https://blackbelt.analyticsvidhya.com/plus?utm_source=blog&utm_medium=Bannerbetweenarticles&utm_campaign=BlackBelt)

*# Sprint Statistics*

print(sklearn.sprint\_statistics())

*# Predict the test data*

pred\_sklearn = sklearn.predict(X\_test)

pred\_sklearn2 = [round(i) for i **in** pred\_sklearn]

After creating the model, the next step is to validate the model with the unseen dataset. The RMSE of the predicted unseen dataset is 0.9886. However, if we check the prediction outputs with the true values using the confusion matrix, we can find that the model cannot predict well in ratings 1, 2, and 3. Note that this does not necessarily mean that AutoSklearn is not good enough. The AutoSklearn was given only 3 minutes to create models automatically. The result RMSE can be better if it has more time.

*# Compute the RMSE*

rmse\_sklearn = mean\_squared\_error(y\_test, pred\_sklearn2)\*\*0.5

print('RMSE: ' + str(rmse\_sklearn))

Output:

auto-sklearn results:

Dataset name: 2fbec688-f37c-11eb-8112-0242ac130202

Metric: r2

Best validation score: 0.287286

Number of target algorithm runs: 27

Number of successful target algorithm runs: 9

Number of crashed target algorithm runs: 0

Number of target algorithms that exceeded the time limit: 9

Number of target algorithms that exceeded the memory limit: 9

RMSE: 0.9885634389230906

*# Prediction results*

print('Confusion Matrix')

print(pd.DataFrame(confusion\_matrix(y\_test, pred\_sklearn2), index=[1,2,3,4,5], columns=[1,2,3,4,5]))

Output:

Confusion Matrix

1 2 3 4 5

1 0 2 66 90 6

2 0 8 132 160 10

3 0 2 177 347 39

4 0 2 136 605 239

5 0 2 79 1170 1257

Let’s repeat the processes above, but this time by using the TF-IDF data frame as the input. AutoSklearn is also applied and the RMSE of the unseen dataset is 0.9770. The result is also similar to the previous one. The predictions are bad in predicting the rating of 1, 2, and 3.

*# Create the model*

sklearn\_idf = AutoSklearnRegressor(time\_left\_for\_this\_task=3\*60, per\_run\_time\_limit=60, n\_jobs=-1)

*# Fit the training data*

sklearn\_idf.fit(X\_train\_idf, y\_train)

*# Sprint Statistics*

print(sklearn\_idf.sprint\_statistics())

*# Predict the test data*

pred\_sklearn\_idf = sklearn\_idf.predict(X\_test\_idf)

pred\_sklearn\_idf2 = [round(i) for i **in** pred\_sklearn\_idf]

*# Compute the RMSE*

rmse\_sklearn\_idf = mean\_squared\_error(y\_test, pred\_sklearn\_idf2)\*\*0.5

print('RMSE: ' + str(rmse\_sklearn\_idf))

Output:

auto-sklearn results:

Dataset name: a573493e-f37f-11eb-8112-0242ac130202

Metric: r2

Best validation score: 0.285567

Number of target algorithm runs: 26

Number of successful target algorithm runs: 8

Number of crashed target algorithm runs: 0

Number of target algorithms that exceeded the time limit: 8

Number of target algorithms that exceeded the memory limit: 10

RMSE: 0.9769930120276676

*# Prediction results*

print('Confusion Matrix')

pred\_sklearn\_idf3 = [i if i <= 5 else 5 for i **in** pred\_sklearn\_idf2]

print(pd.DataFrame(confusion\_matrix(y\_test, pred\_sklearn\_idf3), index=[1,2,3,4,5], columns=[1,2,3,4,5]))

Output:

Confusion Matrix

1 2 3 4 5

1 0 3 75 82 4

2 0 5 147 151 7

3 0 6 178 343 38

4 0 1 124 617 240

5 0 0 84 1198 1226

To see which Machine Learning algorithms are created from the AutoSklearn, run the below code.

*# Show the models*

print(sklearn\_idf.show\_models())

Now, let’s use another autoML, the AutoKeras. AutoKeras automatically creates Deep Learning models. And yes, not only does it cover the hyperparameter-tuning, but also the Deep Learning layers architecture. After installing and importing the AutoKeras package, we can start the data preparation. The data are prepared in array format separating the training and test datasets. Notice that the feature contains only one column the “Review Text”. AutoKeras does not require users to process applying BoW or TF-IDF. The label is still the same, the “Rating”.

!pip install autokeras

import autokeras as ak

*# Preparing the data for autokeras*

X\_train\_ak = np.array(data.loc[X\_train.index, 'Review Text'])

y\_train\_ak = np.array(data.loc[X\_train.index, 'Rating'])

X\_test\_ak = np.array(data.loc[X\_test.index, 'Review Text'])

y\_test\_ak = np.array(data.loc[X\_test.index, 'Rating'])

Then, we create a TextRegressor with maximum trials of 3. The AutoKeras will create a maximum of 3 prediction models. The training data is split to have a validation dataset from the 20% of the total training dataset. Then, we can fit the data with 30 epochs for this demonstration.

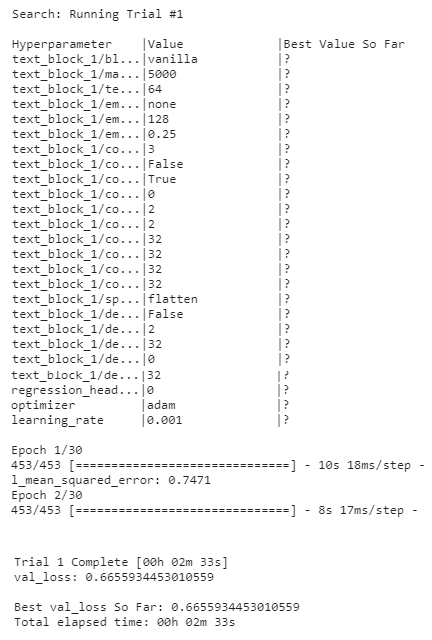
*# Create the model*

keras = ak.TextRegressor(overwrite=True, max\_trials=3)

*# Fit the training dataset*

keras.fit(X\_train\_ak, y\_train\_ak, epochs=30, validation\_split=0.2)

When the AutoKeras is running, it will show the following output cell. But, it will disappear when the process is done.

Fig. 3 running AutoKeras (source: image by author)

After the model creation is done, we can export and summarize it. Observe the layers automatically generated by the AutoKeras. The input layer is followed by expand\_last\_dim, text\_vectorization, embedding, dropout, conv1d, max\_pooling, flatten, and other layers.

*# Show the built models*

keras\_export = keras.export\_model()

keras\_export.summary()

**Output:**

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None,)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

expand\_last\_dim (ExpandLastD (None, 1) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

text\_vectorization (TextVect (None, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding (Embedding) (None, 128, 128) 640128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 128, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv1d (Conv1D) (None, 126, 32) 12320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv1d\_1 (Conv1D) (None, 124, 32) 3104

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling1d (MaxPooling1D) (None, 62, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv1d\_2 (Conv1D) (None, 60, 64) 6208

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv1d\_3 (Conv1D) (None, 58, 32) 6176

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling1d\_1 (MaxPooling1 (None, 29, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten (Flatten) (None, 928) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 32) 29728

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

re\_lu (ReLU) (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 32) 1056

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

re\_lu\_1 (ReLU) (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

regression\_head\_1 (Dense) (None, 1) 33

=================================================================

Total params: 698,753

Trainable params: 698,753

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Now, let’s see and compare the RMSE of the AutoKeras model. The RMSE is 0.8389. Observing the confusion matrix, the AutoKeras can predict all of the rating levels better. It still cannot predict rating 1 well enough though.

*# Predict the test data*

pred\_keras = keras.predict(X\_test\_ak)

pred\_keras = list(chain(\*pred\_keras))

pred\_keras2 = [i if i <= 5 else 5 for i **in** pred\_keras]

pred\_keras2 = [i if i >= 1 else 1 for i **in** pred\_keras2]

pred\_keras2 = [round(i) for i **in** pred\_keras2]

*# Compute the RMSE*

rmse\_keras = mean\_squared\_error(y\_test\_ak, pred\_keras2)\*\*0.5

print('RMSE: ' + str(rmse\_keras))

**Output**:

142/142 [==============================] - 1s 6ms/step

RMSE: 0.8388607442242638

*# Prediction results*

print('Confusion Matrix')

pd.DataFrame(confusion\_matrix(y\_test, pred\_keras2), index=[1,2,3,4,5], columns=[1,2,3,4,5])

**Output**:

Confusion Matrix

1 2 3 4 5

1 20 64 54 23 3

2 20 93 145 49 3

3 11 89 252 166 47

4 2 26 188 423 343

5 0 4 103 676 1725

### 

## Text Classification (Automated Deep Learning)

As mentioned before, text classification is another type of text analysis. Emotions detection, movie genres classification, and book types classification are examples of text classification. Unlike text regression, text classification does predict a continuous label, but a discrete label. For this exercise, we are going to use the [emotion label dataset](https://www.kaggle.com/anjaneyatripathi/emotion-classification-nlp). The dataset consists of only two columns: text and label. Our task is to create a prediction model to perceive the emotion of the text. The emotions are classified into 4 classes: anger, fear, joy, and sadness.

For text classification, let’s build two Deep Learning models. The first model applies the technic Long Short-Term Memory (LSTM) model. LSTM model is a Deep Learning model under Recurrent Neural Network (RNN). LSTM is a more advanced model compared to the usual multilayer perceptron Deep Learning. This article will not discuss further RNN or LSTM, but will only apply it for text classification. The LSTM for text classifier notebook is available [here](https://gitlab.com/rendy_k/lstm-text-classifier/-/blob/main/LSTM_Text_Classification.ipynb).

For data preparation, we apply one-hot-encoding to the data label. It creates 4 columns from 1 label column.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import make\_scorer, accuracy\_score, classification\_report, confusion\_matrix

# Load dataset

train = pd.read\_csv('/content/emotion-labels-train.csv')

test = pd.read\_csv('/content/emotion-labels-test.csv')

# Combine training and test datasets

train = pd.concat([train, test], axis=0)

train.head()

|  |  |  |
| --- | --- | --- |
|  | **text** | **label** |
| 0 | Just got back from seeing @GaryDelaney in Burs… | joy |
| 1 | Oh dear an evening of absolute hilarity I don’… | joy |
| 2 | Been waiting all week for this game  #ch… | joy |
| 3 | @gardiner\_love : Thank you so much, Gloria! Yo… | joy |
| 4 | I feel so blessed to work with the family that… | joy |

# One-hot-encoding

trainSet = train.reset\_index(drop=True)

labels = pd.get\_dummies(trainSet['label'])

trainSet = pd.concat([trainSet, labels], axis=1)

After splitting the training and validation datasets, a tokenizer is applied to tokenize the text. In this demonstration, the tokenizer will keep 5000 most common words. Then, “texts\_to\_sequences” transforms the texts in the dataset into sequences of integers.

# Split data

X\_train, X\_val, y\_train, y\_val = train\_test\_split(trainSet['text'].values,

trainSet[['anger','fear','joy','sadness']].values,

stratify=trainSet['label'],

test\_size=0.2, random\_state=123)

# Apply tokenizer and text to sequence

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

tokenizer = Tokenizer(num\_words=5000, oov\_token='x')

tokenizer.fit\_on\_texts(X\_train)

tokenizer.fit\_on\_texts(X\_val)

seq\_train = tokenizer.texts\_to\_sequences(X\_train)

seq\_val = tokenizer.texts\_to\_sequences(X\_val)

pad\_train = pad\_sequences(seq\_train)

pad\_val = pad\_sequences(seq\_val)

As for the “pad\_sequences”, let’s just look at this example.

[

[17, 154, 3],

[54, 981, 56, 4],

[20, 8]

]

is applied with pad\_sequences to be

[

[17 154 3 0],

[54 981 56 4],

[20 8 0 0]

]

Observe that the commas are removed and zeros are filled to make all the three lists have the same length.

Next, the code below shows LSTM layers creation using keras.Sequential. The LSTM will accept the input dimension of 5000 which is the number for the 5000 most common words from the tokenizer. After some ‘relu’ activation and dropout layers with several neurons, the last layer has 4 neurons with ‘softmax’ activation. This will classify a text into one of the 4 classes. The LSTM is compiled with ‘categorical\_loss entropy’ loss function, ‘adam’ optimizer, and ‘accuracy’ as the scoring metrics.

import tensorflow as tf

from keras.callbacks import EarlyStopping

# Create the model

lstm = tf.keras.Sequential([

tf.keras.layers.Embedding(input\_dim=5000, output\_dim=16),

tf.keras.layers.SpatialDropout1D(0.3),

tf.keras.layers.LSTM(128),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dropout(0.3),

tf.keras.layers.Dense(128, activation='relu'),

tf.keras.layers.Dense(64, activation='relu'),

tf.keras.layers.Dense(64, activation='relu'),

tf.keras.layers.Dropout(0.2),

tf.keras.layers.Dense(4, activation='softmax')

])

lstm.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

The LSTM model will train to find the optimal model in 100 epochs. But, before that, let’s create an early stopping callback. Early stopping is made to avoid spending too much time training the model while the expected target has been achieved or if there is no improvement of the training. In this exercise, the early stopping callback will stop the training before reaching the 100th epochs if a certain target is fulfilled. In this case, I set the target to reach the minimum accuracy of 85.5% for the validation dataset. When this happens, the LSTM model will stop training even though it has not reached 100 epochs. It will also print a message saying that the accuracy has reached more than 85.5%. While fitting the training data, the history is saved in the variable ‘history’.

class earlystop(tf.keras.callbacks.Callback):

def on\_epoch\_end(self, epoch, logs={}):

if(logs.get('val\_accuracy')>0.855):

print("Accuracy has reached > 85.5%!")

self.model.stop\_training = True

es = earlystop()

history = lstm.fit(pad\_train, y\_train, epochs=100, callbacks=[es],

validation\_data=(pad\_val, y\_val), verbose=2, batch\_size=100)

**Output:**

Epoch 1/100

55/55 - 5s - loss: 0.0348 - accuracy: 0.9771 - val\_loss: 2.1571 - val\_accuracy: 0.8505

Epoch 2/100

55/55 - 5s - loss: 0.0400 - accuracy: 0.9750 - val\_loss: 1.9905 - val\_accuracy: 0.8460

Epoch 3/100

55/55 - 5s - loss: 0.0386 - accuracy: 0.9728 - val\_loss: 2.0930 - val\_accuracy: 0.8386

Epoch 4/100

55/55 - 5s - loss: 0.0377 - accuracy: 0.9741 - val\_loss: 2.0917 - val\_accuracy: 0.8505

Epoch 5/100

55/55 - 5s - loss: 0.0392 - accuracy: 0.9743 - val\_loss: 2.1664 - val\_accuracy: 0.8431

Epoch 6/100

55/55 - 5s - loss: 0.0377 - accuracy: 0.9739 - val\_loss: 1.8948 - val\_accuracy: 0.8512

Epoch 7/100

55/55 - 5s - loss: 0.0353 - accuracy: 0.9782 - val\_loss: 2.3116 - val\_accuracy: 0.8564

Accuracy has reached > 85.5%!

Observe the output cell that the LSTM only runs until the 7th epoch and stops. As we have known, the 7th epoch validation dataset has an accuracy of 85.64%. Examine the training and validation datasets’ accuracy below.

# Visualize LSTM history

plt.plot(history.history['accuracy'])

plt.plot(history.history['val\_accuracy'])

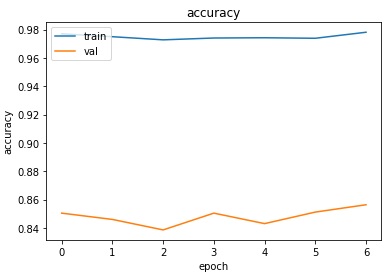
plt.title('accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'val'], loc='upper left')

plt.show()

Fig. 4 history of Deep Learning training (source: [image by author](https://gitlab.com/rendy_k/lstm-text-classifier/-/blob/main/LSTM_Text_Classification.ipynb))

Okay, now let’s try to build another model for emotion classification. This time, AutoKeras will be applied. To do this, the code can be similar to the text regressor AutoKeras performed before. Just change the “TextRegressor” into “TextClassifier”, then the rest will work the same. But, the code below will try to perform an advanced text classifier using AutoModel. With AutoModel, we can specify to use TextToIntSequence and Embedding to transform the texts into integer sequences and embed them. We can also specify to use separable convolutional layers. The AutoKeras Text Classifier notebook is available [here](https://www.kaggle.com/rendyk/autokerasclassification-emotion-prediction).

# Create the model

node\_input = ak.TextInput()

node\_output = ak.TextToIntSequence()(node\_input)

node\_output = ak.Embedding()(node\_output)

node\_output = ak.ConvBlock(separable=True)(node\_output)

node\_output = ak.ClassificationHead()(node\_output)

keras = ak.AutoModel(inputs=node\_input, outputs=node\_output, overwrite=True, max\_trials=3)

# Fit the training dataset

keras.fit(X\_train\_ak, y\_train\_ak, epochs=80, validation\_split=0.2)

Observe the Deep Learning architecture created by the AutoKeras. It has the layers of embedding, separable\_conv1d, and classification\_head as specified.

# Show the built models

keras\_export = keras.export\_model()

keras\_export.summary()

**Output:**

Model: "model"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None,)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

expand\_last\_dim (ExpandLastD (None, 1) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

text\_vectorization (TextVect (None, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

embedding (Embedding) (None, 64, 128) 2560128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout (Dropout) (None, 64, 128) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

separable\_conv1d (SeparableC (None, 62, 32) 4512

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

separable\_conv1d\_1 (Separabl (None, 60, 32) 1152

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling1d (MaxPooling1D) (None, 30, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

separable\_conv1d\_2 (Separabl (None, 28, 32) 1152

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

separable\_conv1d\_3 (Separabl (None, 26, 32) 1152

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling1d\_1 (MaxPooling1 (None, 13, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

global\_max\_pooling1d (Global (None, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense (Dense) (None, 4) 132

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

classification\_head\_1 (Softm (None, 4) 0

=================================================================

Total params: 2,568,228

Trainable params: 2,568,228

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The model has an accuracy of 0.80. Examine the confusion matrix generated below. It shows that the model can predict the emotion based on the text well. To know which emotion is predicted the best, pay attention to the f1-score. Joy has the highest f1-score, followed by fear, anger, and sadness respectively. It means that the model can predict the emotion of joy the best.

# Predict the validation data

pred\_keras = keras.predict(X\_val\_ak)

# Compute the accuracy

print('Accuracy: ' + str(accuracy\_score(y\_val\_ak, pred\_keras)))

**Output:**

25/25 [==============================] - 0s 3ms/step

Accuracy: 0.8042929292929293

# Convert predicted result into pandas series with numeric type

pred\_keras\_ = pd.DataFrame(pred\_keras)

pred\_keras\_ = pred\_keras\_[0]

# Prediction results

print('Confusion Matrix')

print(pd.DataFrame(confusion\_matrix(y\_val, pred\_keras\_), index=['anger','fear','joy','sad'], columns=['anger','fear','joy','sad']))

print('')

print('Classification Report')

print(classification\_report(y\_val, pred\_keras\_))

**Output:**

Confusion Matrix

anger fear joy sad

anger 146 6 8 28

fear 16 191 2 43

joy 9 3 166 2

sad 23 12 3 134

Classification Report

precision recall f1-score support

anger 0.75 0.78 0.76 188

fear 0.90 0.76 0.82 252

joy 0.93 0.92 0.92 180

sadness 0.65 0.78 0.71 172

accuracy 0.80 792

macro avg 0.81 0.81 0.80 792

weighted avg 0.82 0.80 0.81 792

## Conclusion

NLP aims to analyze a large number of text data. Some examples of the applications are for predicting sentiment and emotion from text using Machine Learning. Unlike numerical data, text data need special preprocessing, like BoW or TF-IDF, before fitting them to Machine Learning algorithms.

A regular expression is one of the basic NLP to find and split sentences or words. Word tokenizer splits sentences into words. After that, each tokenized word can be processed with NER, stemming, and lemmatization. Each word is counted for its frequency in the form of a BoW data frame. Stop words that appear frequently, but do not give any meaning, can be removed. Stop words can be identified with NER.

AutoKeras is a package to perform text regression and classification using Deep Learning. Just inputting the text feature will automatically build the model, including word tokenization and preprocessing, hyperparameter-tuning, and deciding the layers.

**About Author**

Connect with me [here](https://www.linkedin.com/in/rendy-kurnia/).

**The media shown in this article is not owned by Analytics Vidhya and are used at the Author’s discretion**

### Related

[](https://www.analyticsvidhya.com/blog/2022/01/master-natural-language-processing-in-2022-with-best-resources/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=0)

#### [Master Natural Language Processing in 2022 with Best Resources](https://www.analyticsvidhya.com/blog/2022/01/master-natural-language-processing-in-2022-with-best-resources/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=0&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=0)

[](https://www.analyticsvidhya.com/blog/2021/06/how-to-perform-basic-text-analysis-without-training-dataset/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=1)

#### [How to Perform Basic Text Analysis without Training Dataset](https://www.analyticsvidhya.com/blog/2021/06/how-to-perform-basic-text-analysis-without-training-dataset/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=1&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=1)

[Graphical user interface, diagram

Description automatically generated](https://www.analyticsvidhya.com/blog/2021/07/understanding-natural-language-processing-a-beginners-guide/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=2)

#### [Understanding Natural Language Processing -A Beginner’s Guide](https://www.analyticsvidhya.com/blog/2021/07/understanding-natural-language-processing-a-beginners-guide/?relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=2&relatedposts_hit=1&relatedposts_origin=86187&relatedposts_position=2)

[Automated deep learning](https://www.analyticsvidhya.com/blog/tag/automated-deep-learning/)[AutoML](https://www.analyticsvidhya.com/blog/tag/automl/)[blogathon](https://www.analyticsvidhya.com/blog/tag/blogathon/)[text prediction](https://www.analyticsvidhya.com/blog/tag/text-prediction/)

[A person's face on a blue background

Description automatically generated with low confidence](https://datahack.analyticsvidhya.com/contest/datahour-build-your-first-chat-bot-using-open-source-tools/?utm_source=sidebanner&utm_medium=blog&utm_campaign=datahour)

### About the Author



###### [rendyk](https://www.analyticsvidhya.com/blog/author/rendyk/)

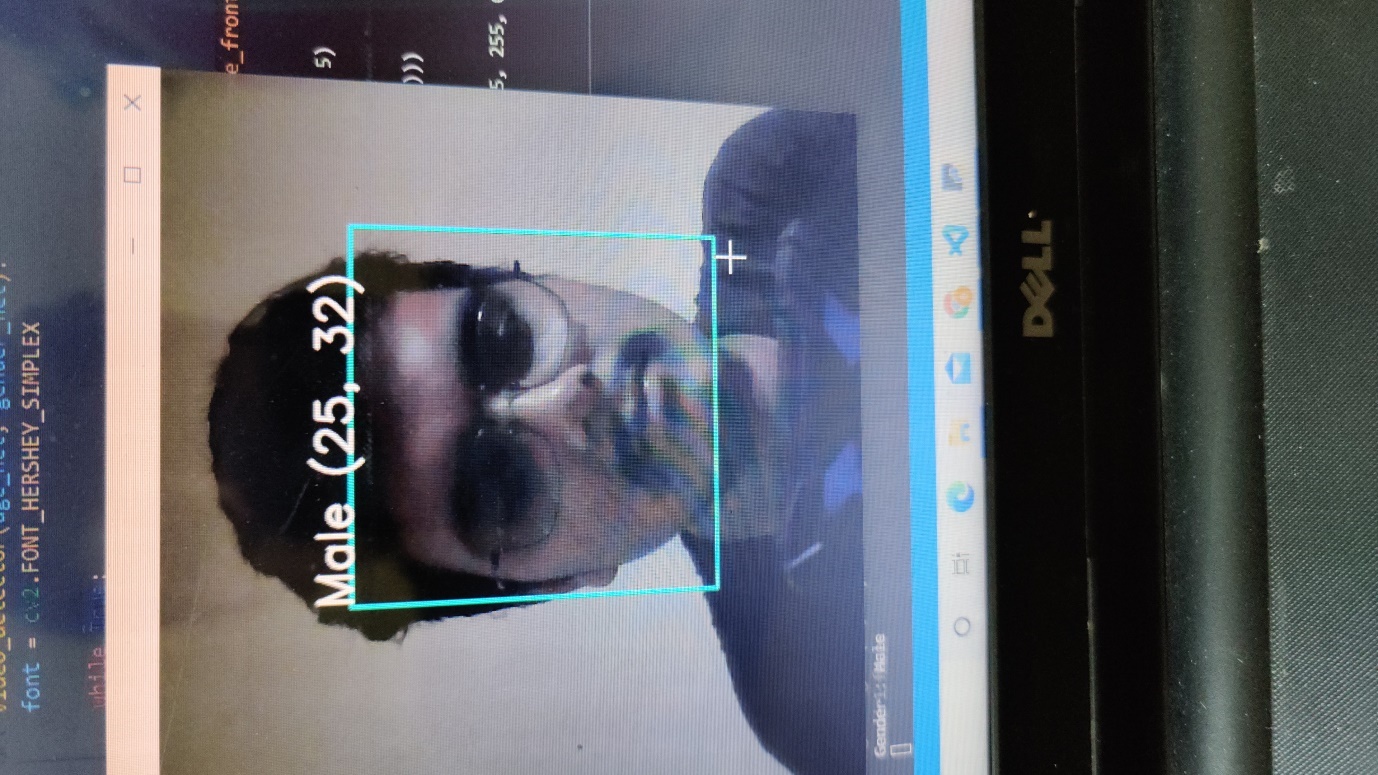
### Our Top Authors

[](https://www.analyticsvidhya.com/blog/author/rahul105/)

[](https://www.analyticsvidhya.com/blog/author/sion/)

[](https://www.analyticsvidhya.com/blog/author/chirag676/)

[](https://www.analyticsvidhya.com/blog/author/barney6/)

[](https://www.analyticsvidhya.com/blog/author/arnab1408/)

[](https://www.analyticsvidhya.com/blog/author/shanthababu/)

[](https://www.analyticsvidhya.com/blog/author/prateekmaj21/)

* 

[**view more**](https://www.analyticsvidhya.com/blog/creators/?utm-medium=blog-footer&utm_source=top-authors)

### DownloadAnalytics Vidhya App for the Latest blog/Article

**Previous Post**[A Tool for Investor – The Art of Web Scraping](https://www.analyticsvidhya.com/blog/2021/11/a-guide-to-automated-deep-machine-learning-for-natural-language-processing-text-prediction/)

**Next Post**[Optimizing Pokemon Team using Python’s PuLP Library](https://www.analyticsvidhya.com/blog/2021/11/a-guide-to-automated-deep-machine-learning-for-natural-language-processing-text-prediction/)

### **Leave a Reply**Your email address will not be published. Required fields are marked \*

Top of Form







 Notify me of follow-up comments by email.

 Notify me of new posts by email.

Submit

Bottom of Form

## Top Resources

[[Graphical user interface, application

Description automatically generated](https://www.analyticsvidhya.com/blog/2021/08/python-tutorial-working-with-csv-file-for-data-science/)](https://www.analyticsvidhya.com/blog/2021/08/python-tutorial-working-with-csv-file-for-data-science/)

[Python Tutorial: Working with CSV file for Data Science](https://www.analyticsvidhya.com/blog/2021/08/python-tutorial-working-with-csv-file-for-data-science/)

[Harika Bonthu -](https://www.analyticsvidhya.com/blog/author/harikabonthu96/)AUG 21, 2021

[[](https://www.analyticsvidhya.com/blog/2020/02/joins-in-pandas-master-the-different-types-of-joins-in-python/)](https://www.analyticsvidhya.com/blog/2020/02/joins-in-pandas-master-the-different-types-of-joins-in-python/)

[Joins in Pandas: Master the Different Types of Joins in..](https://www.analyticsvidhya.com/blog/2020/02/joins-in-pandas-master-the-different-types-of-joins-in-python/)

[Abhishek Sharma -](https://www.analyticsvidhya.com/blog/author/abhishek-shrm/)FEB 27, 2020

[[Text

Description automatically generated](https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/)](https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/)

[Understanding Support Vector Machine(SVM) algorithm from examples (along with code)](https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/)

[sunil -](https://www.analyticsvidhya.com/blog/author/sunil-ray/)SEP 13, 2017

[[A picture containing grass, tree, plant

Description automatically generated](https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/)](https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/)

[Understanding Random Forest](https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/)

[Sruthi E R -](https://www.analyticsvidhya.com/blog/author/sruthi94/)JUN 17, 2021

[](https://www.analyticsvidhya.com/)

###### Download App

###### Analytics Vidhya

[About Us](https://www.analyticsvidhya.com/about/)[Our Team](https://www.analyticsvidhya.com/team/)[Careers](https://www.analyticsvidhya.com/careers/)[Contact us](https://www.analyticsvidhya.com/contact/)

###### Data Scientists

[Blog](https://www.analyticsvidhya.com/blog/)[Hackathon](https://datahack.analyticsvidhya.com/)[Discussions](https://discuss.analyticsvidhya.com/)[Apply Jobs](https://www.analyticsvidhya.com/jobs/)

###### Companies

[Post Jobs](https://www.analyticsvidhya.com/corporate/)[Trainings](https://courses.analyticsvidhya.com/)[Hiring Hackathons](https://datahack.analyticsvidhya.com/)[Advertising](https://www.analyticsvidhya.com/contact/)

Visit us **Step 3:**The data extraction part was traditional i.e. to scrap it from the portals. I used web scraping to get all the data. Now it was time to perform the data cleaning and sentiment analysis. Cleaning the data was a time-consuming task as the review contained sentences with irrelevant things that were not useful for my objective. For data cleaning, I used regular expressions and a few ready-made  
python libraries. The corpus(data) was now ready to be analyzed.

© Copyright 2013-2022 Analytics Vidhya.

[Privacy Policy](https://www.analyticsvidhya.com/privacy-policy/) [Terms of Use](https://www.analyticsvidhya.com/terms/) [Refund Policy](https://www.analyticsvidhya.com/refund-policy/)