# Artificial intelligence and pattern recognition

Denis Festa

University of West Bohemia
Faculty of Applied Sciences

June 12, 2020

# Introduction

# Scopes

What is this project about? The analysis of unstructured data (like text files, different from databases which are considered structured data). With analysis I mean that the user will be able to write a query and as an answer he will receive what is the document in the collection of data that is most likely the one he was looking for (somewhat like a search engine, of course not as complex as a real search engine would be).
The other thing that the user can get is the topic that is more likely to be dealt with from the query he gave (as a text classifier would do, only, this is not as complex as a real classifier would be).
Due to the presence of the classifier feature, the user can also write as a query a document ID instead of writing the whole text of the document, but the result would conceptually be the same.

# Data

As I wrote about data, those data are a corpus of English documents called Reuters-21578, a collection of documents that appeared on Reuters newswire in 1987, available at http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html. All of the documents have an ID, but if you look at any of those articles (in the reut folder) you will see two XML attributes called OLDID and NEWID, the latter is the one I used.

# Setting up

If you didn't receive 3GB of data within the source code, then it's likely that you will have to generate them manually. How to do that is explained in the comments in the source code,
mainly it's about calling functions inside the .py files, I suggest you to call them from a python console.
The two most important files from which you will have to call the functions are:

- indexer.py: this one will create indexes and dictionaries that will be used by:
  - Naive Bayes classifier
  - Optimized cosine distance classifier

- encoder.py: this one will need some results from the previous one, so it's easier if you follow this order; it will create for each document in the collection (7063 are the one I used for training) three different embedded vectors:
  - One hot / bag of word encoding,
  - Term frequency encoding,
  - tfidf encoding.

  The encoder.py will be useful for:
  - Cosine distance classifier,
  - Rocchio's classifier,
  - Neural network classifier.

The two files indexer.py and encoder.py are the most important in relation to the creation of necessary data, but there are still other things needed, they will explained inside each of the .py files related to each specific classifier, for example, in rocchio.py there are two functions to be used, that are

- write_encoded_vectors_for_all_topics()
- write_all_centroids().

# Natural language processing

Before doing any computation on data, every document undergoes a small preprocessing, that is:

- lowercase,
- lemmatization,
- filtering out stopwords.

The last two are done using NLTK library: even if the imports are already done in the source code, it's likely that you'll face an error about nltk library, it will ask you to download WordNetLemmatizer and stopwords giving you the command to do that, it will be something like nltk.download().
Another thing you'll need is BeautifulSoup4 library, which is used to read the content of the Reuters collection, which is provided in .sgm format (somewhat similar to XML).
Everything related to preprocessing and handling .sgm files can be found in preprocesser.py and navigator.py .

# Naive Bayes

# Naive Bayes

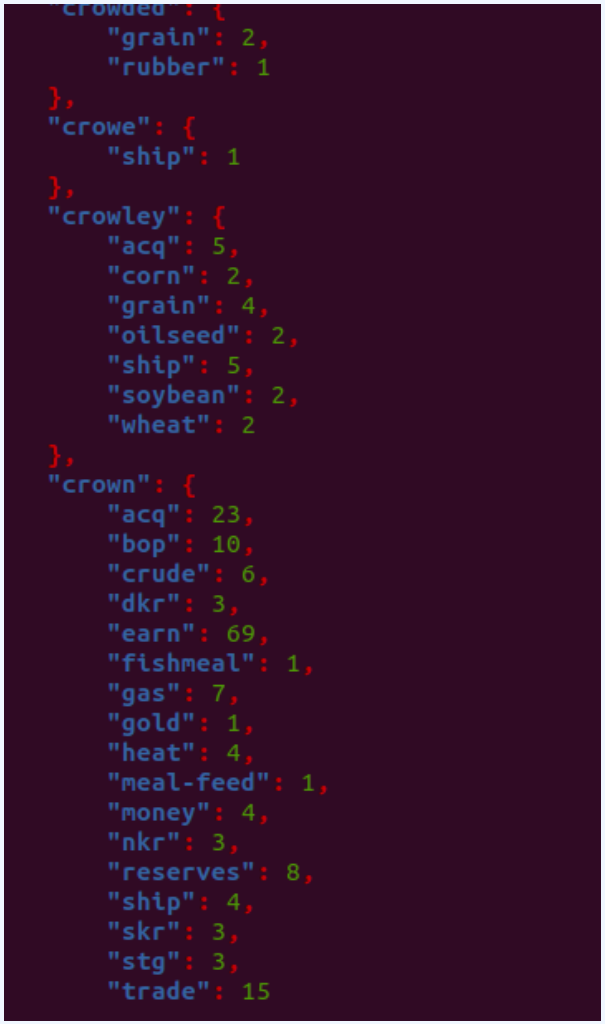
Dependencies:

- topic_inverted_index.json
- topic_nonverted_index.json
- topics_dictionary.json
- topics_frequencies.json
- dictionary.json

All of those can be created using indexer.py

# WHAT WITH THESE FILES?

# TOPIC_INVERTED_INDEX.JSON



Topic inverted index is a dictionary whose key is a word and the associated value is another dictionary whose key is a topic in which that word appears at least once and the associated value is the number of times that word appears in that topic.

# TOPIC_NONVERTED_INDEX.JSON

Topic nonverted index is a dictionary whose key is a topic and the associated value is another dictionary whose key is a word that appears at least once in a document with that topic and the associated value is the number of times that word appears in that topic.

# TOPICS_DICTIONARY.JSON

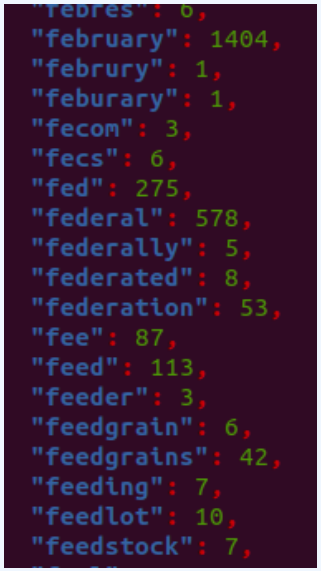

```
"acq": 1488,
"alum": 33,
"austdlr": 4,
"barley": 33,
"bop": 62,
"can": 3,
"carcass": 50,
"castor-oil": 1,
"castorseed": 1,
"citruspulp": 1,
"cocoa": 50,
"coconut": 4,
"coconut-oil": 4,
"coffee": 110,
"copper": 47,
"copra-cake": 2,
"corn": 160,
"corn-oil": 1,
"cornglutenfeed": 2,
"cotton": 38,
"cotton-oil": 1,
"cpi": 60,
"cpu": 2,
"crude": 349,
```

Topics dictionary is a dictionary whose key is a topic and the associated value is the number of documents in the corpus (train set) classified with that topic.

# TOPICS_FREQUENCIES.JSON

Topics frequencies is a dictionary that directly derives from topics dictionary, instead of a natural number, the key is a topic and the value is the frequency (probability) of that topic.

# DICTIONARY.JSON



dictionary is a dictionary whose value is a word and the associated value is the number of times that word appears in the corpus (training set).

# Why are they useful?

The main idea of Naive Bayes classifier is to maximize

$$P(T_k) \prod_{i=1}^{n} P(w_i|T_k)$$

$P(C_k)$ it the probability of topic $k$ to be extracted, and that's where topics_frequencies.json comes useful, topics_dictionary.json is just needed to create the former.
$P(w_i|T_k)$ is the probability of word $i$ to be extracted given that we are looking at a document of topic $T_k$, so it is computed as $\frac{n_{i_k}}{n_k}$ where $n_{i_k}$ is the number of times $w_i$ occurs inside documents with topic $T_k$ and $n_k$ is the sum of all the occurrences of all the words that appear in documents with topic $T_k$.

Here topic_inverted_index.json comes to use because it's easy to get how many time a given word appears in a given topic ($n_{i_k}$), whereas topic_nonverted_index.json comes to use because it's easy, given a topic, to sum over all the values (occurrences) of all the words that occur in that topic ($n_k$).

# A computer-architecture friendly formula

From

$$P(T_k) \prod_{i=1}^{n} P(w_i|T_k)$$

to

$$\log(P(T_k)) + \sum_{i=1}^{n} \log(P(w_i|T_k))$$

means much for the computer architecture, in fact when $n$ starts to increase a product of $n$ factors with value $v$ such that $0 < v < 1$ can lead to results like $10^{-}100$ which the computer is not able to handle. Switching to the logarithm creates no problem, in fact we are not interested in the actual probability value, we just want to maximize it and logarithm is a monotonic crescent function, so it's perfect for our scopes.

# Optimizes cosine distance

# Optimized cosine distance

Dependencies:

- inverted_index.json
- idf.json
- inv_tfidf.json

All of those can be created using indexer.py
Another resource that is not conceptually related to cosine distance but it's used in main.py to get the topics from the closest documents to the query is

- document_topic_dictionary.json, that you can create using the function write_document_topic_dictionary() inside indexer.py .

# WHAT WITH THESE FILES?

# INVERTED_INDEX.JSON

Inverted index is a dictionary whose key is a word and the value is another dictionary whose key is a document ID in which the word appears at least once and the value is the number of times the word appears in that document.

# IDF.JSON


```
"curb": 4.855291984176452,
"curbed": 7.476330808289032,
"curbing": 7.0708657001808675,
"cure": 8.169477988848977,
"curitiba": 8.862625169408922,
"curled": 8.862625169408922,
"currency": 2.9765211379587666,
"current": 2.3361303098381323,
"currently": 3.4644624678911695,
"curry": 8.862625169408922,
"cursory": 8.862625169408922,
"curtail": 7.0708657001808675,
"curtailed": 6.9167150203536085,
"curtailing": 8.169477988848977,
"curtailment": 7.476330808289032,
"curti": 8.862625169408922,
"curtis": 8.169477988848977,
"curve": 7.2531872569748215,
"cushion": 7.2531872569748215,
"custer": 8.169477988848977,
"custom": 5.124955551125554,
"customarily": 7.476330808289032,
"customary": 8.169477988848977,
"customer": 3.9067981118076616,
"customized": 8.862625169408922,
"cut": 2.8587381023023832,
"cutback": 6.029411825352706,
"cutlet": 8.862625169408922,
"cutrale": 8.169477988848977,
"cutter": 8.862625169408922,
```

Idf is a dictionary whose key is a word and the associated value is the idf coefficient of that word.

# INV_TFIDF.JSON

Inverted tf_idf is a dictionary whose key is a document ID and the associated value is another dictionary whose key is a word that appears in that document and the value is the tf_idf coeffcient.

# DOCUMENT_TOPIC_DICTIONARY.JSON


```
{
    "1": [
        "cocoa"
    ],
    "10": [
        "acq"
    ],
    "100": [
        "money-supply"
    ],
    "1000": [
        "acq"
    ],
    "10000": [
        "earn"
    ],
    "10002": [
        "earn"
    ],
    "10005": [
        "trade",
        "acq"
    ],
    "10008": [
        "earn"
    ],
    "10011": [
        "crude",
        "nat-gas"
    ],
    "10014": [
        "coffee",
        "cocoa",
        "sugar"
```

Document topic dictionary is a dictionary whose key is a document ID and the associated value is the list of topics it deals with.

# Why are they useful?

Cosine similarity is a measure of similarity between two documents, that derives from the measure of the angle between two vectors. To get the benefit of this measure, there is need to convert a document in a vector, in this project documents are converted into three different type of vectors:

- one hot encoded vector: a vector with the size equal to the number $n$ of words in the dictionary (dictionary.json). It has 0 at position $i$ when the word $w_i$ in the dictionary is not present in the document, 1 if it is present at least once;
- term frequency encoded vector: with the size equal to the number $n$ of words in the dictionary (dictionary.json). It has 0 at position $i$ when the word $w_i$ in the dictionary is not present in the document, $k$ if it occurs $k$ times.

- tf_idf encoded vector: tf stands for "term frequency" and idf stands for "inverse document frequency". It has 0 at position $i$ when the word $w_i$ in the dictionary is not present in the docuement, $r$ if it occurs, and $r$ is the multiplication between term frequency (the same as in the previous type of encoding) and inverse document frequency. Inverse document frequency is computed as $\frac{N}{n_{w_i}}$ where $N$ is the number of documents in the corpus, $n_{w_i}$ is the number of documents in which the i-th word of the dictionary $w_i$ occurs.

Cosine similarity between two vectors $\vec{u}$ and $\vec{v}$ is computed as

$$\frac{\sum_{i=1}^{n} \vec{u}_i \cdot \vec{v}_i}{\sqrt{\sum_{i=1}^{n} \vec{v}_i^2} \sqrt{\sum_{i=1}^{n} \vec{v}_i^2}}$$

where $n$ is the size of the vectors.
When a document is turned into a vector, it is predictable that the vector will be made of a many zeros and a few of values different from zero, because that vector has a size equal to the dictionary of all the words ever seen so far in all the documents of the corpus.

The two different approaches to compute the similarity are:

- Naive: just compute the product the similarity using a library (like SciPy)

- CPU-friendly: instead of computing the similarity between the vectors obtained from the encoding, which as said before are sparse (many zeroes, few interesting values), we create two Python dictionaries whose keys are the words contained in the respective document and the values are 1 for one-hot encoding or *n* for term frequency or *r* for tf_idf encoding, then the product between the two 'vectors' (now Python dictionaries) is a sum of products that are 0 when a word in one of them doesn't appear in the other. In this case there are still zeros, not in the two vectors but conceptually as "word in document $\vec{u}$ doesn't appear in document $\vec{v}$", anyway they are much less. Here is where inv_tfidf.json comes handy, it already has for each document the list of words in that document with their tf_idf coefficient; inverted_index was useful in order to create the former. Obviously this approach, being CPU-friendly, loses in memory-friendliness, (in this case, inv_tfidf shoud be 16,8 MB).

# Where is classification?

To classify a new document, the closest document (in terms of cosine similarity) in the corpus (or maybe the first *n* closest documents) is (are) taken and its (their) topics are returned as a result, here is where document_topic_dictionary.json comes handy.

# Cosine distance

# Cosine distance

Dependencies:

- dictionary.json
- idf.json
- all the vectors: three vectors for each document in the collection (7063 documents were used for training).

The first two can be created using indexer.py.
The vectors can be created using write_encodings() from encoder.py.
Another resource that is not conceptually related to cosine distance but it's used in main.py to get the topics from the closest documents to the query is

- document_topic_dictionary.json, that you can create using the function write_document_topic_dictionary() inside indexer.py .

# WHAT WITH THESE FILES?

# Rocchio

# Rocchio

Dependencies:

- dictionary.json
- idf.json
- document_topic_dictionary.json
- all the vectors: three vectors for each document in the collection (7063 documents were used for training), they can be created using write_encodings() from encoder.py
- use write_encoded_vectors_for_all_topics() from rocchio.py
- use write_all_centroids() from rocchio.py .

# Why are these files useful?

Rocchio's classification algorithm is not that distant from the previous ones, it still uses cosine similarity (also other metrics can be uses, like Euclidean one, but cosine similarity is one of the best). Instead of computing distance between the document to be classified and all the documents in the corpus to find the closest one, it first computes centroids. A centroid is a sort of average of a list of vectors:

$$\vec{\mu} = \frac{1}{n} \sum_{i=1}^{n} \vec{v}_i$$

where $\vec{\mu}$ is the centroid, $n$ is the number of vectors, $v_i$ is the i-th vector of the list.
For each topic we calculate the centroid of all the vectors (derived from the documents) that deal with that topic, then we find the closest centroid (in terms of cosine similarity) to the document to be classified.