Sentiment Analysis

By

Tom Barasa

03/01/2022

Research Study, objectives and goals.

A 1. Research Study

- Build a neural network model able to classify customers, review sentiments and make predictions. Our data contains 1000 Amazon customer reviews, from UCI sentiment dataset, titled amazon cells labelled. Positive reviews will be predicted as a 1 and negative reviews predicted as a 0.
- Data source : (https://archive.ics.uci.edu/ml/datasets/Sentiment+Labelled+Sentences).

A2. Objectives and Goals.

- Build a neural network designed to learn word usage and context using NLP techniques. Determine if a given review text expresses positive or negative sentiment. Analyze the body of the text and understand our customer opinion, modality and mood.

Neural network capable of performing a text classification task

Convolutional Neural Network(CNN) works well with text classification.

"Text classification involves the use of word embedding, for representing words and CNN for learning how to discriminate documents on classification problems. The non-linearity of the network, as well as the ability to easily integrate pre-trained word embeddings, often lead to superior classification accuracy and predictions".

(https://arxiv.org/abs/1510.00726, A Primer on Neural Network Models for Natural Language Processing , Yoav Goldberg, October 2, 2015).

Data Exploration

1). During these stage, we will have a general overview of our data in terms of :

- General overview such as shape, create new column text length and check number rows.
- Data distribution and understand how many rows have positive and negative reviews.
- Check number of characters in the text feature.
- Count the number of words in the text feature.
- Count number of unique words.
- Check the mean word length.
- Count punction in the text.
- Analyze text Unigrams, Bigrams and N-grams.
- Create word cloud for our target words.
- Check for words frequency in our text review.

General overview

- Check data shape, top five rows and text length.

In [9]:
```
# Check shape of our dataset
df.shape
```

Out[9]:
```
(1000, 3)
```

In [10]:
```
# Create new column called "text length" which is number of words in the text column.
df['text_length'] = df['text'].apply(len)
```

In [12]:
```
df.head()
```

Out[12]:

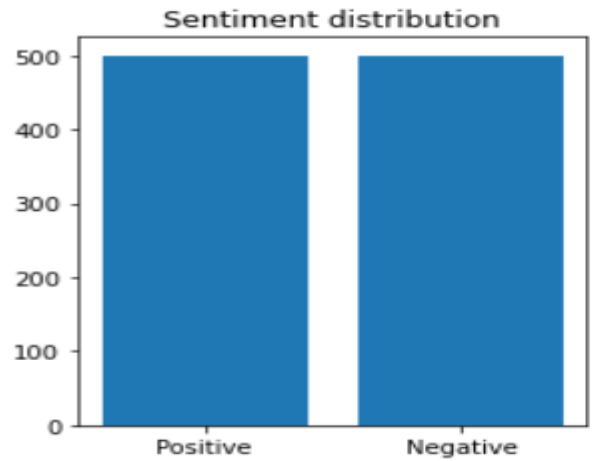
	text	label	text_length
0	So there is no way for me to plug it in here i...	0	82
1	Good case, Excellent value.	1	27
2	Great for the jawbone.	1	22
3	Tied to charger for conversations lasting more...	0	79
4	The mic is great.	1	17

Create our sentiment category and visualize review distribution based on negative or positive reviews.

```
In [11]: # Create sentimet Category of our reviews
         positive = df[df.label == 1].shape[0]
         negative = df[df.label == 0].shape[0]
```

```
In [595]: # Plot sentiment distribution .
          plt.figure(1, figsize=(8, 4))
          plt.subplot(1, 2 , 1)
          _ = plt.bar(["Positive", "Negative"], [positive, negative])

          plt.title('Sentiment distribution')
```

```
Out[595]: Text(0.5, 1.0, 'Sentiment distribution')
```



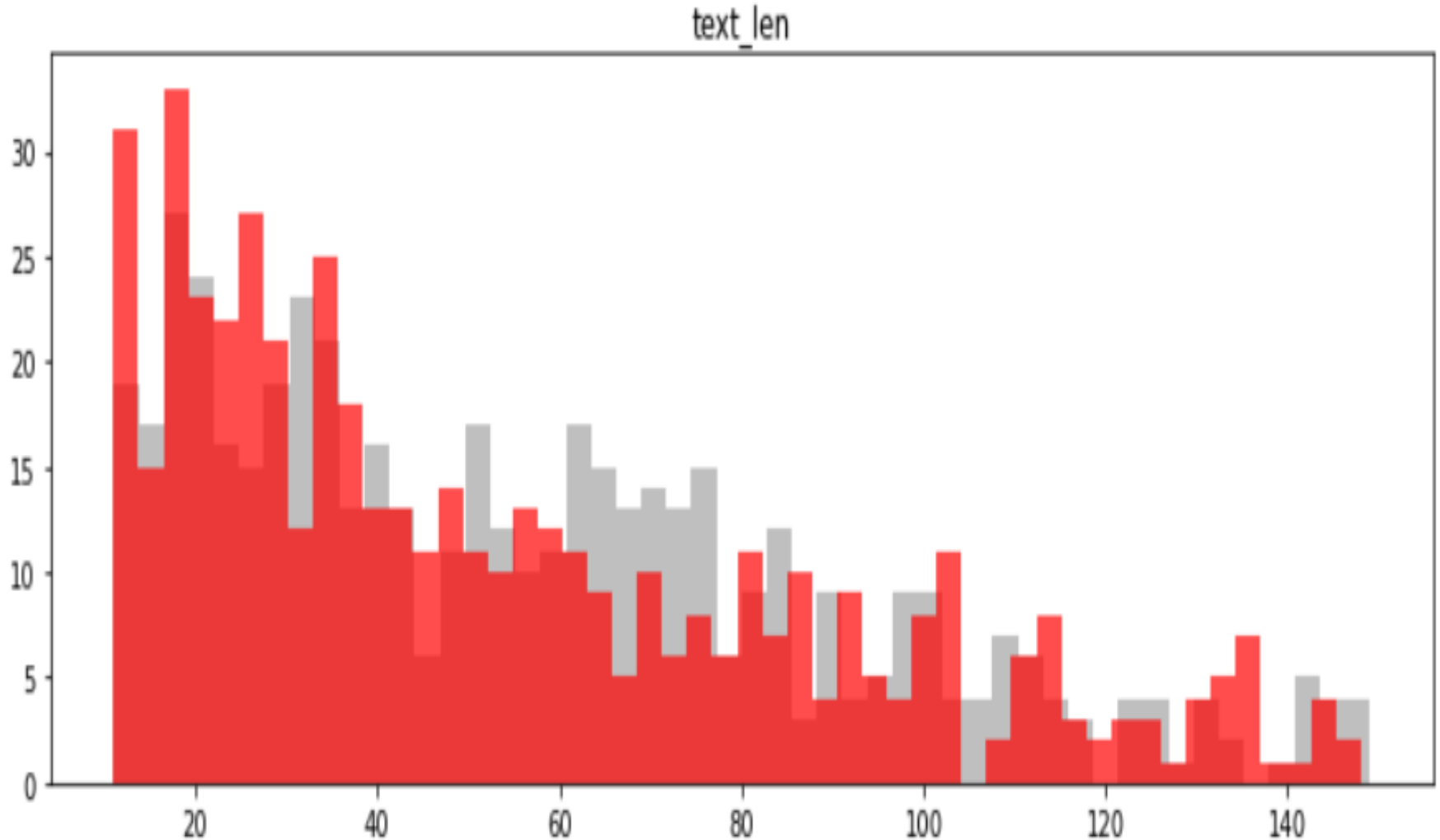
Checking the number of characters in the text feature.

```
.3]: def plot_Label_based_features(feature):
         x1 = yelp[yelp.Label == 1][feature]
         x2 = yelp[yelp. Label == 0][feature]
         plt.figure(1, figsize=(12, 4))
         plt.subplot(1, 1 , 1)
         _ = plt.hist(x2, alpha=0.5, color="grey", bins=50)
         _ = plt.hist(x1, alpha=0.7, color="red", bins=50)

         return _
```

```
.4]: # Check the number of characters in the text feature
     yelp["text_len"] = yelp.text.map(lambda x: len(x))
```

```
.5]: _ = plot_Label_based_features("text_len")

     plt.title('text_len')
```

```
.5]: Text(0.5, 1.0, 'text_len')
```

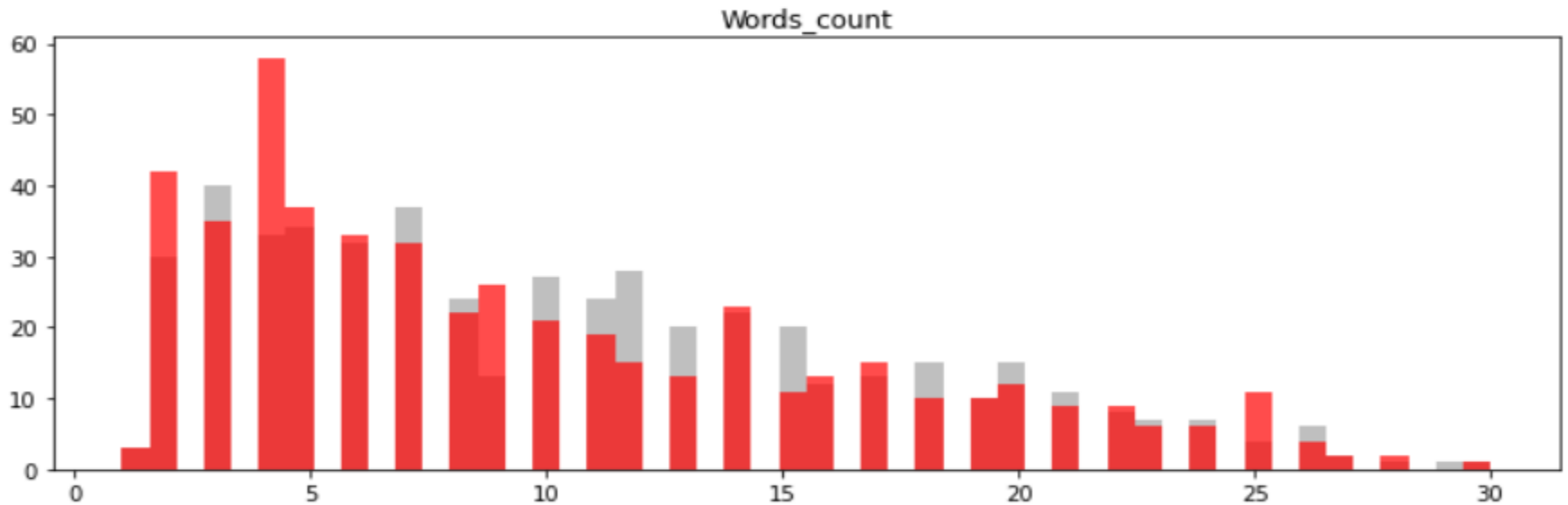

text_len
30
25
20
15
10
5
0
20
40
60
80
100
120
140

Counting the number of words in the text feature

```
5]: # Count the number of words in the text feature
    yelp["words_count"] = yelp.text.str.split().map(lambda x: len(x))
```

```
7]: _ = plot_Label_based_features("words_count")
    plt.title('Words_count')
```

```
7]: Text(0.5, 1.0, 'Words_count')
```

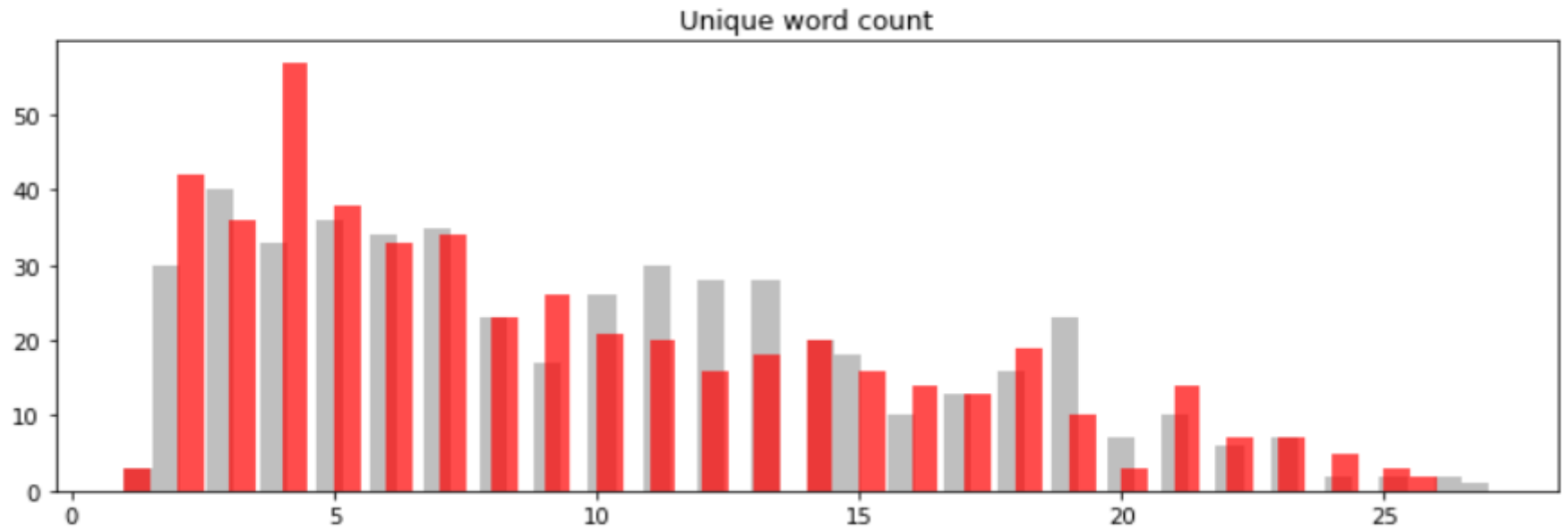


Counting the number of unique words.

In [18]:
```
# Count the number of unique words
yelp["unique_word_count"] = yelp.text.map(lambda x: len(set(str(x).split())))
```

In [19]:
```
_ = plot_Label_based_features("unique_word_count")
plt.title('Unique word count')
```

Out[19]:
```
Text(0.5, 1.0, 'Unique word count')
```

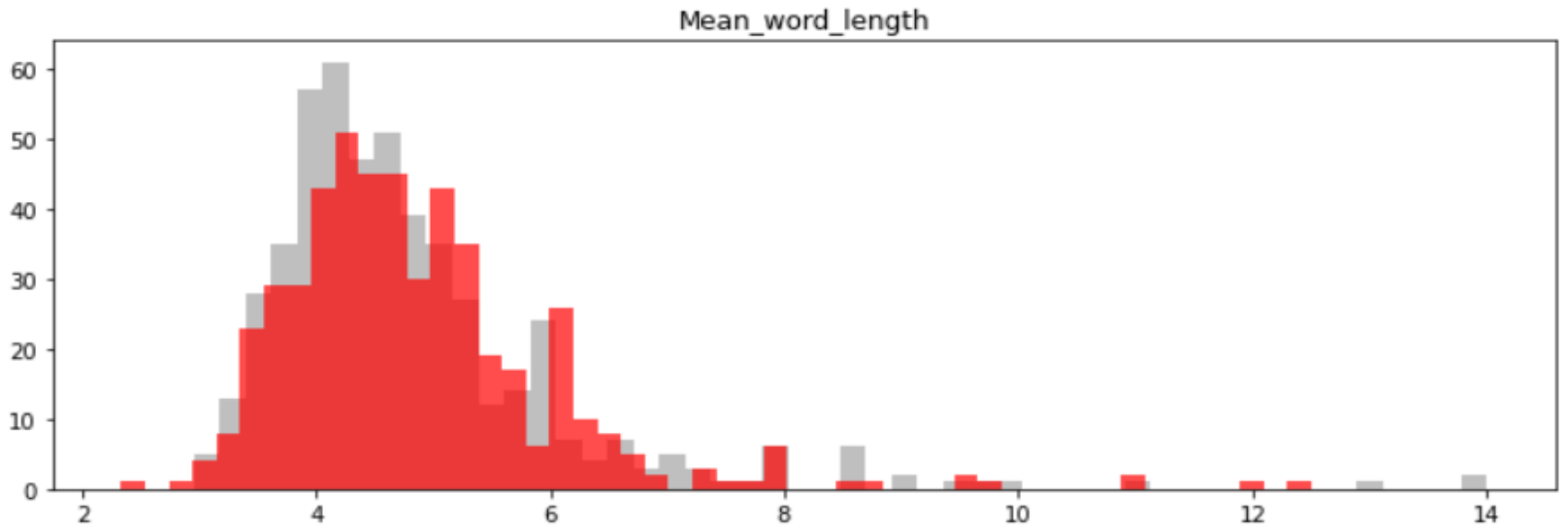


Checking the Mean Word Length

```
# Mean word length
yelp["mean_word_length"] = yelp.text.map(
    lambda x: np.mean([len(w) for w in str(x).split()])
)
```

```
_ = plot_Label_based_features("mean_word_length")
plt.title('Mean_word_length')
```

```
Text(0.5, 1.0, 'Mean_word_length')
```

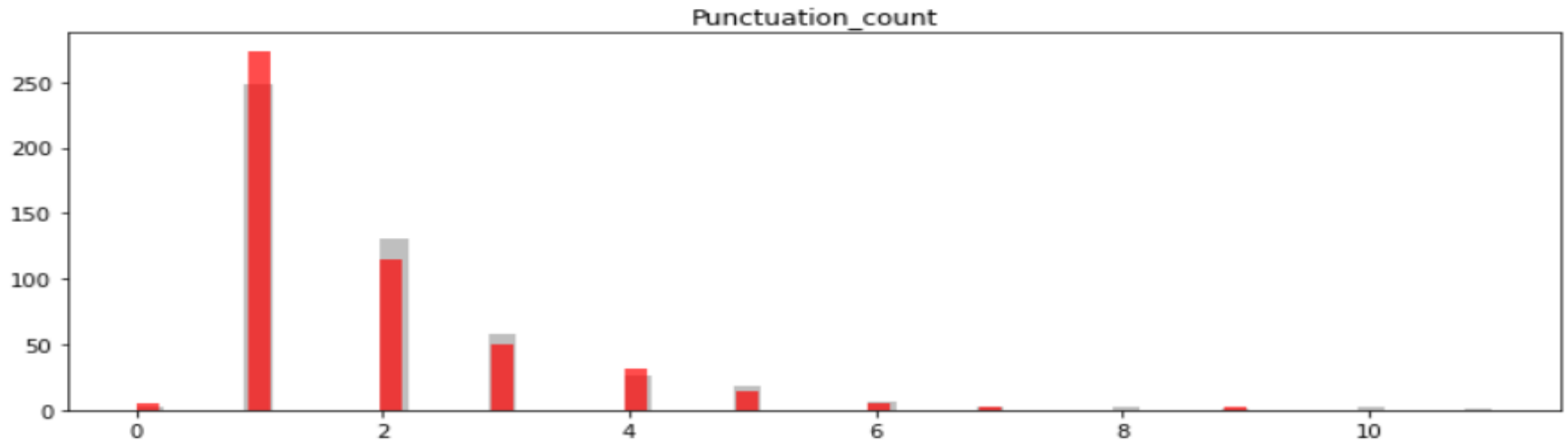


Checking Punctuation Count in the text.

```
# Punction count
import string

yelp["punctuation_count"] = yelp["text"].map(
    lambda x: len([c for c in str(x) if c in string.punctuation])

)
```

```
_ = plot_Label_based_features("punctuation_count")

plt.title('Punctuation_count')
```

```
Text(0.5, 1.0, 'Punctuation_count')
```



Unigrams, Bigrams and N-Grams Analysis

- N-grams are neighboring sequences of items (words, letters or symbols) in a corpus or document.
- To get the best results, we will need to analyze multiple n-grams to see what works best for each science problem.

N-Grams Analysis.

```
from nltk.corpus import stopwords
```

```
def generate_ngrams(text, n_gram=1, stop=True):
    """
    Simple n-gram generator
    """
    stop = set(stopwords.words("english")) if stop else {}
    token = [
        token for token in text.lower().split(" ")if token != " " if token not in stop
    ]
    z = zip(*[token[i:] for i in range(n_gram)])
    ngrams = ["".join(ngram) for ngram in z]

    return ngrams
```

Unigrams Analysis

- When a new word is encountered and is missing from the mapping, the default_factory function will call int() to supply a default count of zero and then, the increment operation builds up the count.
- Therefore, the default_factory, int() will assign the defaultdict useful for counting.

In [603]:
```
# Create unigrams
from collections import defaultdict
positive_unigrams = defaultdict(int)
negative_unigrams = defaultdict(int)
for text in df[df.label == 1].text:
    for word in generate_ngrams(text):
        positive_unigrams[word] += 1

for text in df[df.label == 0].text:
    for word in generate_ngrams(text):
        negative_unigrams[word] += 1
```

```
positive_unigrams.items()
```

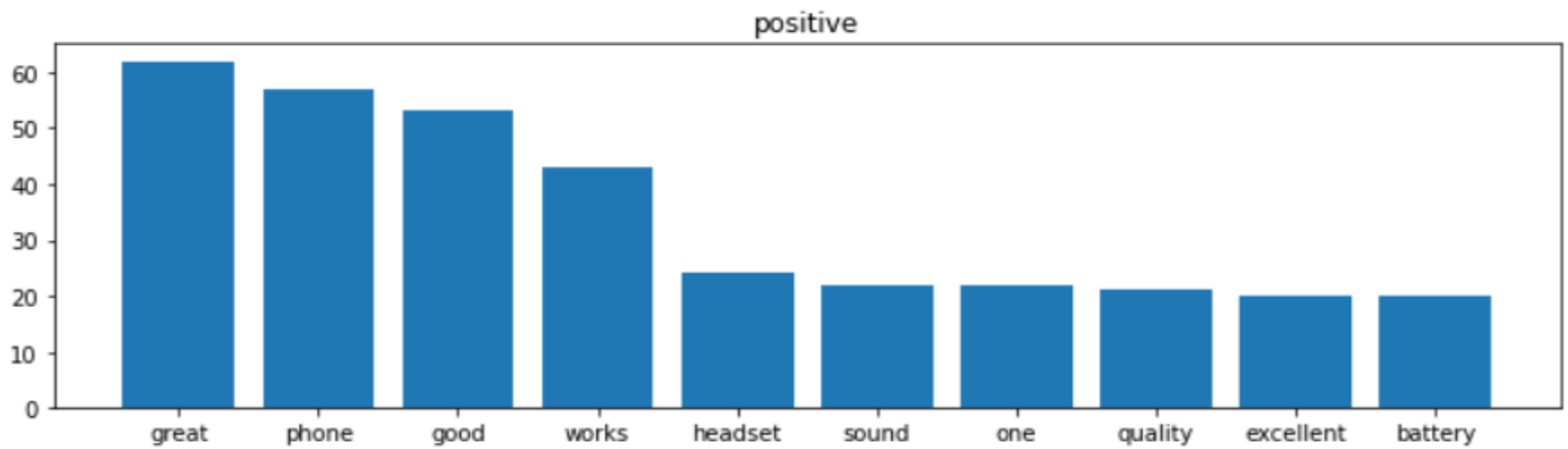
```
dict_items([('good', 53), ('case,', 3), ('excellent', 20), ('value.', 4), ('great', 62), ('jawbone.', 1), ('mic', 1), ('grea
t.', 15), ('razr', 3), ('owner...you', 1), ('must', 3), ('this!', 1), ('sound', 22), ('quality', 21), ('impressed', 5), ('goi
ng', 2), ('original', 4), ('battery', 20), ('extended', 2), ('battery.', 1), ('though', 1), ('highly', 8), ('recommend', 18),
('one', 22), ('blue', 3), ('tooth', 1), ('phone.', 20), ('far', 3), ('good!.', 1), ('works', 43), ('great!.', 3), ('bought',
8), ('use', 13), ('kindle', 1), ('fire', 2), ('absolutely', 3), ('loved', 1), ('it!', 3), ('yet', 1), ('run', 2), ('new', 1
0), ('two', 6), ('bars', 1), ("that's", 1), ('three', 2), ('days', 2), ('without', 7), ('charging.', 1), ('pocket', 2), ('p
c', 2), ('/', 1), ('phone', 57), ('combination.', 1), ("i've", 19), ('owned', 1), ('7', 1), ('months', 2), ('say', 4), ('bes
t', 19), ('mobile', 2), ('had.', 4), ('product', 15), ('ideal', 1), ('people', 6), ('like', 18), ('whose', 1), ('ears', 3),
('sensitive.', 1), ('car', 8), ('charger', 5), ('well', 17), ('ac', 1), ('included', 1), ('make', 4), ('sure', 3), ('never',
2), ('juice.highy', 1), ('recommended', 1), ('kept', 1), ('well.', 12), ('case', 7), ('fine', 6), ('680.', 1), ('camera', 7),
('thats', 2), ('2mp,', 1), ('pics', 1), ('nice', 19), ('clear', 7), ('picture', 1), ('quality.', 8), ('headset', 24), ('price
d', 4), ('right.', 3), ('bluetooth', 10), ('headset.', 4), ('features', 3), ('want', 1), ('seems', 4), ('made.', 2), ('protec
tion', 2), ('bulky.', 1), ('usable', 1), ('keyboard', 3), ('actually', 2), ('turns', 1), ('pda', 2), ('real-world', 1), ('use
ful', 1), ('machine', 1), ('instead', 2), ('neat', 1), ('gadget.', 1), ('pretty', 8), ('sturdy', 4), ('large', 1), ('problem
s', 4), ('it.', 13), ('love', 20), ('thing!', 1), ('everything', 6), ('reasonable', 1), ('price', 11), ('i.e.', 1), ('even',
11), ('dropped', 2), ('stream', 1), ('submerged', 1), ('15', 1), ('seconds', 1), ('still', 4), ('great!', 5), ('happy', 13),
('510', 2), ('complaints', 1), ('regarding', 1), ('end.', 1), ('really', 15), ('faceplates', 1), ('since', 3), ('looks', 4),
('nice,', 1), ('elegant', 1), ('cool.', 2), ('headphones', 2), ('find', 2), ('-', 14), ('think', 4), ('perhaps', 1), ('purcha
se', 1), ('made', 5), ('last', 2), ('several', 4), ('years', 6), ('seriously.', 1), ('feels', 3), ('comfortable', 11), ('head
```

In [605]:
```
# Text in index one in our data.
df_positive_unigrams = pd.DataFrame(
    sorted(positive_unigrams.items(), key=lambda x: x[1], reverse=True)
)
df_negative_unigrams = pd.DataFrame(
    sorted(negative_unigrams.items(), key=lambda x: x[1], reverse=True)
)
sorted(positive_unigrams.items(), key=lambda x: x[1], reverse=True)[:5]
```

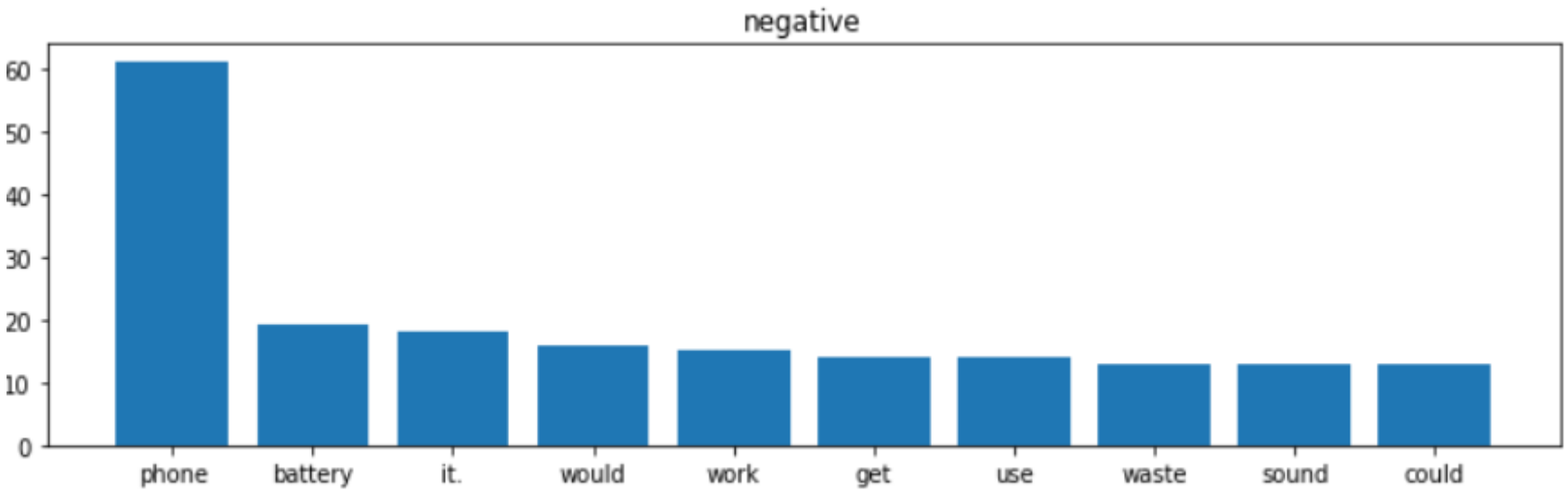
Out[605]:
```
[('great', 62), ('phone', 57), ('good', 53), ('works', 43), ('headset', 24)]
```

In [40]:
```
# Visualizing the unigrams in our text.
# Positive
d1 = df_positive_unigrams[0][:10]
d2 = df_positive_unigrams[1][:10]
# Negative
nd1 = df_negative_unigrams[0][:10]
nd2 = df_negative_unigrams[1][:10]
plt.figure(1, figsize=(12, 7))
plt.subplot(2,1,1)
_ = plt.bar(d1, d2)
plt.title('positive')
plt.subplot(2, 1, 2)
_ = plt.bar(nd1, nd2)
plt.title('negative')
```

Out[40]:
```
Text(0.5, 1.0, 'negative')
```



Negative unigrams



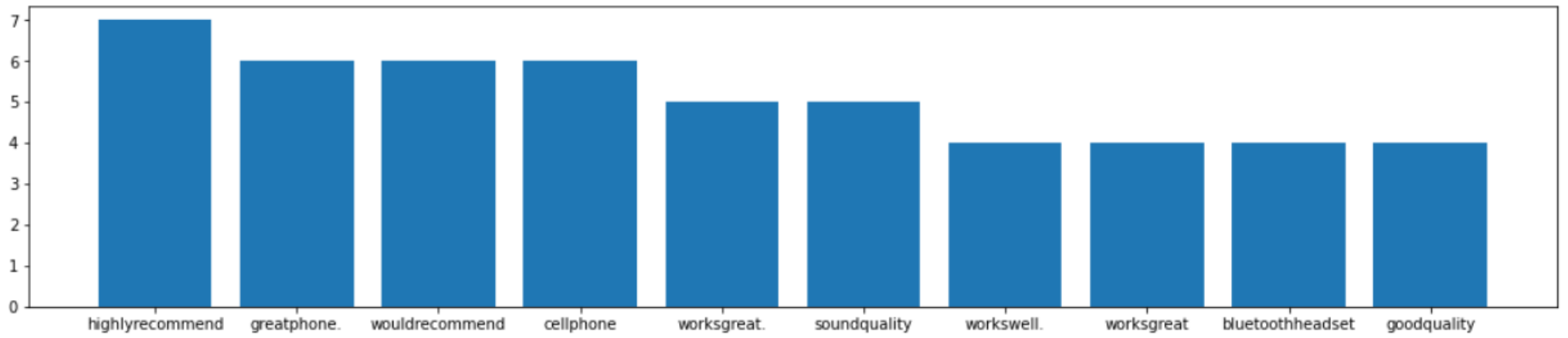
- Based on the two plots above, we get the most common shared unigrams in our text whether is referring to positive or negative context.

Bigrams Analysis

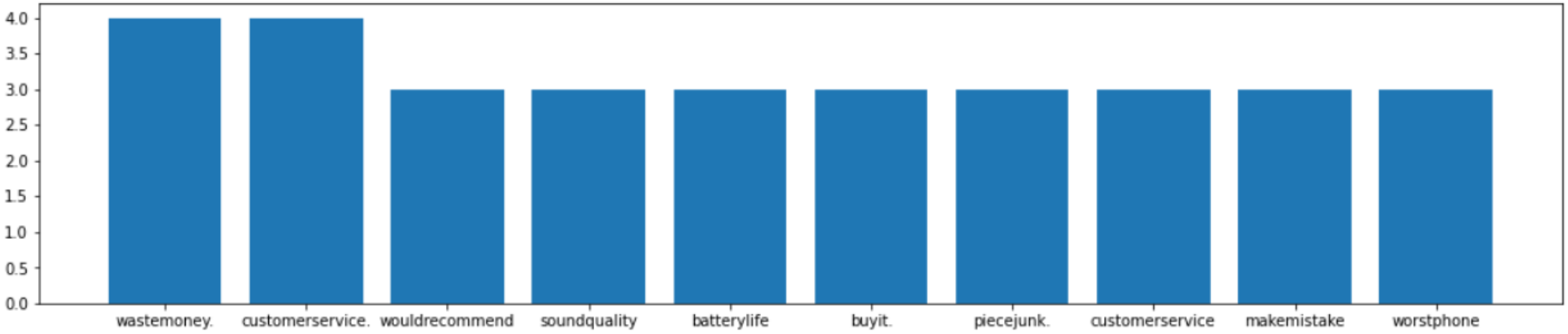
In [607]:
```
# Create bigrams
positive_bigrams = defaultdict(int)
negative_bigrams = defaultdict(int)
for text in df[df.label == 1]. text:
    for word in generate_ngrams(text, n_gram=2):
        positive_bigrams[word] += 1
for text in df[df.label == 0]. text:
    for word in generate_ngrams(text, n_gram=2):
        negative_bigrams[word] += 1
df_positive_bigrams = pd.DataFrame(
    sorted(positive_bigrams.items(), key=lambda x: x[1])[: : -1]
)
df_negative_bigrams = pd.DataFrame(
    sorted(negative_bigrams.items(), key=lambda x: x[1])[: : -1]
)
```

Positive Bigrams

In [42]:
```
# Visualizing both positive and negative bigrams
d1 = df_positive_bigrams[0][:10]
d2 = df_positive_bigrams[1][:10]
nd1 = df_negative_bigrams[0][:10]
nd2 = df_negative_bigrams[1][:10]
plt.figure(1, figsize=(18, 8))
plt.subplot(2, 1, 1)
_ = plt.bar(d1, d2)
plt.subplot(2, 1 , 2)
_ = plt.bar(nd1, nd2)
```



Negative Bigrams



Checking most common bigrams in the corpus .

```
from sklearn.feature_extraction.text import CountVectorizer


def get_top_text_ngrams(corpus, ngrams=(1, 1), nr=None):

    """
    Creates a bag of ngrams and counts ngram frequency.

    Returns a sorted list of tuples: (ngram, count)

    """

    vec = CountVectorizer(ngram_range=ngrams).fit(corpus)
    bag_of_words = vec.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx in vec.vocabulary_.items()]
    words_freq = sorted(words_freq, key=lambda x: x[1], reverse=True)
    return words_freq[:nr]
```

Top 5 rows of our bigrams

```
# Print top 5 rows
top_text_bigrams = get_top_text_ngrams(df.text, ngrams=(2, 2), nr=10)
top_text_bigrams
```

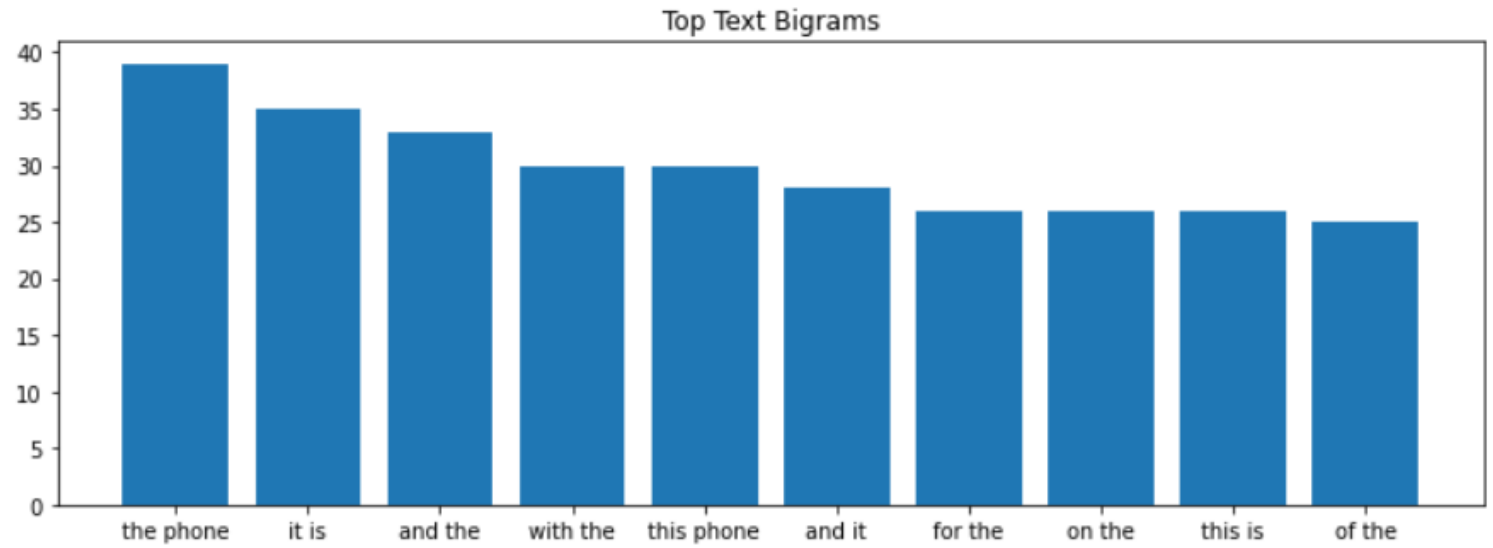
Out[609]:

```
[('the phone', 39),
 ('it is', 35),
 ('and the', 33),
 ('with the', 30),
 ('this phone', 30),
 ('and it', 28),
 ('for the', 26),
 ('on the', 26),
 ('this is', 26),
 ('of the', 25)]
```

Visualizing top text bigrams

In [610]:
```
# Visualize top bigrams
X, y = zip(*top_text_bigrams)
plt.figure(1, figsize=(12, 4))
plt.subplot(1, 1, 1)
plt.bar(X, y)
plt.title('Top Text Bigrams')
```

Out[610]:
```
Text(0.5, 1.0, 'Top Text Bigrams')
```



Data Cleaning Process

- When dealing with numerical data, data cleaning involves removing null values, duplicates, outliers, etc.
- With text data, we use text pre-processing techniques to clean our data. In these text:

1).First we will visualize punctuations, emojis, URLS, HTML and everything in the html tag and stop words in our text.

2).Remove URLS, HTML and everything in the html tag

3). Remove punctuations.

4). Remove emojis

5). Remove stop words.

In [611]:
```
def create_corpus(df, label):
    """
    Create corpus based on our target label.
    """
    corpus = []
    for X in df[df['label'] == label].text.str.split():
        for i in X:
            corpus.append(i)
    return corpus
```

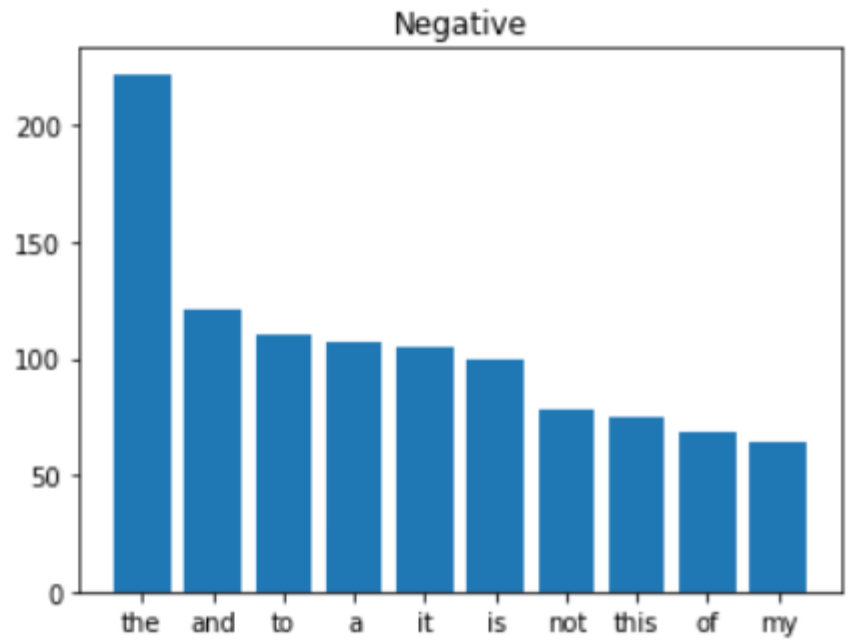
Checking for Stop words.

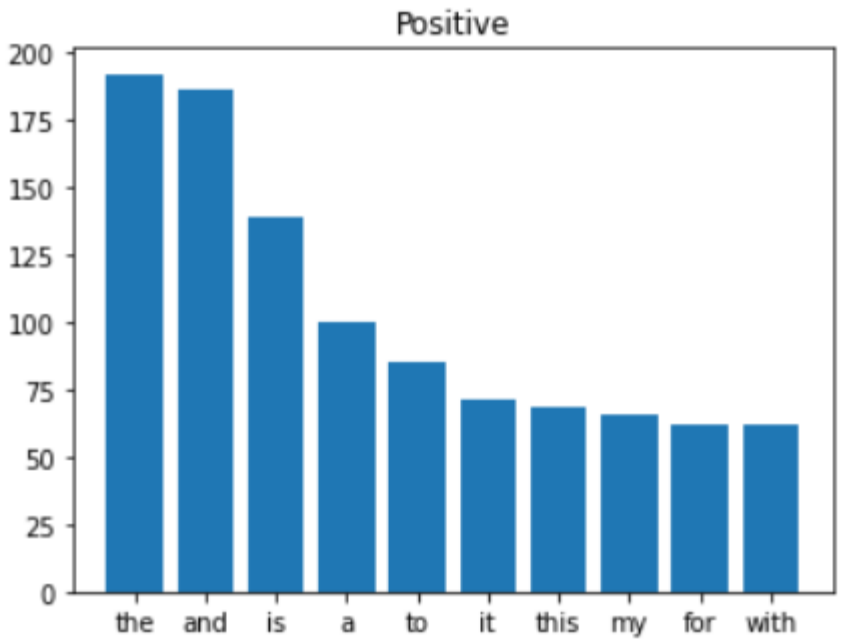
In [612]:
```
# Check for Stopwords in our text corpus
from nltk.corpus import stopwords
stop = set(stopwords.words('english'))
from collections import defaultdict
corpus0 = create_corpus(df=df, label=0)
corpus1 = create_corpus(df=df, label=1)
d0 = defaultdict(int)
for word in corpus0:
    if word in stop:
        d0[word] += 1
d1 = defaultdict(int)
for word in corpus1:
    if word in stop:
        d1[word] += 1
```

In [613]:
```
top0 = sorted(d0.items(), key=lambda x: x[1], reverse=True)[:10]
top1 = sorted(d1.items(), key=lambda x: x[1], reverse=True)[:10]
x0,  y0 = zip(*top0)
x1,  y1 = zip(*top1)
```

In [614]:
```
# Visualize our stopwords
plt.figure(1, figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.bar(x0, y0)
plt.title('Negative')
plt.figure(1, figsize=(16, 4))
plt.subplot(1, 2, 2)
plt.bar(x1, y1)
plt.title('Positive')
```

Out[614]: Text(0.5, 1.0, 'Positive')





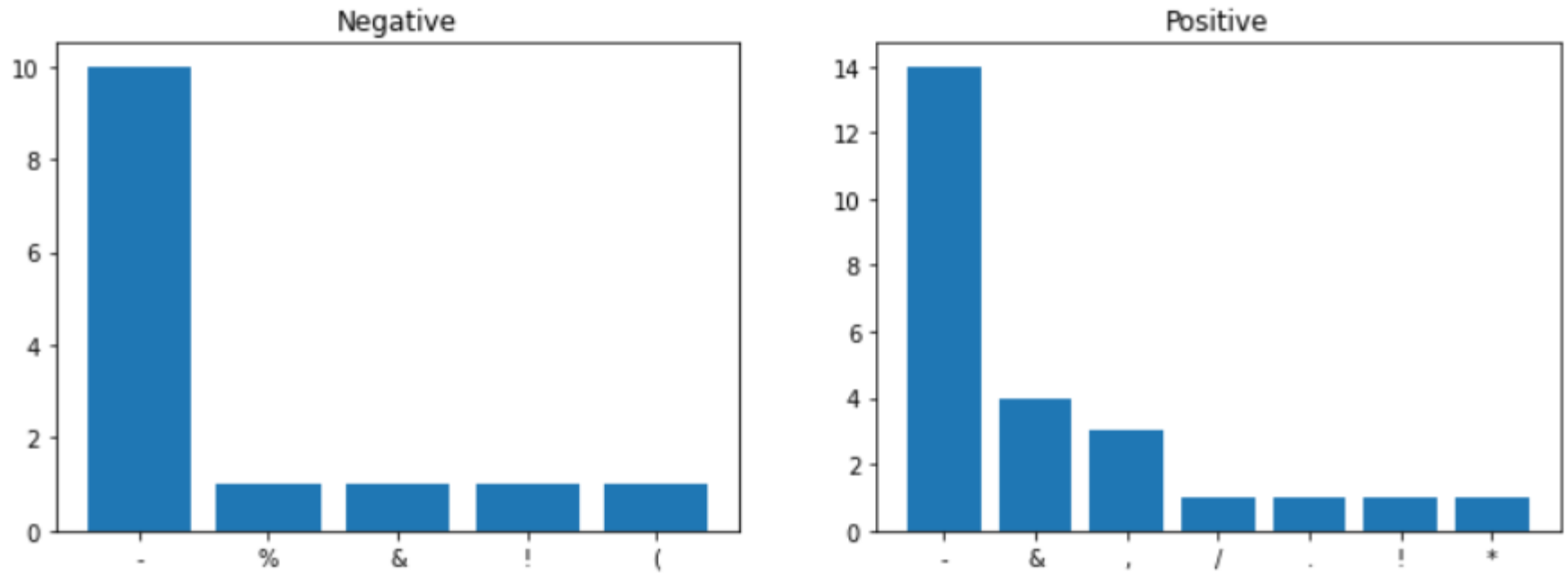
Checking for Punctions in the Corpus

In [615]:
```
# Check for Punctuations in the Corpus
import string
punc0 = defaultdict(int)
for word in corpus0:
    if word in string.punctuation:
        punc0[word] += 1
punc1 = defaultdict(int)
for word in corpus1:
    if word in string.punctuation:
        punc1[word] += 1
```

In [616]:
```
top0 = sorted(punc0.items(), key=lambda X: X[1], reverse=True)[:10]
top1 = sorted(punc1.items(), key=lambda X: X[1], reverse=True)[:10]
x0, y0 = zip(*top0)
x1, y1 = zip(*top1)
```

In [617]:
```
# Visualize our punctuations.
plt.figure(1, figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.bar(x0, y0)
plt.title('Negative')
plt.figure(1, figsize=(16, 4))
plt.subplot(1, 2, 2)
plt.bar(x1, y1)
plt.title('Positive')
```

Out[617]: Text(0.5, 1.0, 'Positive')



Removing URLS, HTML and everything in html tag.

In [618]:
```
# Remove URLS, HTML and everything in html tag
import re
def remove_URL(text):
    url = re.compile(r"https?://\S+/www\.\S+")
    return url.sub(r"", text)
def remove_html(text):
    html = re.compile(r"<.*?>")
    return html.sub(r"", text)
```

Remove Emojis

In [619]:
```
# Remove Emojis
def remove_emoji(string):
    emoji_pattern = re.compile(
          "["
          u"\U0001F600-\U0001F64F" # emoticons
          u"\U0001F300-\U0001F5FF" # symbols & pictographs
          u"\U0001F680-\U0001F6FF" # transport & map symbols
          u"\U0001F1E0-\U0001F1FF" # flags (ios)
          u"\U00002702-\U000027B0"
          u"\U000024C2-\U0001F251"
          "]+",
         flags=re.UNICODE,
)
    return emoji_pattern.sub(r"", string)
```

Remove Punctuations

In [620]:
```
# Remove Punctuation
def remove_punc(text):
    table = str.maketrans("", "", string.punctuation)
    return text.translate(table)
df['text']=df.text.map(lambda X: remove_URL(X))
df['text']=df.text.map(lambda X: remove_html(X))
df['text']=df.text.map(lambda X: remove_emoji(X))
df['text']=df.text.map(lambda X: remove_punc(X))
```

Remove Stop words

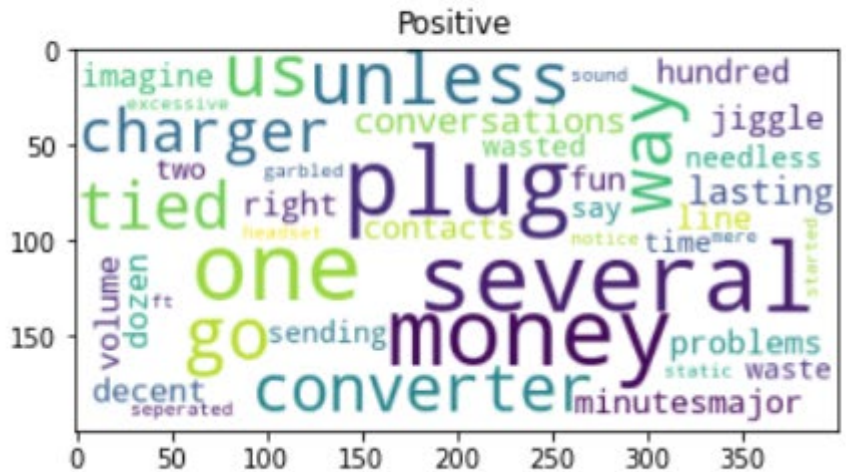
In [621]:
```
# Remove Stopwords
def remove_stopwords(text):
    text = [word.lower() for word in text.split() if word.lower() not in stop]
    return " ".join(text)
df['text'] =df['text'].map(remove_stopwords)
```

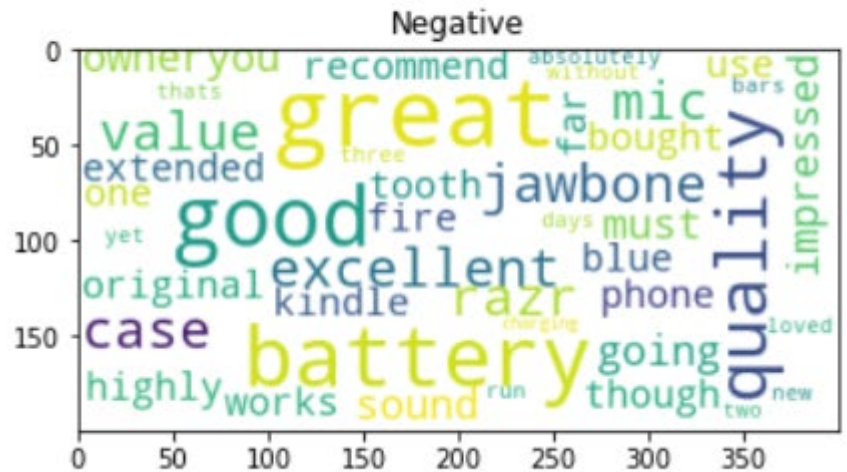
Word Cloud target words

In [622]:
```
# First visualization
from wordcloud import WordCloud
corpus0 = create_corpus(df=df, label=0)
corpus1 = create_corpus(df=df, label=1)
word_cloud0 = WordCloud(background_color='white', max_font_size=50).generate(
    " ".join(corpus1[:50])
)
word_cloud1 = WordCloud(background_color='white', max_font_size=50).generate(
    " ".join(corpus0[:50])
)
```

In [623]:
```
# Visualize our initial words
plt.figure(1, figsize=(12, 8))
plt.subplot(1, 2, 1)
plt.imshow(word_cloud1)
plt.title('Positive')
plt.subplot(1, 2, 2)
plt.imshow(word_cloud0)
plt.title('Negative')
```

Out[623]: Text(0.5, 1.0, 'Negative')





Stemming

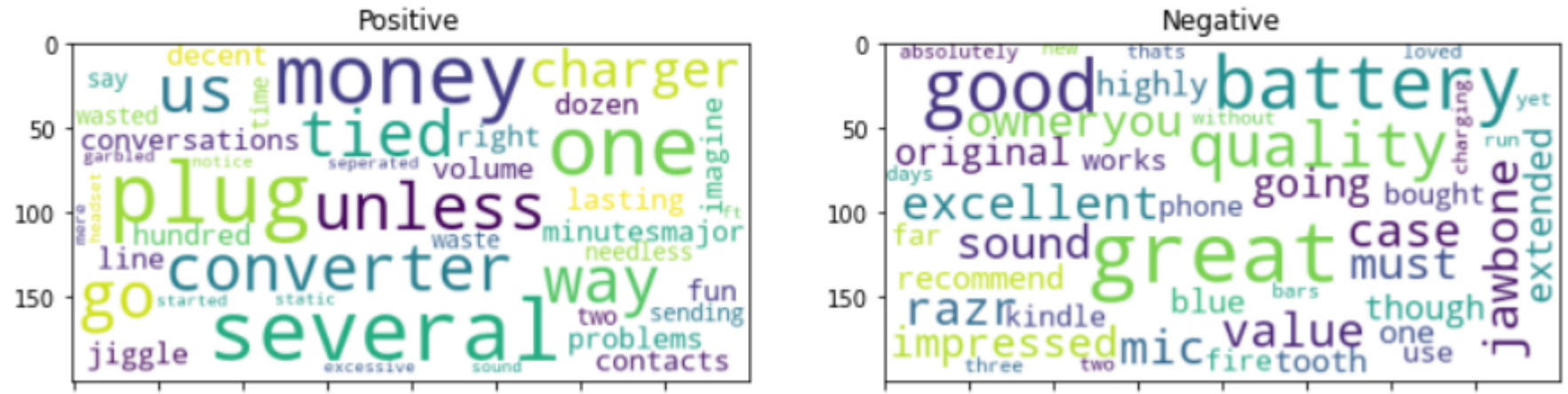
- Is the bundling together of words of the same root.

In [85]:
```
from nltk.stem.porter import PorterStemmer
stemmer = PorterStemmer()

def stemming(text):
    text = [stemmer.stem(word) for word in text.split()]

    return " ".join(text)
```

In [86]:
```
df['text'] = df['text'].map(stemming)
```

In [87]:
```
# Re- create our word cloud after stemming
word_cloud0 = WordCloud(background_color='white', max_font_size=50).generate(
    " ".join(corpus1[:50])
)
word_cloud1 = WordCloud(background_color='white', max_font_size=50).generate(
    " ".join(corpus0[:50])
)
# Visualize our words
plt.figure(1, figsize=(12, 8))
plt.subplot(1, 2, 1)
plt.imshow(word_cloud1)
plt.title('Positive')
plt.subplot(1, 2, 2)
plt.imshow(word_cloud0)
plt.title('Negative')
```

Out[87]: Text(0.5, 1.0, 'Negative')



Tokenization

- " Process of removing sensitive data and placing unique symbols of identification in its place to retain all the essential information, (Simplilearn)".
- It works by breaking words and sentences using space and punctuation as shown in our code below. (https://www.youtube.com/watch?v=7WfoYl-EPtI, Simplilearn, June 3, 2020).

In [626]:
```
# Tokenization
from nltk.tokenize import TreebankWordTokenizer
tokenizer = TreebankWordTokenizer()
df['tokens']= df['text'].map(tokenizer.tokenize)
df[['text', 'tokens']].head(5)
```

Out[626]:

	text	tokens
0	way plug us unless go convert	[way, plug, us, unless, go, convert]
1	good case excel valu	[good, case, excel, valu]
2	great jawbon	[great, jawbon]
3	tie charger convers last 45 minutesmajor problem	[tie, charger, convers, last, 45, minutesmajor...
4	mic great	[mic, great]

Bag of words

- It associates an index to each word in the vocabulary and embeds each sentence as a list of 0s, with a 1 at each index corresponding to a word present in the sentence.

In [627]:
```
# Create our bag of words
def count_vect(data, ngrams=(1, 1)):
    count_vectorizer = CountVectorizer(ngram_range=ngrams)
    emb = count_vectorizer.fit_transform(data)
    return emb, count_vectorizer
df_counts, count_vectorizer = count_vect(df['text'])
df_counts = count_vectorizer.transform(df['text'])
```

In [628]:
```
# Print the output
df.shape
```

Out[628]: (1000, 9)

In [629]:
```
# Vocabulary size
len(df_counts.todense()[0].tolist()[0])
```

Out[629]: 1471

In [630]:
```
# Check words in index 0
df.text.iloc[0]
```

Out[630]:
```
'way plug us unless go convert'
```

In [631]:
```
print(df_counts.todense()[0][0:].sum())
```

```
6
```

In [632]:
```
df_counts, count_vectorizer = count_vect(df['text'], ngrams=(1, 2))
df_counts = count_vectorizer.transform(df['text'])
print(df_counts.todense()[0][0:].sum())
```

```
11
```

Checking words frequency in the text

In [633]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
def tfidf(data, ngrams=(1,1)):
    tfidf_vectorizer = TfidfVectorizer(ngram_range=ngrams)
    df = tfidf_vectorizer.fit_transform(data)
    return df, tfidf_vectorizer
df_tfidf, tfidf_vectorizer = tfidf(df['text'])
df_tfidf = tfidf_vectorizer.transform(df['text'])
```

In [634]:
```
# Check Tfidf Vectorizer for weight given to each word
[x for x in df_tfidf.todense()[0][0:].tolist()[0] if x !=0]
```

Out[634]:
```
[0.4719628881974797,
 0.34953090316309415,
 0.3359495002163575,
 0.44544197767581506,
 0.44544197767581506,
 0.3812872962100649]
```

In [635]:
```
# Results of words frequency using tfidf
print(df_tfidf.todense()[0][0:].sum())
```

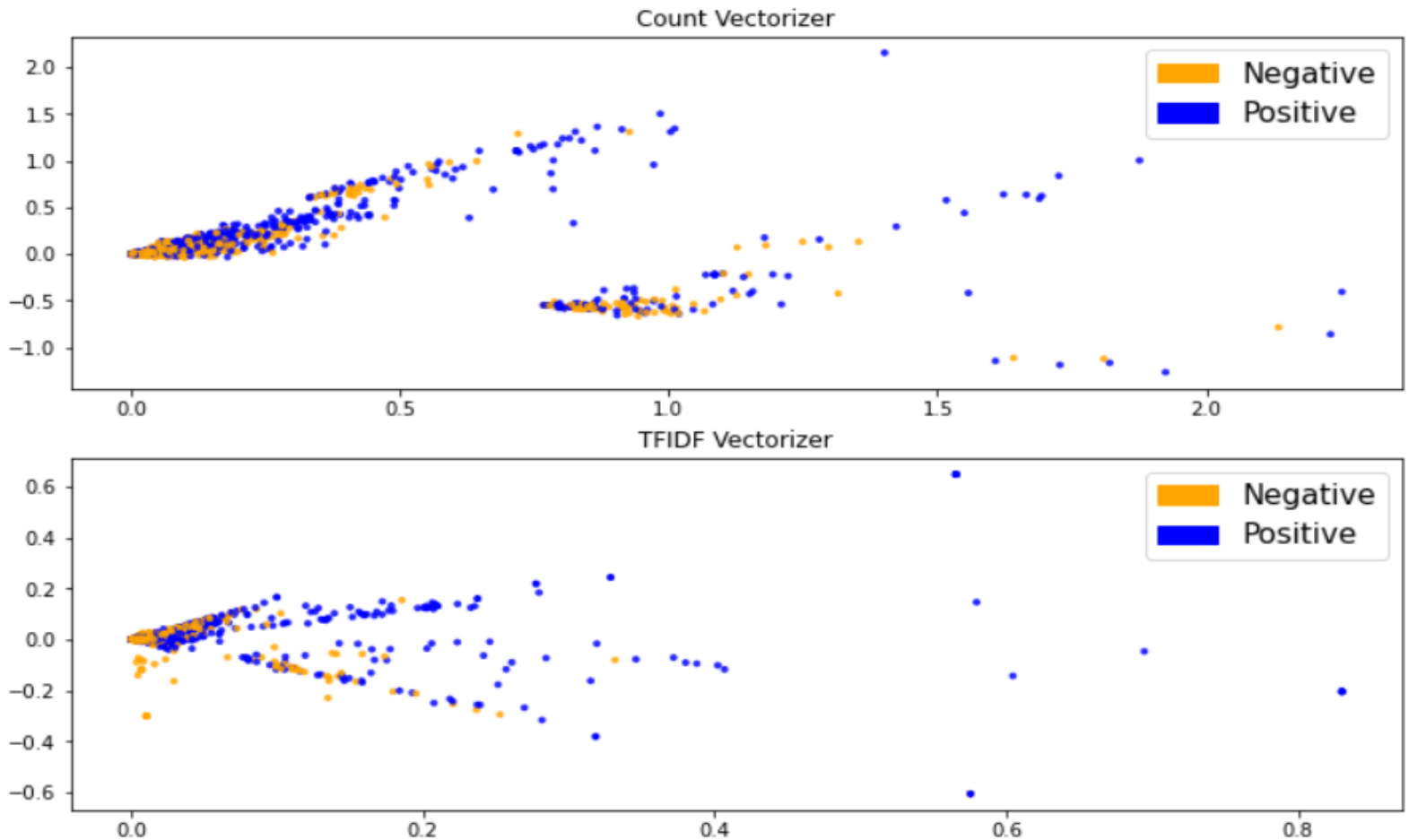
```
2.4296145431386265
```

In [637]:
```
# Visualize how  words embeddings occur frequently.
from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib
import matplotlib.patches as mpatches
X_counts = df_counts
X_tfidf = df_tfidf
y = df['label'].values
```

In [638]:
```
def plot_LSA(df_data, df_labels, plot=True):
    lsa = TruncatedSVD(n_components=2)
    lsa.fit(df_data)
    lsa_scores = lsa.transform(df_data)
    color_mapper = {label: idx for idx, label in enumerate(set(df_labels))}
    color_column = [color_mapper[label] for label in df_labels]
    colors = ['orange', 'blue', 'blue']
    if plot:
        plt.scatter(
            lsa_scores[:, 0],
            lsa_scores[:, 1],
            s=8,
            alpha=0.8,
            c=df_labels,
            cmap=matplotlib.colors.ListedColormap(colors),
     )
        red_patch = mpatches.Patch(color='orange', label='Negative')
        green_patch = mpatches.Patch(color='blue', label='Positive')
        plt.legend(handles=[red_patch, green_patch], prop={'size': 15})
```

```
plt.figure(1, figsize=(12, 8))
plt.subplot(2,1,1)
plot_LSA(X_counts, y)
plt.title('Count Vectorizer')
plt.subplot(2,1,2)
plot_LSA(X_tfidf, y)
plt.title('TFIDF Vectorizer')
```

Out[638]: Text(0.5, 1.0, 'TFIDF Vectorizer')



Unique vocabulary words in the text

In [639]:
```
# Count vocab_size words in our text.
from collections import Counter
# Count unique words in our text
def counter_word(text):
    count = Counter()
    for i in text.values:
        for word in i.split():
            count[word] += 1
    return count
text = df.text
counter = counter_word(text)
```

In [640]:
```
# Print number of unique words.
len(counter)
```

Out[640]: 1480

In [641]:
```
# Print actual words
counter
```

Out[641]:
```
Counter({'way': 7,
         'plug': 15,
         'us': 2,
         'unless': 2,
         'go': 12,
         'convert': 1,
         'good': 75,
         'case': 35,
         'excel': 29,
         'valu': 5,
         'great': 97,
         'jawbon': 3,
         'tie': 1,
         'charger': 22,
         'convers': 5,
         'last': 15,
         '45': 1,
         'minutesmajor': 1,
         'problem': 24,
```

Text Sequencing.

- Turns sentences in our text into data.
- Creates sequence of numbers from our sentences and using tools to process them to make them ready for teaching neural networks.
- Check below slides to see how I created sequence of numbers from our text reviews. (https://www.youtube.com/watch?v=r9QjkdSJZ2g, TensorFlow, February 25, 2020).

In [642]:
```
# Setting Maximum words to use in our model
num_words = len(counter)
# max number of words in a sequence
max_length = 50
```

Splitting data into training and testing set

In [171]:
```
# Split data to Train and Test set
from sklearn.model_selection import train_test_split
# Split the data
train_size = int(df.shape[0] * 0.7)
train_sentences = df.text[:train_size]
train_labels = df.label [:train_size]

test_sentences = df.text[train_size:]
test_labels = df.label[train_size:]
```

In [172]:
```
# Tokenize our data set again
from keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(num_words=num_words)
tokenizer.fit_on_texts(train_sentences)
# index our words
word_index = tokenizer.word_index
```

In [173]:
```
# Train sequences using text in index 0
train_sequences = tokenizer.texts_to_sequences(train_sentences)
train_sequences[0]
```

Out[173]:
```
[91, 49, 322, 323, 84, 499]
```

Padding Process.

- Padding occurs after text sequencing and its main goal is to make our word length to be equal and work well in our model.
- Example, below shows padded text review from index 0.
- (https://www.youtube.com/watch?v=r9QjkdSJZ2g, TensorFlow Feb 25, 2020).

```
In [174]: from keras.preprocessing.sequence import pad_sequences
          train_padded = pad_sequences(
              train_sequences, maxlen=max_length, padding='post', truncating='post'
          )
```

```
In [175]: # Print train_padded for index 0
          train_padded[0]
```

```
Out[175]: array([ 91,  49, 322, 323,  84, 499,   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])
```

In [176]:
```
test_sequences = tokenizer.texts_to_sequences(test_sentences)
test_padded = pad_sequences(
    test_sequences, maxlen=max_length, padding='post', truncating='post'
)
```

In [177]:
```
# Print shape of train and test
print(f'Shape of train{train_padded.shape}')
print(f'Shape of test {test_padded.shape}')
```

```
Shape of train(700, 50)
Shape of test (300, 50)
```

Sentiment category in the text.

Are two categories of sentiments in our text: (Positive and Negative).

1).Positive sentiment - Has more positive words than negative in our review and is labeled as a 1.

2).Negative sentiment- Has more negative words than positive in our review and is labeled as a 0.

Steps to prepare the data for analysis

- Step 1. Import python libraries.
- Step 2. Import data into python notebook environment.
- Step 3. Explore our data.
- Step 4. Clean our data.
- Step 5. Split our data into Test and Train dataset.
- Step 6. Build the model
- Step 7. Evaluate the model

Model Summary

- Image of the model summary from tensor flow.

```
In [158]: # Print our model summary
          model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 50, 16)            23680

 global_average_pooling1d (G  (None, 16)               0
 lobalAveragePooling1D)

 dropout (Dropout)           (None, 16)                0

 dense (Dense)               (None, 1)                 17

=================================================================
Total params: 23,697
Trainable params: 23,697
Non-trainable params: 0
_________________________________________________________________
```

Network Architecture

In [153]:
```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout

embedding_dim = 16
vocab_size = 1480
max_length = 50
model = keras.Sequential([
                layers.Embedding(vocab_size, embedding_dim, input_length= max_length),
                layers.GlobalAveragePooling1D(),
                # half the neurons will be turned off randomly to help with overfitting.
                layers.Dropout(0.5),
                # Binary Classification
                layers.Dense(1, activation='sigmoid')])

# Compile our model
model.compile(loss='binary_crossentropy',optimizer='adam', metrics=['accuracy'])
```

Number of layers in the model architecture.

- I have three dense layers in my model and one output layer.

Input layers

1). 1st layer consist of embedded vocabulary size from our text, defined embedding_dim of 16, maximum length of our input words not more than fifty.

2). Our second layer is "The 1D Global average pooling layer, takes a 2-dimensional tensor of size (input size) x (input channels) and computes the maximum of all the (input size) values for each of the (input channels). Used as an alternative to the flattening block after pooling block of a convolutional Neural network (Peltarion)" . (https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/global-average-pooling-1d, Peltarion, n.d.).

3). Dropout dense layer will be used in training the CNN model to prevent overfitting during training.

Output layer

4). The last layer is an output layer with a sigmoid function, because we will be looking at one input and predicting which of the two possible class it belongs to.

Hyperparameters

- Hyperparameters are used in neural network model to control the learning process.

1). Adam as an optimizer, as a replacement optimization algorithm for stochastic gradient descent for training deep learning model.

2). Sigmoid activation function, to help the model decide whether the input neuron to the network is import or not in the process of prediction and it exists between 0 to 1. Since we will be predicting the probability of the output being binary or (negative or positive).

3). Loss function, to quantify the difference between the expected outcome and the outcome produced by the machine learning model.

4). I also used accuracy metric, as true positives and true negatives is more important in the model outcome.

5). Dropout set at 0.5 to help the model training data not to overfit.

Stopping Criteria

- Used early stopping to help the model to avoid overfitting when training, remove guess work when choosing what is the right epochs number to use during the training process.

Training process.

In [185]:
```
# to minimize overfitting of our model we will use callback as shown below to allow model to choose training set itself.
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor='val_loss', mode='min', verbose=1, patience=20)
```

In [186]:
```
history = model.fit(train_padded,train_labels,  epochs=500, validation_data=(test_padded, test_labels), callbacks=[early_stop])
```

```
Epoch 175/500
22/22 [==============================] - 0s 6ms/step - loss: 0.1373 - accuracy: 0.9886 - val_loss: 0.4784 - val_accuracy: 0.7
767
Epoch 176/500
22/22 [==============================] - 0s 5ms/step - loss: 0.1332 - accuracy: 0.9900 - val_loss: 0.4792 - val_accuracy: 0.7
800
Epoch 177/500
22/22 [==============================] - 0s 4ms/step - loss: 0.1331 - accuracy: 0.9871 - val_loss: 0.4788 - val_accuracy: 0.7
833
Epoch 178/500
22/22 [==============================] - 0s 5ms/step - loss: 0.1311 - accuracy: 0.9900 - val_loss: 0.4795 - val_accuracy: 0.7
800
Epoch 179/500
22/22 [==============================] - 0s 4ms/step - loss: 0.1253 - accuracy: 0.9871 - val_loss: 0.4787 - val_accuracy: 0.7
833
Epoch 180/500
22/22 [==============================] - 0s 3ms/step - loss: 0.1244 - accuracy: 0.9886 - val_loss: 0.4796 - val_accuracy: 0.7
767
Epoch 180: early stopping
```

Evaluating the model fitness.

- To check the fitness of the model, I calculated the model accuracy, model loss, validation accuracy and validation loss to find out if the model is fit for use. Then plotted line graph as shown below.

```
In [187]: model_loss = pd.DataFrame(model.history.history)
```

```
In [188]: model_loss
```

Out[188]:

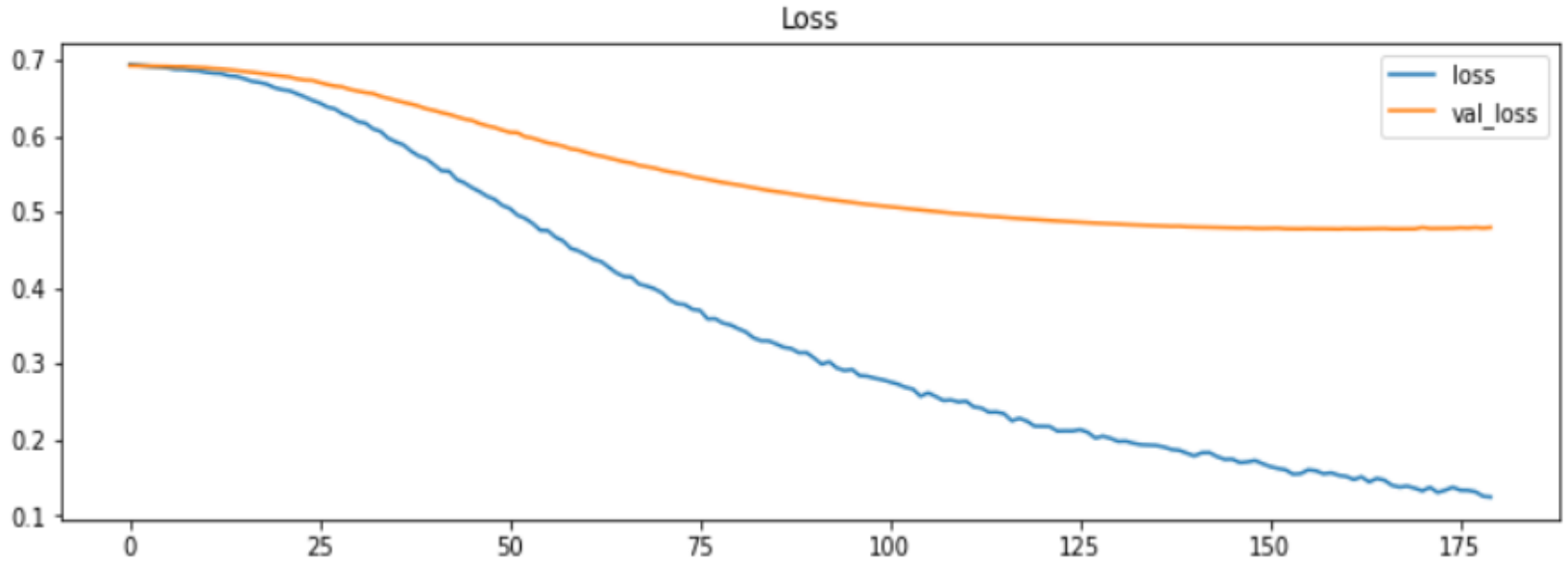
	loss	accuracy	val_loss	val_accuracy
0	0.694167	0.480000	0.692696	0.520000
1	0.693181	0.492857	0.692540	0.543333
2	0.692008	0.524286	0.692632	0.480000
3	0.691403	0.528571	0.692462	0.480000
4	0.690873	0.554286	0.692069	0.480000
...	...	...	...	...
175	0.133249	0.990000	0.479199	0.780000
176	0.133103	0.987143	0.478783	0.783333
177	0.131141	0.990000	0.479523	0.780000
178	0.125318	0.987143	0.478734	0.783333
179	0.124362	0.988571	0.479568	0.776667

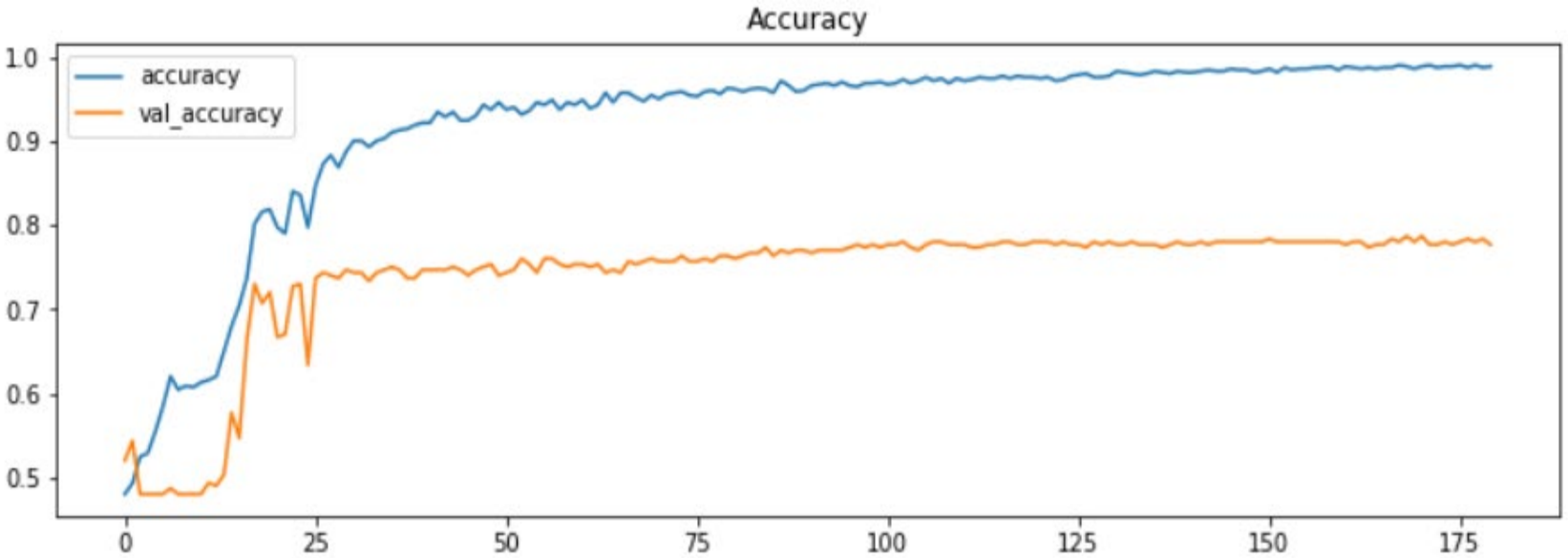
180 rows × 4 columns

In [168]:

```
# evaluate the model
from matplotlib import pyplot
_, train_acc = model.evaluate(train_padded, train_labels, verbose=0)
_, test_acc = model.evaluate(test_padded, test_labels, verbose=0)
print('Train_accuracy: %.3f, Test_accuracy: %.3f' % (train_acc, test_acc))
# plot loss during training
plt.figure(1, figsize=(12, 8))
pyplot.subplot(211)
pyplot.title('Loss')
pyplot.plot(history.history['loss'], label='loss') # train data
pyplot.plot(history.history['val_loss'], label='val_loss') # test data
pyplot.legend()

# plot accuracy during training
pyplot.subplot(212)
pyplot.title('Accuracy')
pyplot.plot(history.history['accuracy'], label='accuracy') # trained data
pyplot.plot(history.history['val_accuracy'], label='val_accuracy') # tested data
pyplot.legend()
pyplot.show()
```

```
Train_accuracy: 0.990, Test_accuracy: 0.777
```





Model_ loss interpretation.

- Large separation between our loss and val_loss shows that our model is overfitting. Means works great on training data and not great on the test data. Which means the data we are using is not the right one to solve our problem. (https://www.youtube.com/watch?v=p3CcfIjycBA , DigitalSreeni, Sep 1, 2020).

Accuracy of the model

- Although, our training accuracy is 0.990 and test accuracy is 0.777. The gap between training accuracy and testing accuracy is large, a clear indication of overfitting in our model. Therefore, we can conclude our model is able to predict 77.7% accurately.

(https://towardsdatascience.com/useful-plots-to-diagnose-your-neural-network-521907fa2f45, George V Jose, Oct 2, 2019).

Saving the trained network within the neural network.

In [191]:
```
# Save my trained model
import pickle
```

In [192]:
```
with open('model_pickle', 'wb')as m:
    pickle.dump(model,m)
```

```
INFO:tensorflow:Assets written to: ram://263b1e3f-c971-4c99-8635-25cc22defdf8/assets
```

Model Functionality

- We will use the lime library to help us predict our text reviews whether positive or negative..
- Lime creates permutations of our data instances and collects the model predictions.
- It then gives weights to the new created text or review based on how closely they match the data of the original prediction.
- Also trains a new less complex, interpretable model on the data variations using the weights to each variation.

In [656]:
```
from lime.lime_text import LimeTextExplainer
```

In [657]:
```
# Our sentiment category
explainer = LimeTextExplainer(class_names=['negative', 'positive'])
```

In [195]:
```
# Create prediction function
def new_predict(texts):
    _seq = tokenizer.texts_to_sequences(texts)
    _text_data = pad_sequences(
        _seq, maxlen=max_length, padding="post", truncating="post"
    )
    return np.array([[float(1 -x), float(x)] for x in model.predict(_text_data)])
```

In [196]:
```
test_sentences.iloc[5]
```

Out[196]:
```
'love cabl allow connect miniusb devic pc'
```

In [197]:
```
test_padded[5]
```

Out[197]:
```
array([ 29, 215, 794, 274,  61, 333,   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])
```

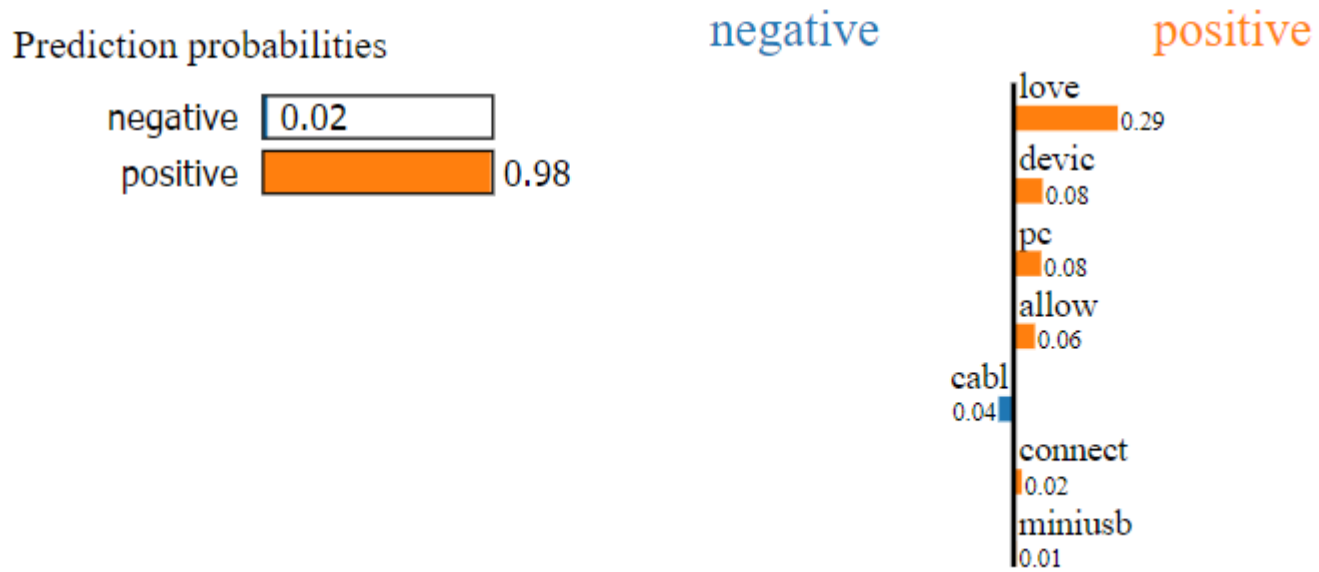
In [198]:
```
# Let us predict
y_pred =model.predict(test_padded[5].reshape(1, -1))
y_pred
```

Out[198]:
```
array([[0.9801453]], dtype=float32)
```

Visualized model prediction weight for each word in the review.

In [199]:
```
# Pass in our text to predict sentiment of review number 5 in our index.
exp = explainer.explain_instance(
    test_sentences.iloc[5], new_predict, num_features=max_length, top_labels=1
)
```

In [200]:
```
# Let us now visualize our prediction sentiment for review number 5
exp.show_in_notebook(text=False)
```



Recommendation

- Based on our predictions results, we can conclude that our model is very good at predicting our customers sentiment words. Whether negative or positive in our reviews.