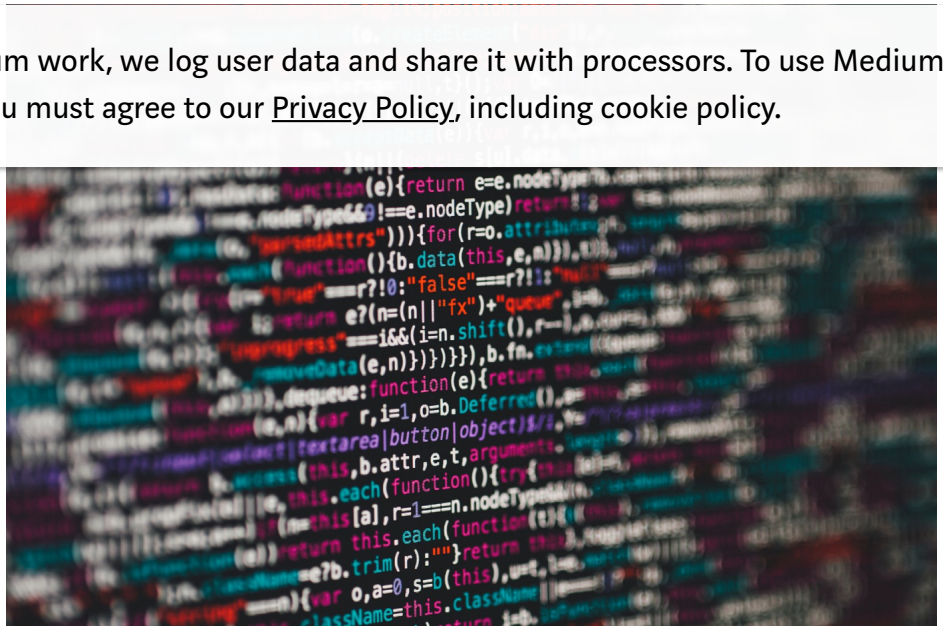


Automated Keyword Extraction from Articles using NLP

Sowmya Vivek Follow

Dec 17, 2018 · 9 min read ★

Background

In research & news articles, keywords form an important component since they provide a concise representation of the article's content. Keywords also play a crucial role in locating the article from information retrieval systems, bibliographic databases and for search engine optimization. Keywords also help to categorize the article into the relevant subject or discipline.

Conventional approaches of extracting keywords involve manual assignment of keywords based on the article content and the authors' judgment. This involves a lot of time & effort and also may not be accurate in terms of selecting the appropriate keywords. With the emergence of Natural Language Processing (NLP), keyword extraction has evolved into being effective as well as efficient.

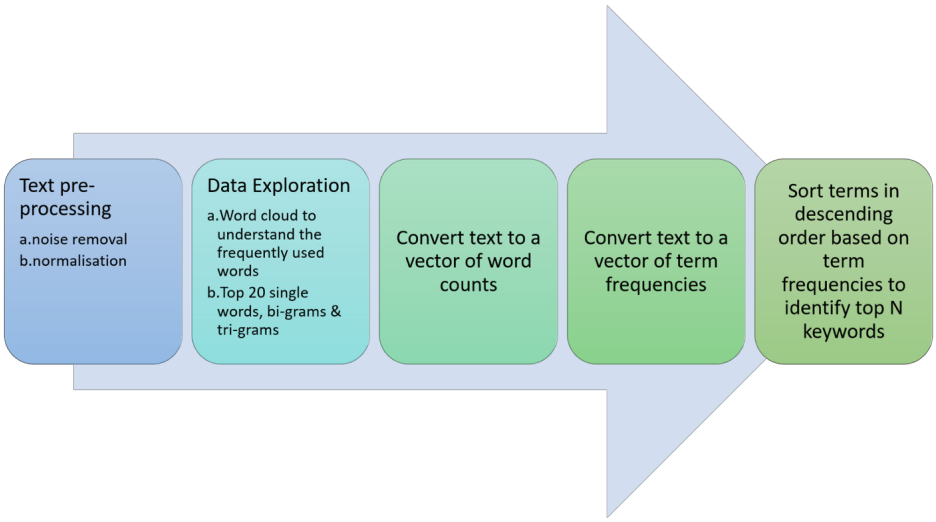
And in this article, we will combine the two—we'll be applying NLP on a collection of articles (more on this below) to extract keywords.

About the dataset

In this article, we will be extracting keywords from a dataset that contains about 3,800 abstracts. The original dataset is from Kaggle —NIPS Paper. Neural Information Processing Systems (NIPS) is one of the top machine learning conferences in the world. This dataset includes the title and abstracts for all NIPS papers to date (ranging from the first 1987 conference to the current 2016 conference).

The original dataset also contains the article text. However, since the focus is on understanding the concept of keyword extraction and using the full article text could be computationally intensive, only abstracts have been used for NLP modelling. The same code block can be used on the full article text to get a better and enhanced keyword extraction.

High-level approach



Importing the dataset

The dataset used for this article is a subset of the *papers.csv* dataset provided in the NIPS paper datasets on Kaggle. Only those rows that contain an abstract have been used. The title and abstract have been concatenated after which the file is saved as a tab separated *.txt file.

```
import pandas
# load the dataset
dataset = pandas.read_csv('papers2.txt', delimiter = '\t')
dataset.head()
```

	id	year	abstract1
0	1861	2000	Algorithms for Non-negative Matrix Factorizati...
1	1975	2001	Characterizing Neural Gain Control using Spike...
2	3163	2007	Competition Adds Complexity It is known that d...
3	3164	2007	Efficient Principled Learning of Thin Junction...
4	3167	2007	Regularized Boost for Semi-Supervised Learning...

As we can see, the dataset contains the article ID, year of publication and the abstract.

Preliminary text exploration

Before we proceed with any text pre-processing, it is advisable to quickly explore the dataset in terms of word counts, most common and most uncommon words.

Fetch word count for each abstract

```
#Fetch wordcount for each abstract
dataset['word_count'] = dataset['abstract1'].apply(lambda
x: len(str(x).split(" ")))
dataset[['abstract1','word_count']].head()
```

	abstract1	word_count
0	Algorithms for Non-negative Matrix Factorizati...	112
1	Characterizing Neural Gain Control using Spike...	88
2	Competition Adds Complexity It is known that d...	70
3	Efficient Principled Learning of Thin Junction...	150
4	Regularized Boost for Semi-Supervised Learning...	124

```
##Descriptive statistics of word counts
dataset.word_count.describe()
```

```
count    3847.000000
mean      155.710684
std        46.080919
min        27.000000
25%       122.000000
50%       150.000000
75%       185.000000
max       325.000000
Name: word_count, dtype: float64
```

The average word count is about 156 words per abstract. The word count ranges from a minimum of 27 to a maximum of 325. The word count is important to give us an indication of the size of the dataset that we are handling as well as the variation in word counts across the rows.

Most common and uncommon words

A peek into the most common words gives insights not only on the frequently used words but also words that could also be potential data specific stop words. A comparison of the most common words and the

default English stop words will give us a list of words that need to be added to a custom stop word list.

```
#Identify common words
freq = pandas.Series('
'.join(dataset['abstract1']).split()).value_counts()[:20]
freq
```

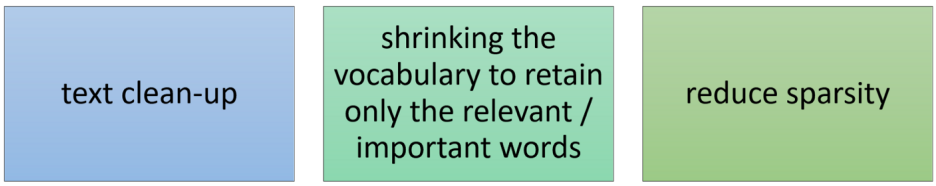
```
the          29516
of           21238
a            16181
and          13882
to           12832
in            9272
for           8219
that          7680
is            7518
We            6112
on            5577
we            5064
with          5015
as            3608
this          3603
are           3458
an            3323
by            3237
can           2891
learning      2844
dtype: int64
```

Most common words

```
#Identify uncommon words
freq1 =  pandas.Series(' '.join(dataset
```

```
            ['abstract1']).split()).value_counts()[-20:]
freq1
```

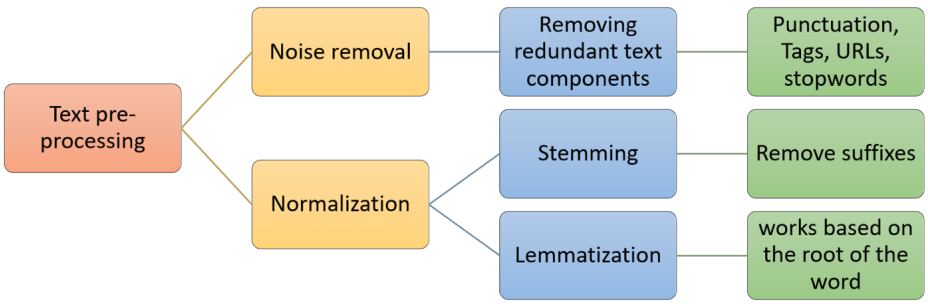
Text Pre-processing



Objectives of text pre-processing

> **Sparsity**: *In text mining, huge matrices are created based on word frequencies with many cells having zero values. This problem is called sparsity and is minimized using various techniques.*

Text pre-processing can be divided into two broad categories—noise removal & normalization. Data components that are redundant to the core text analytics can be considered as noise.



Text pre-processing

Handling multiple occurrences / representations of the same word is called normalization. There are two types of normalization—stemming and lemmatization. Let us consider an example of various versions of the

word learn—learn, learned, learning, learner. Normalisation will convert all these words to a single normalised version—"learn".

> *Stemming normalizes text by removing suffixes.*

> *Lemmatisation is a more advanced technique which works based on the root of the word.*

The following example illustrates the way stemming and lemmatisation work:

```
from nltk.stem.porter import PorterStemmer
from nltk.stem.wordnet import WordNetLemmatizer

lem = WordNetLemmatizer()
stem = PorterStemmer()

word = "inversely"

print("stemming:",stem.stem(word))
print("lemmatization:", lem.lemmatize(word, "v"))
```

```
stemming: invers
lemmatization: inversely
```

To carry out text pre-processing on our dataset, we will first import the required libraries.

```
# Libraries for text preprocessing
import re
import nltk
#nltk.download('stopwords')
from nltk.corpus import stopwords
from nltk.stem.porter import PorterStemmer
from nltk.tokenize import RegexpTokenizer
```

```
#nltk.download('wordnet')
from nltk.stem.wordnet import WordNetLemmatizer
```

Removing stopwords: Stop words include the large number of prepositions, pronouns, conjunctions etc in sentences. These words need to be removed before we analyse the text, so that the frequently used words are mainly the words relevant to the context and not common words used in the text.

There is a default list of stopwords in python nltk library. In addition, we might want to add context specific stopwords for which the "most common words" that we listed in the beginning will be helpful. We will now see how to create a list of stopwords and how to add custom stopwords:

```
##Creating a list of stop words and adding custom stopwords
stop_words = set(stopwords.words("english"))
```

```
##Creating a list of custom stopwords
new_words = ["using", "show", "result", "large", "also",
"iv", "one", "two", "new", "previously", "shown"]
stop_words = stop_words.union(new_words)
```

We will now carry out the pre-processing tasks step-by-step to get a cleaned and normalised text corpus:

```
corpus = []
for i in range(0, 3847):
    #Remove punctuations
    text = re.sub('[^a-zA-Z]', ' ', dataset['abstract1']
[i])

    #Convert to lowercase
    text = text.lower()

    #remove tags
    text=re.sub("&lt;/?.*?&gt;"," &lt;&gt; ",text)

    # remove special characters and digits
    text=re.sub("(\\d|\\W)+"," ",text)

    ##Convert to list from string
```

```
    text = text.split()

    ##Stemming
    ps=PorterStemmer()
```

```
    #Lemmatisation
    lem = WordNetLemmatizer()
    text = [lem.lemmatize(word) for word in text if not
word in
            stop_words]
    text = " ".join(text)
    corpus.append(text)
```

Let us now view an item from the corpus:

```
#View corpus item
corpus[222]
```

```
'extended level method efficient multiple kernel learning consider problem multiple kernel learning mkl formulated convex conca
ve problem past efficient method e semi infinite linear programming silp subgradient descent sd proposed scale multiple kernel
learning despite success method shortcoming sd method utilizes gradient current solution b silp method regularize approximate s
olution obtained cutting plane model work extend level method originally designed optimizing non smooth objective function conv
ex concave optimization apply multiple kernel learning extended level method overcomes drawback silp sd exploiting gradient com
puted past iteration regularizing solution via projection level set empirical study eight uci datasets show extended level meth
od significantly improve efficiency saving average computational time silp method sd method'
```

Data Exploration

We will now visualize the text corpus that we created after pre-processing to get insights on the most frequently used words.

```
#Word cloud
from os import path
from PIL import Image
from wordcloud import WordCloud, STOPWORDS,
ImageColorGenerator


import matplotlib.pyplot as plt
% matplotlib inline


wordcloud = WordCloud(
                          background_color='white',
                          stopwords=stop_words,
                          max_words=100,
                          max_font_size=50,
```

```
                            random_state=42
                           ).generate(str(corpus))
```

```
print(wordcloud)
fig = plt.figure(1)
plt.imshow(wordcloud)
plt.axis('off')
plt.show()
fig.savefig("word1.png", dpi=900)
```

Word cloud

Text preparation

Text in the corpus needs to be converted to a format that can be interpreted by the machine learning algorithms. There are 2 parts of this conversion—Tokenisation and Vectorisation.

> *Tokenisation is the process of converting the continuous text into a list of words. The list of words is then converted to a matrix of integers by the process of vectorisation. Vectorisation is also called feature extraction.*

For text preparation we use the ***bag of words model*** which ignores the sequence of the words and only considers word frequencies.

Creating a vector of word counts

As the first step of conversion, we will use the *CountVectoriser* to tokenise the text and build a vocabulary of known words. We first create a

variable “cv” of the CountVectoriser class, and then evoke the fit_transform function to learn and build the vocabulary.

```
from sklearn.feature_extraction.text import CountVectorizer
import re


cv=CountVectorizer(max_df=0.8,stop_words=stop_words,
max_features=10000, ngram_range=(1,3))
X=cv.fit_transform(corpus)
```

Let us now understand the parameters passed into the function:

- cv=CountVectorizer(max_df=0.8,stop_words=stop_words, max_features=10000, ngram_range=(1,3))
- max_df—When building the vocabulary ignore terms that have a document frequency strictly higher than the given threshold (corpus-specific stop words). This is to ensure that we only have words relevant to the context and not commonly used words.
- max_features—determines the number of columns in the matrix.
- n-gram range—we would want to look at a list of single words, two words (bi-grams) and three words (tri-gram) combinations.

An encoded vector is returned with a length of the entire vocabulary.

```
list(cv.vocabulary_.keys())[:10]
```

```
['algorithm',
 'non',
 'negative',
 'matrix',
 'factorization',
 'nmf',
 'useful',
 'decomposition',
 'multivariate',
 'data']
```

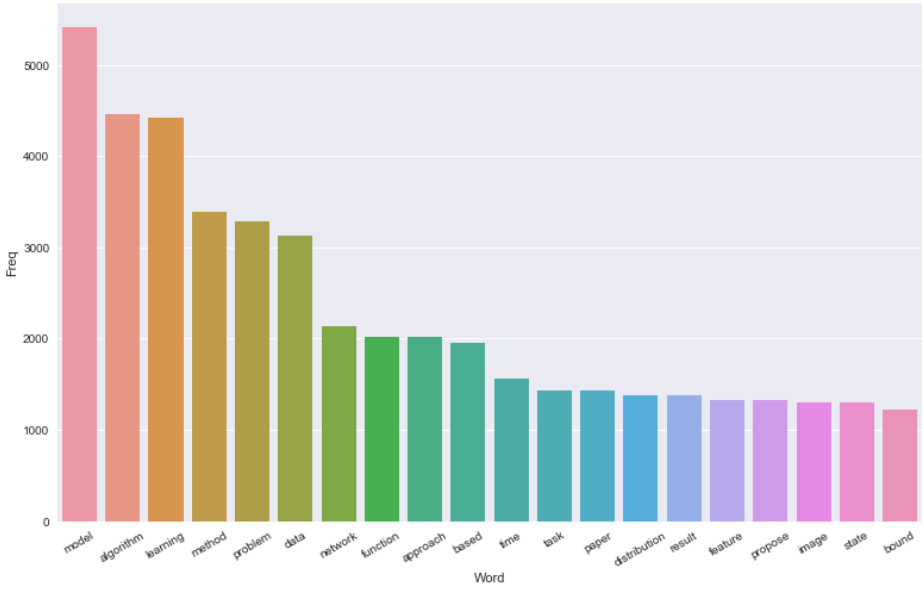
Visualize top N uni-grams, bi-grams & tri-grams

We can use the CountVectoriser to visualise the top 20 unigrams, bi-grams and tri-grams.

```
#Most frequently occuring words
def get_top_n_words(corpus, n=None):
    vec = CountVectorizer().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[:n]


#Convert most freq words to dataframe for plotting bar plot
top_words = get_top_n_words(corpus, n=20)
top_df = pandas.DataFrame(top_words)
top_df.columns=["Word", "Freq"]
```

```
#Barplot of most freq words
import seaborn as sns
sns.set(rc={'figure.figsize':(13,8)})
g = sns.barplot(x="Word", y="Freq", data=top_df)
g.set_xticklabels(g.get_xticklabels(), rotation=30)
```

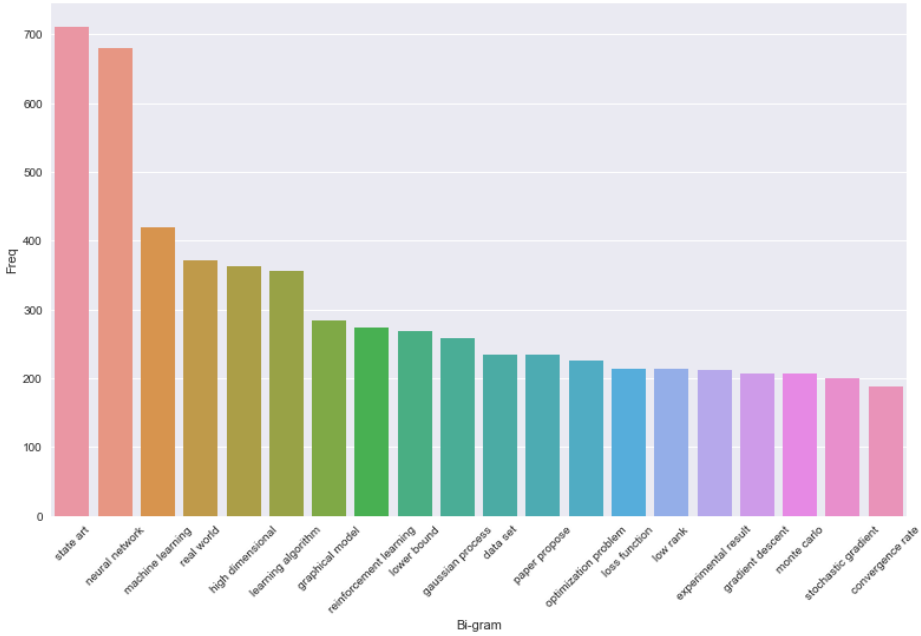


Bar plot of most frequently occurring uni-grams

```
#Most frequently occuring Bi-grams
def get_top_n2_words(corpus, n=None):
    vec1 = CountVectorizer(ngram_range=(2,2),
            max_features=2000).fit(corpus)
    bag_of_words = vec1.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx
in
                  vec1.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1],
                reverse=True)
    return words_freq[:n]


top2_words = get_top_n2_words(corpus, n=20)
top2_df = pandas.DataFrame(top2_words)
top2_df.columns=["Bi-gram", "Freq"]
print(top2_df)
```

```
#Barplot of most freq Bi-grams
import seaborn as sns
sns.set(rc={'figure.figsize':(13,8)})
h=sns.barplot(x="Bi-gram", y="Freq", data=top2_df)
h.set_xticklabels(h.get_xticklabels(), rotation=45)
```

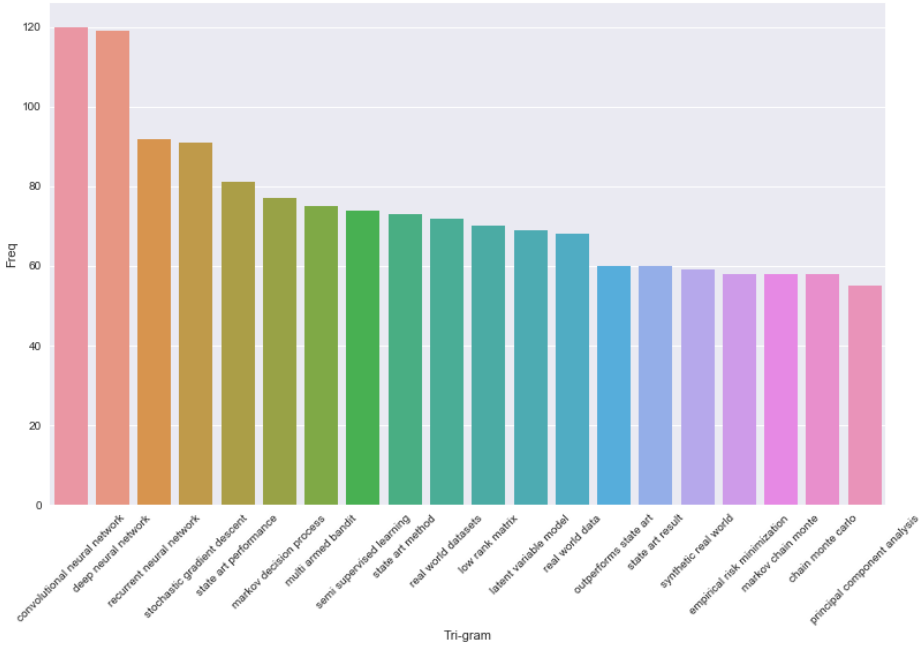


Bar plot of most frequently occurring bi-grams

```
#Most frequently occuring Tri-grams
def get_top_n3_words(corpus, n=None):
    vec1 = CountVectorizer(ngram_range=(3,3),
           max_features=2000).fit(corpus)
    bag_of_words = vec1.transform(corpus)
    sum_words = bag_of_words.sum(axis=0)
    words_freq = [(word, sum_words[0, idx]) for word, idx
in
                  vec1.vocabulary_.items()]
    words_freq =sorted(words_freq, key = lambda x: x[1],
                reverse=True)
    return words_freq[:n]

top3_words = get_top_n3_words(corpus, n=20)
top3_df = pandas.DataFrame(top3_words)
top3_df.columns=["Tri-gram", "Freq"]
print(top3_df)
```

```
#Barplot of most freq Tri-grams
import seaborn as sns
sns.set(rc={'figure.figsize':(13,8)})
j=sns.barplot(x="Tri-gram", y="Freq", data=top3_df)
j.set_xticklabels(j.get_xticklabels(), rotation=45)
```



Bar plot of most frequently occurring tri-grams

Converting to a matrix of integers

The next step of refining the word counts is using the TF-IDF vectoriser. The deficiency of a mere word count obtained from the countVectoriser is that, large counts of certain common words may dilute the impact of more context specific words in the corpus. This is overcome by the TF-IDF vectoriser which penalizes words that appear several times across the document. TF-IDF are word frequency scores that highlight words that are more important to the context rather than those that appear frequently across documents.

TF-IDF consists of 2 components:

- TF—term frequency
- IDF—Inverse document frequency

$$\bullet\ \text{TF} = \frac{\text{Frequency of a term in a document}}{\text{total number of terms in the document}}$$

$$\bullet\ \text{IDF} = \frac{\log(\text{Total documents})}{\#\ \text{of documents with the term}}$$

```
from sklearn.feature_extraction.text import
TfidfTransformer

tfidf_transformer=TfidfTransformer(smooth_idf=True,use_idf=
True)
tfidf_transformer.fit(X)
```

```
# get feature names
feature_names=cv.get_feature_names()

# fetch document for which keywords needs to be extracted
doc=corpus[532]

#generate tf-idf for the given document
tf_idf_vector=tfidf_transformer.transform(cv.transform([doc
]))
```

Based on the TF-IDF scores, we can extract the words with the highest scores to get the keywords for a document.

```
#Function for sorting tf_idf in descending order
from scipy.sparse import coo_matrix
def sort_coo(coo_matrix):
    tuples = zip(coo_matrix.col, coo_matrix.data)
    return sorted(tuples, key=lambda x: (x[1], x[0]),
reverse=True)

def extract_topn_from_vector(feature_names, sorted_items,
topn=10):
    """get the feature names and tf-idf score of top n
items"""

    #use only topn items from vector
    sorted_items = sorted_items[:topn]

    score_vals = []
    feature_vals = []

    # word index and corresponding tf-idf score
    for idx, score in sorted_items:

        #keep track of feature name and its corresponding
score
        score_vals.append(round(score, 3))
        feature_vals.append(feature_names[idx])

    #create a tuples of feature,score
    #results = zip(feature_vals,score_vals)
    results= {}
    for idx in range(len(feature_vals)):
        results[feature_vals[idx]]=score_vals[idx]
```

```
    return results
```

```
#sort the tf-idf vectors by descending order of scores
```

```
sorted_items=sort_coo(tf_idf_vector.tocoo())


#extract only the top n; n here is 10
keywords=extract_topn_from_vector(feature_names,sorted_item
s,5)

# now print the results
print("\nAbstract:")
print(doc)
print("\nKeywords:")
for k in keywords:
    print(k,keywords[k])
```

```
Abstract:
hierarchical learning dimensional bias human categorization existing model categorization typically represent classified item p
oint multidimensional space mathematical point view infinite number basis set used represent point space choice basis set psych
ologically crucial people generally choose basis dimension strong preference generalize along ax dimension diagonally make choi
ce dimension special explore idea dimension used people echo natural variation environment specifically present rational model
assume dimension learns type dimensional generalization people display bias shaped exposing model many category structure hypot
hesized like child encounter model viewed type transformed dirichlet process mixture model learning base distribution dirichlet
process allows dimensional generalization learning behaviour model capture developmental shift roughly isotropic child axis ali
gned generalization adult

Keywords:
dimension 0.301
people 0.268
basis 0.234
child 0.205
categorization 0.193
```

Concluding remarks

Ideally for the IDF calculation to be effective, it should be based on a large corpora and a good representative of the text for which the keywords need to be extracted. In our example, if we use the full article text instead of the abstracts, the IDF extraction would be much more effective. However, considering the size of the dataset, I have limited the corpora to just the abstracts for the purpose of demonstration.

This is a fairly simple approach to understand fundamental concepts of NLP and to provide a good hands-on practice with some python codes on a real-life use case. The same approach can be used to extract keywords from news feeds & social media feeds.