Keyphrase Extraction

All the code examples showcased in this chapter are available on the book's official GitHub repository which you can access here: https://github.com/dipanjanS/text-analytics-with-python/tree/master/New-Second-Edition.

Keyphrase Extraction

This is one of the most simple yet extremely powerful techniques of extracting important information from unstructured text documents. Keyphrase extraction, also known as terminology extraction is defined as the process or technique of extracting key important and relevant terms or phrases from a body of unstructured text such that the core topics or themes of the text document(s) are captured in these key phrases. This technique falls under the broad umbrella of information retrieval and extraction. Keyphrase extraction finds its uses in many areas some of which are mentioned below.

- Semantic web
- Query based search engines and crawlers
- Recommendation systems
- Tagging systems
- Document similarity
- Translation

Keyphrase extraction is often the starting point for carrying out more complex tasks in text analytics or natural language processing and the output from this can itself act as features for more complex systems. There are various approaches for keyphrase extraction and we will be covering the following two major techniques.

- **Collocations**
- **Weighted Tag-based Phrase Extraction**

An important thing to remember here is that we will be extracting phrases which are usually a collection of words and can sometimes include even single words. If you are extracting keywords, that is also known as keyword extraction and it is a subset of keyphrase extraction.

Collocations

The term collocation is actually a concept borrowed from analyzing corpora and linguistics. A collocation can be defined as a sequence or group of words which tend to occur frequently such that this frequency tends to be more than what could be termed as a random or chance occurrence. Various types of collocations can be formed based on the parts of speech of the various terms like nouns, verbs and so on. There are various ways to extract collocations and one of the best ways to do that is to use an n-gram grouping or segmentation approach where we construct n-grams out of a corpus and then counting the frequency of each n-gram and ranking them based on their frequency of occurrence to get the most frequent n-gram collocations.

The idea is to have a corpus of documents which could be paragraphs or sentences, tokenize them to form sentences, flatten the list of sentences to form one large sentence or string over which we slide a window of size **n** based on the n-gram range and compute n-grams across the string. Once computed, we count each n-gram based on its frequency of occurrence and then rank them based on their frequency. This yields the most frequent collocations on the basis of frequency. We will implement this from scratch initially so that you can understand the algorithm better and then we will use some of nltk's inbuilt capabilities to depict the same.

Let' start by first loading some necessary dependencies and a corpus on which we will be computing collocations. We will use the nltk Gutenberg corpus's book, Lewis Carroll's Alice in Wonderland as our corpus. We also normalize the corpus to standardize the text content using our handy text_normalizer module which we built and also used in the previous chapters.

```
from nltk.corpus import gutenberg
import text_normalizer as tn
import nltk
from operator import itemgetter

# load corpus
alice = gutenberg.sents(fileids='carroll-alice.txt')
alice = [' '.join(ts) for ts in alice]
norm_alice = list(filter(None,
                         tn.normalize_corpus(alice, text_lemmatization=False)))

# print and compare first line
print(alice[0], '\n', norm_alice[0])
```

```
[ Alice ' s Adventures in Wonderland by Lewis Carroll 1865 ]
 alice adventures wonderland lewis carroll
```

Now we will define a function to compute n-grams based on some input list of tokens and the parameter n which determines the degree of the n-gram like a unigram, bigram and so on. The following code snippet computes n-grams for an input sequence.

```
def compute_ngrams(sequence, n):
    return list(
```

```
            zip(*(sequence[index:]
                    for index in range(n)))
    )
```

This function basically takes in a sequence of tokens and computes a list of lists having sequences where each list contains all items from the previous list except the first item removed from the previous list. It constructs n such lists and then zips them all together to give us the necessary n-grams. We wrap the final result in a list since in Python 3; zip gives us a generator object and not a raw list. We can see the function in action on a sample sequence in the following snippet.

```
In [7]: compute_ngrams([1,2,3,4], 2)
Out[7]: [(1, 2), (2, 3), (3, 4)]

In [8]: compute_ngrams([1,2,3,4], 3)
Out[8]: [(1, 2, 3), (2, 3, 4)]
```

The preceding output shows bigrams and trigrams for an input sequence. We will now utilize this function and build upon it to generate the top n-grams based on their frequency of occurrence. For this, we first need to define a function to flatten the corpus into one big string of text. The following function will help us achieve the same for a corpus of documents.

```
def flatten_corpus(corpus):
    return ' '.join([document.strip()
                     for document in corpus])
```

We can now build a function which should help us in getting the top n-grams from a corpus of text.

```
def get_top_ngrams(corpus, ngram_val=1, limit=5):

    corpus = flatten_corpus(corpus)
    tokens = nltk.word_tokenize(corpus)

    ngrams = compute_ngrams(tokens, ngram_val)
    ngrams_freq_dist = nltk.FreqDist(ngrams)
    sorted_ngrams_fd = sorted(ngrams_freq_dist.items(),
                              key=itemgetter(1), reverse=True)
    sorted_ngrams = sorted_ngrams_fd[0:limit]
    sorted_ngrams = [(' '.join(text), freq)
                     for text, freq in sorted_ngrams]

    return sorted_ngrams
```

We make use of nltk's FreqDist class to create a counter of all the n-grams based on their frequency and then we sort them based on their frequency and return the top n-grams based on the specified user limit. We will now compute the top bigrams and trigrams on our corpus using the following code snippet.

```
# top 10 bigrams
```

```
In [11]: get_top_ngrams(corpus=norm_alice, ngram_val=2,
     ...:               limit=10)
Out[11]:
[('said alice', 123),
 ('mock turtle', 56),
 ('march hare', 31),
 ('said king', 29),
 ('thought alice', 26),
 ('white rabbit', 22),
 ('said hatter', 22),
 ('said mock', 20),
 ('said caterpillar', 18),
 ('said gryphon', 18)]

# top 10 trigrams
In [12]: get_top_ngrams(corpus=norm_alice, ngram_val=3,
     ...:               limit=10)
Out[12]:
[('said mock turtle', 20),
 ('said march hare', 10),
 ('poor little thing', 6),
 ('little golden key', 5),
 ('certainly said alice', 5),
 ('white kid gloves', 5),
 ('march hare said', 5),
 ('mock turtle said', 5),
 ('know said alice', 4),
 ('might well say', 4)]
```

The above output shows us sequences of two and three words generated by n-grams along with the number of times they occur throughout the corpus. We can see most of the collocations point to people who are speaking something as *"said <person>".* We also see the people who are popular characters in *"Alice in Wonderland"* like the *mock turtle*, the *king*, the *rabbit*, the *hatter* and of course *Alice* herself being depicted in the above collocations.

We will now look at nltk's collocation finders which enable us to find collocations using various measures like raw frequencies, pointwise mutual information and so on. Just to explain briefly, ***pointwise mutual information*** can be computed for two events or terms as the logarithm of the ratio of the probability of them occurring together by the product of their individual probabilities assuming that they are independent of each other. Mathematically we can represent it as,

$$pmi(x,y) = log\ \frac{p(x,y)}{p(x)p(y)}$$

and this measure is symmetric. The following code snippet shows us how to compute these collocations using these measures.

```
# bigrams
from nltk.collocations import BigramCollocationFinder
from nltk.collocations import BigramAssocMeasures
```

```
finder = BigramCollocationFinder.from_documents([item.split()
                                                 for item
                                                 in norm_alice])
finder

<nltk.collocations.BigramCollocationFinder at 0x1c2c2c4f358>


# raw frequencies
In [14]: finder.nbest(bigram_measures.raw_freq, 10)
Out[14]:
[(u'said', u'alice'),
 (u'mock', u'turtle'),
 (u'march', u'hare'),
 (u'said', u'king'),
 (u'thought', u'alice'),
 (u'said', u'hatter'),
 (u'white', u'rabbit'),
 (u'said', u'mock'),
 (u'said', u'caterpillar'),
 (u'said', u'gryphon')]

# pointwise mutual information
In [15]: finder.nbest(bigram_measures.pmi, 10)
Out[15]:
[(u'abide', u'figures'),
 (u'acceptance', u'elegant'),
 (u'accounting', u'tastes'),
 (u'accustomed', u'usurpation'),
 (u'act', u'crawling'),
 (u'adjourn', u'immediate'),
 (u'adoption', u'energetic'),
 (u'affair', u'trusts'),
 (u'agony', u'terror'),
 (u'alarmed', u'proposal')]


# trigrams
from nltk.collocations import TrigramCollocationFinder
from nltk.collocations import TrigramAssocMeasures

finder = TrigramCollocationFinder.from_documents([item.split()
                                                  for item
                                                  in norm_alice])
trigram_measures = TrigramAssocMeasures()

# raw frequencies
In [17]: finder.nbest(trigram_measures.raw_freq, 10)
Out[17]:
[(u'said', u'mock', u'turtle'),
 (u'said', u'march', u'hare'),
```

```
 (u'poor', u'little', u'thing'),
 (u'little', u'golden', u'key'),
 (u'march', u'hare', u'said'),
 (u'mock', u'turtle', u'said'),
 (u'white', u'kid', u'gloves'),
 (u'beau', u'ootiful', u'soo'),
 (u'certainly', u'said', u'alice'),
 (u'might', u'well', u'say')]

# pointwise mutual information
In [18]: finder.nbest(trigram_measures.pmi, 10)
Out[18]:
[(u'accustomed', u'usurpation', u'conquest'),
 (u'adjourn', u'immediate', u'adoption'),
 (u'adoption', u'energetic', u'remedies'),
 (u'ancient', u'modern', u'seaography'),
 (u'apple', u'roast', u'turkey'),
 (u'arithmetic', u'ambition', u'distraction'),
 (u'brother', u'latin', u'grammar'),
 (u'canvas', u'bag', u'tied'),
 (u'cherry', u'tart', u'custard'),
 (u'circle', u'exact', u'shape')]
```

Now you know how to compute collocations for a corpus using an n-gram generative approach. We will now look at a better way of generating key phrases based on parts of speech tagging and term weighing in the next section.

Weighted Tag-based Phrase Extraction

We will now look at a slightly different approach to extracting keyphrases. This method borrows concepts from a couple of papers, namely K. Barker and N. Cornachhia. *"Using Noun Phrase Heads to Extract Document Keyphrases"* and Ian Witten et al. *"KEA: Practical Automatic Keyphrase Extraction"* which you can refer to if you are more interested in further details on their experimentations and approaches. We follow a two-step process in our algorithm here. These steps are mentioned as follows.

- Extract all noun phrase chunks using shallow parsing
- Compute TF-IDF weights for each chunk and return the top weighted phrases

For the first step, we will use a simple pattern based on parts of speech (POS) tags to extract noun phrase chunks. You will be familiar with this from Chapter 3 where we explored chunking and shallow parsing in details. Before discussing our algorithm, let us load up the corpus on which we will be testing our implementation. We use a sample description of elephants taken from Wikipedia available in the file titled 'elephants.txt' which you can obtain from the GitHub repository for this book at https://github.com/dipanjanS/text-analytics-with-python.

```
data = open('elephants.txt', 'r+').readlines()
```

```
sentences = nltk.sent_tokenize(data[0])
len(sentences)
```

```
29
```

```
# viewing the first three lines
sentences[:3]
```

```
['Elephants are large mammals of the family Elephantidae and the order
Proboscidea.',
 'Three species are currently recognised: the African bush elephant (Loxodonta
africana), the African forest elephant (L. cyclotis), and the Asian elephant
(Elephas maximus).',
 'Elephants are scattered throughout sub-Saharan Africa, South Asia, and
Southeast Asia.']
```

Let's now use our nifty text_normalizer module now to do some very basic text pre-processing on our corpus.

```
norm_sentences = tn.normalize_corpus(sentences, text_lower_case=False,
                                     text_stemming=False,
text_lemmatization=False,
                                     stopword_removal=False)
norm_sentences[:3]
```

```
['Elephants are large mammals of the family Elephantidae and the order
Proboscidea',
 'Three species are currently recognised the African bush elephant Loxodonta
africana the African forest elephant L cyclotis and the Asian elephant Elephas
maximus',
 'Elephants are scattered throughout subSaharan Africa South Asia and Southeast
Asia']
```

Now that we have our corpus ready, we will use the pattern, " NP: {<DT>? <JJ>* <NN.*>+}" for extracting all possible noun phrases from our corpus of documents\sentences. You can always experiment with more sophisticated patterns later incorporating verb, adjective or even adverb phrases. However we keep things simple and concise here to focus on the core logic. Once we have our pattern, we will define a function to parse and extract these phrases using the following snippet. We also load any other necessary dependencies at this point.

```
import itertools
stopwords = nltk.corpus.stopwords.words('english')

def get_chunks(sentences, grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
               stopword_list=stopwords):

    all_chunks = []
    chunker = nltk.chunk.regexp.RegexpParser(grammar)

    for sentence in sentences:
```

```
        tagged_sents = [nltk.pos_tag(nltk.word_tokenize(sentence))]

        chunks = [chunker.parse(tagged_sent)
                      for tagged_sent in tagged_sents]

        wtc_sents = [nltk.chunk.tree2conlltags(chunk)
                         for chunk in chunks]

        flattened_chunks = list(
                             itertools.chain.from_iterable(
                                 wtc_sent for wtc_sent in wtc_sents)
                            )

        valid_chunks_tagged = [(status, [wtc for wtc in chunk])
                                    for status, chunk
                                        in itertools.groupby(flattened_chunks,
                                           lambda word_pos_chunk:
word_pos_chunk[2] != 'O')]

        valid_chunks = [' '.join(word.lower()
                                    for word, tag, chunk in wtc_group
                                        if word.lower() not in stopword_list)
                                            for status, wtc_group in
valid_chunks_tagged
                                                if status]

        all_chunks.append(valid_chunks)

    return all_chunks
```

In the above function we have a defined grammar pattern for chunking or extracting noun phrases. We define a chunker over the same pattern and for each sentence in the document, we first annotate it with its POS tags and then build a shallow parse tree with noun phrases as the chunks and all other POS tag based words as chinks which are not parts of any chunks. Once this is done, we use the tree2conlltags function to generate (w,t,c) triples which are words, POS tags and the IOB formatted chunk tags which we discussed in Chapter 3. We remove all tags with chunk tag of 'O' since they are basically words or terms which do not belong to any chunk if you remember our discussion from shallow parsing in Chapter 3. Finally from these valid chunks, we combine the chunked terms to generate phrases from each chunk group. We can see this function in action on our corpus in the following snippet.

```
chunks = get_chunks(norm_sentences)
chunks
```

```
[['elephants', 'large mammals', 'family elephantidae', 'order proboscidea'],
 ['species', 'african bush elephant loxodonta', 'african forest elephant l
cyclotis',
  'asian elephant elephas maximus'],
 ['elephants', 'subsaharan africa south asia', 'southeast asia'],
```

```
...,
...,
 ['incisors', 'tusks', 'weapons', 'tools', 'objects'],
 ['elephants', 'flaps', 'body temperature'],
 ['pillarlike legs', 'great weight'],
...,
...,
 ['threats', 'populations', 'ivory trade', 'animals', 'ivory tusks'],
 ['threats', 'elephants', 'habitat destruction', 'conflicts', 'local people'],
 ['elephants', 'animals', 'asia'],
 ['past', 'war today', 'display', 'zoos', 'entertainment', 'circuses'],
 ['elephants', 'art folklore religion literature', 'popular culture']]
```

The above output shows us all the valid keyphrases per sentence of our document. You can already see since we targeted noun phrases, all phrases talk about noun based entities. We will now build on top of our get_chunks() function by implementing the necessary logic for Step 2 where we will build a TF-IDF based model on our keyphrases using gensim and then compute TF-IDF based weights for each keyphrase based on its occurrence in the corpus. Finally we will sort these keyphrases based on their TF-IDF weights and show the top **N** keyphrases where top_n is specified by the user.

```
from gensim import corpora, models

def get_tfidf_weighted_keyphrases(sentences,
                                  grammar=r'NP: {<DT>? <JJ>* <NN.*>+}',
                                  top_n=10):

    valid_chunks = get_chunks(sentences, grammar=grammar)

    dictionary = corpora.Dictionary(valid_chunks)
    corpus = [dictionary.doc2bow(chunk) for chunk in valid_chunks]

    tfidf = models.TfidfModel(corpus)
    corpus_tfidf = tfidf[corpus]

    weighted_phrases = {dictionary.get(idx): value
                        for doc in corpus_tfidf
                            for idx, value in doc}

    weighted_phrases = sorted(weighted_phrases.items(),
                              key=itemgetter(1), reverse=True)
    weighted_phrases = [(term, round(wt, 3)) for term, wt in weighted_phrases]

    return weighted_phrases[:top_n]
```

We can now test this function on our toy corpus from before by using the following code snippet to generate the top thirty keyphrases.

```
# top 30 tf-idf weighted keyphrases
get_tfidf_weighted_keyphrases(sentences=norm_sentences, top_n=30)
```

```
[('water', 1.0), ('asia', 0.807), ('wild', 0.764), ('great weight', 0.707),
 ('pillarlike legs', 0.707), ('southeast asia', 0.693),
 ('subsaharan africa south asia', 0.693), ('body temperature', 0.693),
 ('flaps', 0.693), ('fissionfusion society', 0.693), ('multiple family groups',
0.693),
 ('art folklore religion literature', 0.693), ('popular culture', 0.693),
 ('ears', 0.681), ('males', 0.653), ('males bulls', 0.653), ('family
elephantidae', 0.607),
 ('large mammals', 0.607), ('years', 0.607), ('environments', 0.577),
('impact', 0.577),
 ('keystone species', 0.577), ('cetaceans', 0.577), ('elephant intelligence',
0.577),
 ('primates', 0.577), ('dead individuals', 0.577), ('kind', 0.577),
('selfawareness', 0.577),
 ('different habitats', 0.57), ('marshes', 0.57)]
```

Interestingly we see various types of elephants being depicted in the keyphrases like Asian and African elephants and also typical attributes of elephants like "*great weight*", "*fission fusion society*" and "*pillar like legs*".

We can also leverage gensim's summarization module which has a keywords function which can be used to extract keywords or phrases from text. This uses a variation of the TextRank algorithm which we shall be exploring during the document summarization section later on.

```
from gensim.summarization import keywords

key_words = keywords(data[0], ratio=1.0, scores=True, lemmatize=True)
[(item, round(score, 3)) for item, score in key_words][:25]
```

```
[('african bush elephant', 0.261), ('including', 0.141), ('family', 0.137),
 ('cow', 0.124), ('forests', 0.108), ('female', 0.103), ('asia', 0.102),
 ('objects', 0.098), ('tigers', 0.098), ('sight', 0.098), ('ivory', 0.098),
 ('males', 0.088), ('folklore', 0.087), ('known', 0.087), ('religion', 0.087),
 ('larger ears', 0.085), ('water', 0.075), ('highly recognisable', 0.075),
 ('breathing lifting', 0.074), ('flaps', 0.073), ('africa', 0.072),
 ('gomphotheres', 0.072), ('animals tend', 0.071), ('success', 0.071),
 ('south', 0.07)]
```

Thus you can get an idea of how keyphrase extraction can extract key important concepts from text documents and summarize them. Try out these functions on other corpora to see interesting results!