**IST 664**

**Natural Language Processing**

**Homework 2**

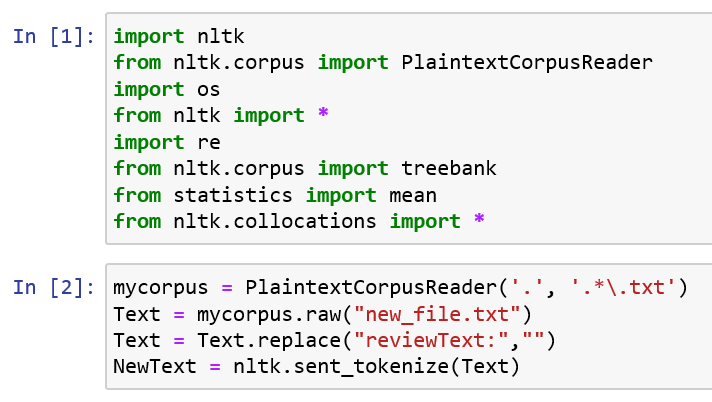
Dhwani Rekhang Gandhi

[dgandhi@syr.edu](mailto:dgandhi@syr.edu)

Table of Contents

1. Data Pre-Processing
2. Data Processing
   1. Task 1
   2. Task 2
3. Descriptive Statistics
4. Frequency Analysis
   1. Question sentences with Adjective phrase(s)
      1. Task 1
      2. Task 2
      3. Task 3
   2. Imperative sentences with Adjective phrase(s)
      1. Task 1
      2. Task 2
      3. Task 3
5. Sentiment Analysis
6. **Data Pre-Processing**

We have used the reviews that were compiled in homework 1 for further analysis.



In the second block I have used the PlaintextCorpusReader to access the new\_file.txt text file. This file is now treated as any regular corpus. The new\_file\_txt has the word reviewText before each review so I have used the replace function to replace all the “reviewText:” with no value/character. Once the reviews are compiled as a corpus, as one string, the string is split into sentences using the sent\_tokenize method present in the nltk module as shown above. The NewText variable now contains a list of sentences.

1. **Data Processing**
   1. **Task 1: Question sentences having an adjective phrase(s)**



In the first block, I have used regular expression to find all those sentences having a question mark at the end. I have filtered those sentences and stored them in a new variable called question\_sent.

In the second block, I have used the treebank module to get the tags. If any word is not present in the treebank it is given a default tag of NN i.e. a noun tag. The nltk.UnigramTagger finds the mostly likely tag for each word. A nltk.BigramTagger chooses a token's tag based on its word string and on the preceding words' tag. It uses the UnigramTagger tuples as a backoff i.e. in particular, a tuple consisting of the previous tag and the word is looked up in a table, and the corresponding tag is returned. This is stored in t2.

In the third block, we first split the question\_sent into tokens and store them in tokentext. After this, the tags from t2 are used to tag the tokens created and a tuple is created of the token and the tag which is stored in taggedtext.

In the next block, I have first defined the rules for an adjective phrase as follows:

1. Adverb + Adjective
2. Determinant + Adjective
3. Adjective + Noun
4. Adjective +Pronoun
5. Wh(Determiner/Pronoun/Adverb) + Adjective

These rules are given to the label JJP which is then given as a parameter to the nltk.Regexparser command. We then parse the question sentences one by one and check if any of the rules defined are present in the sentence. If the rule is present, we filter out those sentences and store them in adj\_question\_sent array.

In the last block we convert all the tuples of the tokens and the tags back to sentences so that we can uses these adjective, question sentences for further analysis.

* 1. **Task 2: Imperative sentences starting with a verb, having an exclamation and an adjective phrase(s)**



In the first block, I have used regular expression to find all those sentences having an exclamation mark at the end. I have filtered those sentences and stored them in a new variable called exclamation\_sent.

In the second block, we first split the exclamation\_sent into tokens and store them in tokentext1. After this, the tags from t2 defined earlier are used to tag the tokens created and a tuple is created of the token and the tag which is stored in taggedtext1.

In the third block, I have first defined the rules for the sentence to begin with a verb phrase. The sentence either begins with a verb or a determinant followed by a verb. These rules are given to the label VP which is then given as a parameter to the nltk.Regexparser command. We then parse the exclamation sentences one by one and check if any of the rules defined are present in the sentence. If the rule is present, we filter out those sentences and store them in imperative\_sent array.

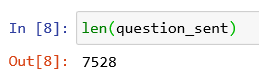
In the next block, I have first defined the rules for an adjective phrase as follows:

1. Adverb + Adjective
2. Adjective + Conjunction
3. Determinant + Adjective
4. Adjective +Pronoun

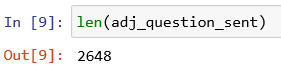
These rules are given to the label ADJ\_VP which is then given as a parameter to the nltk.Regexparser command. We then parse the imperative, exclamation sentences one by one and check if any of the rules defined are present in the sentence. If the rule is present, we filter out those sentences and store them in adj\_imperative\_sent array.

In the last block we convert all the tuples of the tokens and the tags back to sentences so that we can uses these adjective, imperative sentences for further analysis.

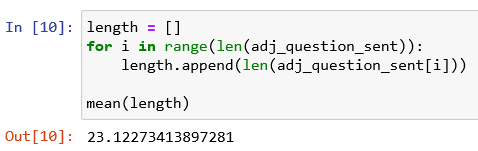
1. **Descriptive Statistics**
2. Number of question sentences:



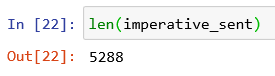
1. Number of question sentences having an adjective phrase(s):



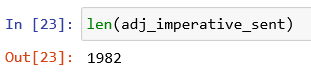
1. Average length of question sentences having an adjective phrase(s):



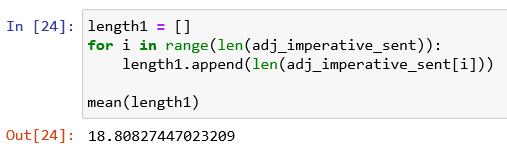
1. Number of imperative sentences (i.e sentences starting with a verb phrase and ending with an exclamation mark):



1. Number of imperative sentences having an adjective phrase(s):



1. Average length of imperative sentences having an adjective phrase(s):

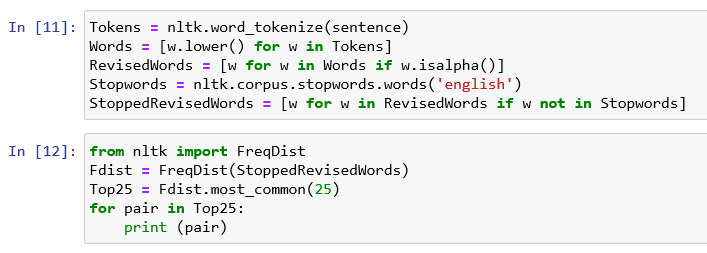


|  |  |
| --- | --- |
|  | **Count** |
| Question sentences | **7528** |
| Question sentences with Adjective phrase | **2648** |
| Average length of Question sentences with Adjective phrase | **23.12** |
| Imperative sentences | **5288** |
| Imperative sentences with Adjective phrase | **1982** |
| Average length of Imperative sentences with Adjective phrase | **18.81** |

1. **Frequency Analysis**

**4.1 Question Sentences with Adjective phrase(s):**

* + 1. **Task 1: Top 25 words by frequency**



Firstly, I tokenized the adjective, question sentences, after which I converted all the words into lower-case. I did not want the upper-case words and the lower-case words to be counted differently. I wanted the totality of each word, so I converted all of them into lower-case. I also filtered all the special characters and the stop words since those are mainly for giving a sense to the text grammatically. They do not hold much importance when we want a count of words to understand the sentences briefly. I did this to get a general idea on which words are most frequently used in these sentences. I realized the result as below:

('would', 259)

('like', 249)

('one', 199)

('size', 197)

('good', 194)

('maybe', 180)

('could', 163)

('get', 159)

('wear', 146)

('really', 138)

('pair', 132)

('fit', 128)

('watch', 122)

('little', 122)

('price', 120)

('know', 120)

('much', 119)

('want', 115)

('right', 113)

('say', 112)

('great', 109)

('shoe', 104)

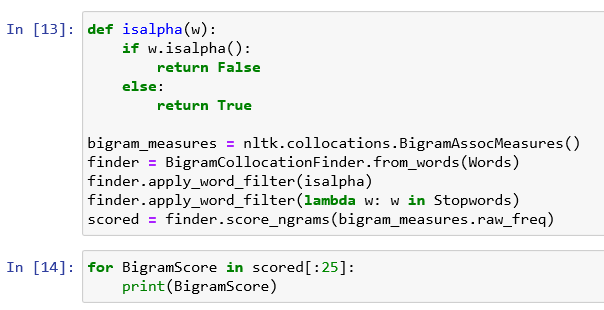
('comfortable', 102)

('shoes', 101)

('look', 100)

The words indicate that the people are probably asking about the size, fit and the price of the watch and shoes.

* + 1. **Task 2: Top 25 bigrams by frequency**



In the first block, a function isalpha is defined which is used as a special character filter for the bigrams. In the second block the NLTK package is used to create a finder consisting of all the bigrams in the original lower case, tokenized words. I did not want to differentiate between the upper case and the lower-case bigrams and hence used the previous lower-case words. Using a similar approach as before, the bigrams where if either of them is a special character or a stop word is removed. Only those bigrams consisting of lower-case alphabets are used for the analysis. After the filtering is done a raw frequency score is calculated for each bigram present and are then stored. The top 25 bigrams with the highest score are then displayed. These are those bigrams that have appeared together the greatest number of times. The results achieved are as below:

(('look', 'like'), 0.0002612671456564337)

(('looks', 'like'), 0.0002286087524493795)

(('many', 'times'), 0.0002286087524493795)

(('great', 'price'), 0.0002122795558458524)

(('another', 'pair'), 0.00019595035924232528)

(('good', 'quality'), 0.00017962116263879816)

(('good', 'thing'), 0.00016329196603527107)

(('high', 'quality'), 0.00016329196603527107)

(('looks', 'great'), 0.00016329196603527107)

(('feel', 'like'), 0.00014696276943174395)

(('first', 'pair'), 0.00013063357282821686)

(('first', 'time'), 0.00013063357282821686)

(('larger', 'size'), 0.00013063357282821686)

(('smaller', 'size'), 0.00013063357282821686)

(('wrong', 'size'), 0.00013063357282821686)

(('every', 'time'), 0.00011430437622468974)

(('gon', 'na'), 0.00011430437622468974)

(('good', 'price'), 0.00011430437622468974)

(('high', 'end'), 0.00011430437622468974)

(('light', 'weight'), 0.00011430437622468974)

(('long', 'time'), 0.00011430437622468974)

(('shoe', 'size'), 0.00011430437622468974)

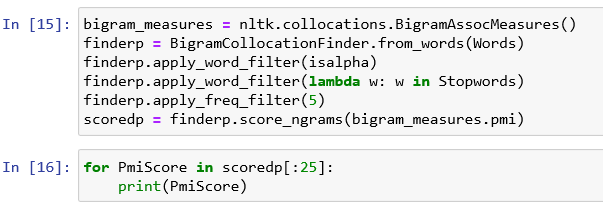
(('two', 'pairs'), 0.00011430437622468974)

(('well', 'made'), 0.00011430437622468974)

(('another', 'one'), 9.797517962116264e-05)

The bigrams indicate that maximum times the customers want to know what a particular product looks like and the quality of the product.

* + 1. **Task 3: Top 25 bigrams by their mutual information scores (using minimum frequency 5)**



The same approach is used for calculating the mutual information score like calculating the raw frequency. The lower-case tokenized words are used to create a finder consisting of bigrams using the NLTK package. Two filters are applied to remove the stop words and the special character. In addition to these two filters another filter is applied which defines that the min frequency of the bigram in the text should be 5. All these bigrams are then stored and displayed. The top 25 bigrams by their mutual information score are displayed. The score displayed is the measure of mutual dependence of the two words on each other. Once the top 25 bigrams are printed the results obtained are as below:

(('gon', 'na'), 12.73226166094484)

(('midway', 'briefs'), 12.442755043749854)

(('bathing', 'suit'), 12.17972063791606)

(('customer', 'service'), 12.17972063791606)

(('flip', 'flops'), 12.165221068220944)

(('arch', 'support'), 10.41675983521691)

(('free', 'shipping'), 9.99529606677863)

(('cold', 'water'), 9.82537106533632)

(('high', 'end'), 8.978222553419693)

(('light', 'weight'), 8.864051533500382)

(('ever', 'seen'), 8.857792543028697)

(('many', 'times'), 8.46280098595161)

(('two', 'pairs'), 8.458059173824545)

(('real', 'deal'), 8.166123034515337)

(('left', 'foot'), 8.143297229666187)

(('best', 'part'), 8.088405471170113)

(('anyone', 'else'), 7.540242888651909)

(('running', 'shoe'), 7.523675039133421)

(('every', 'time'), 7.379745246224054)

(('different', 'sizes'), 7.2493416893851705)

(('high', 'quality'), 7.222706562881704)

(('second', 'pair'), 7.1948275303062665)

(('low', 'price'), 7.12082694886249)

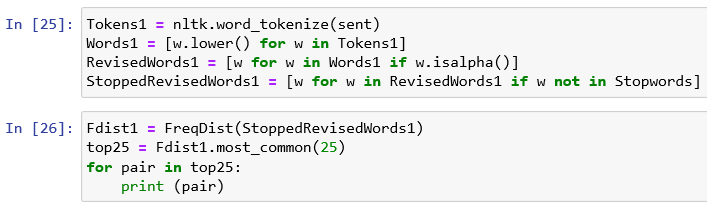
(('another', 'pair'), 6.888166192072214)

(('full', 'size'), 6.695172342209618)

These bigrams indicate those words which appear together majority of the times in the adjective, question sentences. Like midway briefs, bathing suits, customer service and flip flops are those bigrams that appear together in the question sentences.

**4.2 Imperative Sentences with Adjective phrase(s):**

* + 1. **Task 1: Top 25 words by frequency**



Firstly, I tokenized the adjective, imperative sentences, after which I converted all the words into lower-case. I also filtered all the special characters and the stop words. They do not hold much importance when we want a count of words to understand the sentences briefly. I did this to get a general idea on which words are most frequently used in these sentences. I realized the result as below:

('great', 415)

('comfortable', 288)

('cute', 183)

('nice', 165)

('ever', 157)

('really', 152)

('good', 148)

('best', 136)

('one', 120)

('like', 120)

('shoes', 119)

('pair', 118)

('price', 118)

('beautiful', 110)

('wear', 106)

('buy', 101)

('fit', 97)

('size', 96)

('love', 92)

('quality', 88)

('first', 85)

('well', 83)

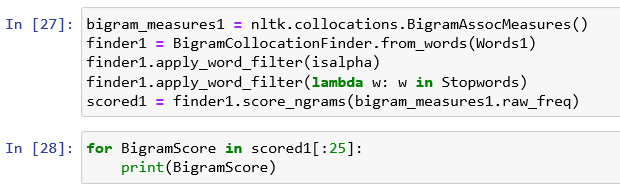
('look', 78)

('time', 74)

('bought', 72)

These unigrams indicate that the imperative sentences majorly contain the words great, comfortable, cute and nice indicating that the customers are generally very satisfied with the products.

* + 1. **Task 2: Top 25 bigrams by frequency**



In the first block, the NLTK package is used to create a finder consisting of all the bigrams in the original lower case, tokenized words. Using a similar approach as before, the bigrams where if either of them is a special character or a stop word is removed. After the filtering is done a raw frequency score is calculated for each bigram present and are then stored. The top 25 bigrams with the highest score are then displayed. These are those bigrams that have appeared together the greatest number of times. The results achieved are as below:

(('make', 'sure'), 0.0008314781535820615)

(('really', 'nice'), 0.0008314781535820615)

(('ever', 'owned'), 0.0007510125258160555)

(('ever', 'worn'), 0.0005632593943620417)

(('well', 'made'), 0.0005632593943620417)

(('great', 'buy'), 0.0005364375184400397)

(('great', 'price'), 0.0005364375184400397)

(('really', 'cute'), 0.0005364375184400397)

(('first', 'pair'), 0.00048279376659603575)

(('good', 'quality'), 0.00048279376659603575)

(('comfortable', 'shoes'), 0.0004559718906740337)

(('flip', 'flops'), 0.0004559718906740337)

(('first', 'time'), 0.00042915001475203173)

(('super', 'cute'), 0.00042915001475203173)

(('second', 'pair'), 0.00037550626290802777)

(('comfortable', 'shoe'), 0.0003218625110640238)

(('great', 'purchase'), 0.0003218625110640238)

(('high', 'quality'), 0.0003218625110640238)

(('really', 'great'), 0.0003218625110640238)

(('feel', 'like'), 0.0002950406351420218)

(('great', 'quality'), 0.0002950406351420218)

(('look', 'great'), 0.0002950406351420218)

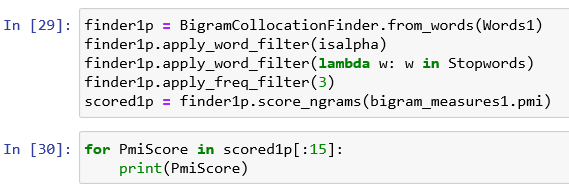
(('looks', 'great'), 0.0002950406351420218)

(('extremely', 'comfortable'), 0.00026821875922001983)

(('great', 'deal'), 0.00026821875922001983)

These bigrams are also on similar positive lines indicating that the customers are greatly happy with the products especially footwear.

* + 1. **Task 3: Top 25 bigrams by their mutual information scores (using minimum frequency 3)**



The same approach is used for calculating the mutual information score like calculating the raw frequency. The lower-case tokenized words are used to create a finder consisting of bigrams using the NLTK package. Two filters are applied to remove the stop words and the special character. In addition to these two filters another filter is applied which defines that the min frequency of the bigram in the text should be 3. All these bigrams are then stored and displayed. The top 15 bigrams by their mutual information score are displayed. The score displayed is the measure of mutual dependence of the two words on each other. Once the top 15 bigrams are printed the results obtained are as below: (We see that very few bigrams are present that have occurred at least 3 times in the sentences)

(('stainless', 'steel'), 13.601267830853399)

(('reasonably', 'priced'), 12.601267830853399)

(('swim', 'suit'), 12.186230331574553)

(('previous', 'reviews'), 10.864302236687191)

(('flip', 'flops'), 10.777838145937725)

(('jewelry', 'cleaner'), 10.77683939543685)

(('nursing', 'bras'), 10.156482988180501)

(('highly', 'recommend'), 10.02523845490225)

(('arch', 'support'), 10.016305330132244)

(('new', 'balance'), 9.900828112712304)

(('several', 'times'), 9.864302236687191)

(('rain', 'boots'), 9.424679099130074)

(('must', 'say'), 9.201337223964764)

(('year', 'old'), 9.063833700214827)

(('light', 'weight'), 9.006321241559618)

Firstly, we see that very few bigrams are present in the imperative sentences that are used more than three times. This is majorly because the number of imperative sentences found in the reviews are less. These bigrams are the ones that interchangeably used with each other in the sentences.

1. **Sentiment Analysis**

For sentiment analysis we can first find words that are positive and negative and we can then give them a score of +1 and -1 further analysis we can even create trigrams that can give a much better insight than the unigrams or the bigrams. For better visualization we can even create word clouds. These word clouds are not only appealing to the eye, but it gives us a general picture about the text within seconds. We should also take care of negative words when used in a sentence. Words like ‘not’, ‘didn’t’ negate the whole sentence. So words like ‘good’, ‘like’ should be converted to ‘not\_good’, ‘not\_like’ to get an accurate answer.