Università Degli Studi di Napoli
Federico II

Academic Year 2019/2020

Automatic Text Summarization

Information Retrieval Systems

Author
Aldo Altobelli

Prof.
Antonio Maria Rinaldi

Contents

1 Introduction **2**

1.1 Extractive Text Summarization 2

1.1.1 Topic Representations 4

1.1.2 Indicator Representations 5

1.2 Project Goals 5

2 Solr Configuration **7**

2.1 Instance Creation 8

2.2 Schema Customization 9

2.3 Solr-Python Interaction 12

2.3.1 Anatomy of a Solr Response 13

3 TF-IDF **15**

3.1 General Functions 15

3.2 Algorithm 17

4 Machine Learning **20**

4.1 Introduction 20

4.2 Feature Extraction 21

4.2.1 Article-Independent Features 23

4.2.2 Article-Dependent Features 25

4.2.3 Feature Matrix 27

4.3 Training 28

5 Evaluation **30**

5.1 ROUGE Details 30

5.2 Summaries Generation 32

5.3 Comparisons 33

5.4 Conclusions 36

Bibliography **38**

Chapter 1

Introduction

The main goal of Automatic Text Summarization is the production of summaries from documents. Summaries are excerpts of the original text that preserve key information content and overall meaning. They are useful in contexts of large amounts of data availability, since they provide key concepts and ideas of the document, freeing the user from the tedious task of exploring the text in its entirety.

The overwhelming growth of data and information in recent years has rekindled the interest in Automatic Text Summarization, causing great demand for research in such field. The volume of text is an invaluable source of information and therefore it needs to be efficiently summarized to be useful.

Summarization techniques can be categorized in two main families: **Extractive** and **Abstractive** Text Summarization. Methods belonging to the former simply extract the most "relevant" sentences verbatim from the document, while techniques of the latter aim at producing summaries from the ground up that resemble the form, syntax and semantic of a human generated summary.

It is clear that Abstract Summarization poses a much greater challenge: for summaries to be effective, processes must implement some way of mimicking human behavior. Even with modern technological advancements such techniques are still in an embryonic phase and therefore the focus of the following work remains on Extractive techniques.

1.1 Extractive Text Summarization

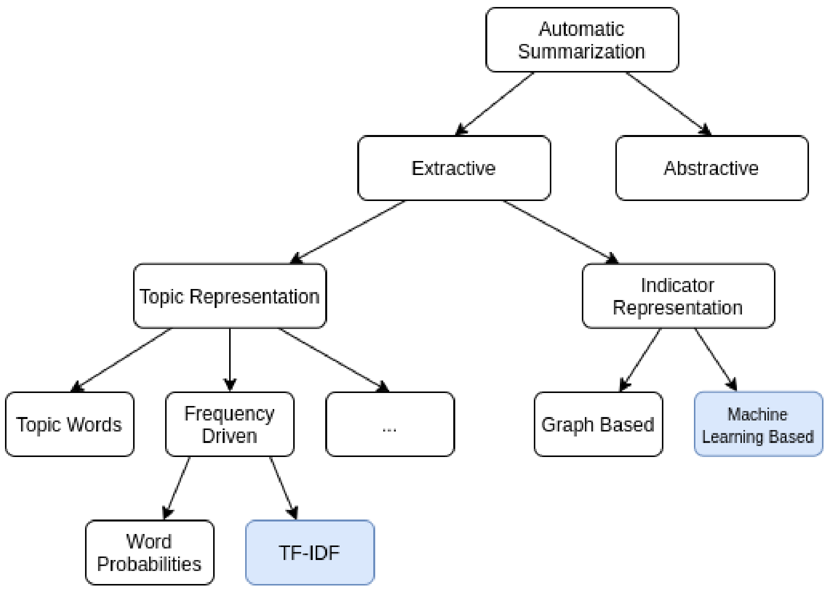
The goal of the following work is to analyze and implement different techniques of Extractive Text Summarization, and then compare their results. As mentioned, extractive techniques produce summaries by selecting a subset of the document's original sentences. The main tasks of an Extractive Summarization process are:

1. Construction of an intermediate representation of the input text;
2. Scoring of the sentences based on such representation;
3. Selection of a summary comprising of a number of sentences (e.g. sentences with the highest score).

The intermediate representation's main task is the highlighting or the taking out of the most important information of the text to be summarized. The methods of extractive summarization differ in the way in which they build this intermediate representation of the text (the first step), and they can be classified in two families:

- **Topic Representations**: transform the text, with techniques that differ in terms of their complexity and representation models, in order to *interpret the topic(s) discussed*;
- **Indicator Representations**: describe every sentence as a list of features (indicators) of importance such as sentence length, position in the document, having certain phrases, etc.

The following image shows a taxonomy of extractive methods, based on such classification:



Once the intermediate representations have been computed, scores are assigned to each sentence to calculate their *summarization score.* In topic representation, the score of a sentence depends on the topic words it contains. In indicator representation, the score depends on the features of the sentences. Finally, the summaries are generated by selecting the "best" sentences, and the selection methods are varied.

1.1.1 Topic Representations

Topic Words

The main goal is to identify words that describe the topic of the input document. Earliest works used frequency thresholds to locate the descriptive words in the document and represent the topic of the document. Advanced techniques use log-likelihood ratio test to identify explanatory words which in summarization literature are called the “topic signature”. Sentences are then scored on the proportion of the topic signature in the sentence.

Frequency Driven

This approach uses frequency of words as indicators of *summarization importance.* A score is assigned to each word in a sentence and the sentence is judged on an aggregation of those scores. The two most common techniques in this category use *word probabilities* or *TF-IDF scores.* The probability of a word w is the number of occurrences of w in the input, $f(w)$, divided by the number of total words in the input:

$$P(w) = \frac{f(w)}{N}$$

Input is intentionally vaguely defined since it can relate to sentences, documents or collections.

The TF-IDF score is the classic weighting scheme used in IR systems. Given a word w_i in document d_j its score is:

$$\text{TF-IDF}(w_{i,j}) = (1 + \log f_{i,j}) \cdot \log \frac{N}{n_i}$$

where $f_{i,j}$ is the frequency of occurrence of w_i in d_j and n_i is the number of documents in which w_i appears.

After weights have been assigned to words, sentences are then scored based on the sum of the weights of their words.

1.1.2 Indicator Representations

In these approaches, the importance of a sentence is not dependent directly on the words it contains but rather on sentence features extracted from the text.

Graph Based

These methods are inspired by the Page Rank algorithm. They represent text documents as connected graphs. The sentences are represented as the nodes of the graphs and edges between the nodes are a measure of distance between the two sentences. These techniques measure the similarity between two sentences and if it is greater then a threshold they are connected. The most often used method for similarity measure is cosine similarity with TF-IDF weights for words. Sentences that are connected to many other sentences in the partition are possibly the center of the graph and more likely to be included in the summary.

Machine Learning Based

Machine learning methods approach the summarization problem as a binary classification problem. The models try to classify sentences into *summary* or *non-summary* from the extracted features. Features can be document-independent such as sentence length, or document-dependent such as the position of the sentence in a document. These features are explored in depth in the next chapters. The most used models in this context are Naive Bayes Classifiers, Decision Trees, and SVMs.

1.2 Project Goals

The main goal of the following work is to implement one method for each category of extractive techniques, namely the TF-IDF approach and the Machine Learning approach. Furthermore, the implementation must be contextualized in a typical Information Retrieval System, and specifically Apache Solr.

Solr is a standalone enterprise search server with a REST-like API. Documents are stored in it ("indexed") via JSON, XML, CSV or binary over HTTP. An instance of Solr is implemented and queried in this work and its features will be used to implement the different algorithms.

The final goal is to evaluate the effectiveness and the usefulness of the generated summaries and compare the results among the two explored techniques. Classic evaluation metrics, such as Recall, can be used to perform such evaluation by counting how many sentences the automatic generated (candidate) summary and a human (reference) summary have in common, and divide the total by the number of sentences in the reference.

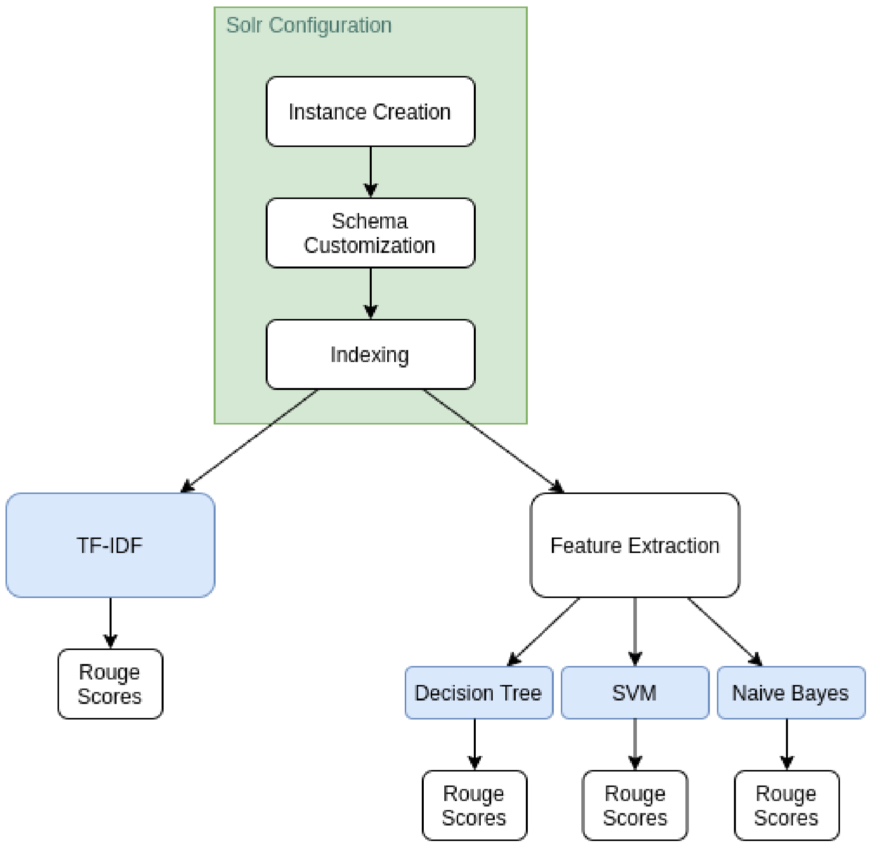
For a more sophisticated approach, **ROUGE** is the most widely used metric in literature for automatic summarization. It stands for *Recall-Oriented Understudy for Gisting Evaluation* and it automatically determines the quality of a summary by comparing common N-grams among candidate and reference summary. More details on this metric are explained in the Evaluation chapter.

The obvious main challenge of evaluation is that, in order to fine tune an automatic summarization pipeline, it is necessary to work with datasets that include a reference, that is a human generated summary. Many publicly available datasets satisfy this constraint. The one chosen for this project is the **BBC News Dataset** which consists of 2225 documents and their respective summaries from the BBC news website, corresponding to stories in five topical areas (business, entertainment, politics, sport, tech) from 2004-2005 [6].

Chapter 2

Solr Configuration

Before implementing the aforementioned summarization techniques, it is important to properly configure, customize and index a Solr Collection. These preliminary operations are an integral part of the project, and represent the first part of a more general workflow:



The first step of the process is the creation of a Solr collection that indexes the documents in the chosen dataset.

In this project, Solr was adopted in its *cloud* form. That is because Streaming expressions[1], extensively used in the following sections, are only available in such mode. Streaming expressions provide useful tools much needed in the context of the summarization. As an example, they can extract TF-IDF values from a *sub-set* of the collection's documents or they can run an **Analyzer** on a particular sentence, extracting processed tokens from that sentence.

2.1 Instance Creation

To create a document collection in cloud mode, we can use a script made available by Solr, which provides an easy and immediate initialization process:

```
$ [Solr_dir]/bin/solr start -e cloud
```

The user is then guided in the creation process:

```
This interactive session will help you launch a SolrCloud
   cluster on your local workstation. To begin, how many
   Solr nodes would you like to run in your local cluster?
   (specify 1-4 nodes) [2]: _
```

For this example, we set the number of nodes to 1.

```
Ok, let's start up 1 Solr nodes for your example SolrCloud
   cluster.
Please enter the port for node1 [8983]: _
```

The port number is arbitrary, but we set it to 8984.

```
Now let's create a new collection for indexing documents in
    your 1-node cluster.
Please provide a name for your new collection: _
```

[1]From the Solr documentation: *"Streaming expressions expose the capabilities of SolrCloud as composable functions. These functions provide a system for searching, transforming, analyzing, and visualizing data stored in SolrCloud collections."*

We call our collection "myDocs", and set to 1 the number of shards, to 1 the number of shards per replica, and to "_default" the configuration.

2.2 Schema Customization

In order to index collections in Solr, it is necessary to customize its **schema**, according to the documents in the dataset. Specifically, the schema is first injected with custom FieldTypes that model the properties of the documents' single fields, and then with the Fields themselves that model the documents' structure.

It is necessary to analyze the documents of the dataset to better comprehend what kind of field must be defined. The following is a document of the dataset in JSON format:

```
{"docId": 117,
 "category": "business",
 "title": "Manufacturing recovery 'slowing'",
 "full_text": "UK manufacturing grew at its slowest pace in
    one-and-a-half years in January, according to a survey.
    The Chartered Institute of Purchasing and Supply (CIPS)
    said its purchasing manager index (PMI) fell to 51,8 from
     a revised 53,3 in December.  But, despite missing
    forecasts of 53,7, the PMI number remained above 50 -
    indicating expansion in the sector.  The CIPS said that
    the strong pound had dented exports while rising oil and
    metals prices had kept costs high. The survey added that
    rising input prices and cooling demand had deterred
    factory managers from hiring new workers in an effort to
    cut costs.  That triggered the second successive monthly
    fall in the CIPS employment index to 48,3 - its lowest
    level since June 2003.  The survey is more upbeat than
    official figures - which suggest that manufacturing is in
     recession - but analysts said the survey did suggest
    that the manufacturing recovery was running out of steam.
     "It appears that the UK is in a two-tier economy again,"
     said Prebon Yamane economist Lena Komileva. "You have
    weakness in manufacturing, which I think would concern
    policymakers at the Bank of England.""
 "summary": "The survey is more upbeat than official figures
    - which suggest that manufacturing is in recession - but
    analysts said the survey did suggest that the
    manufacturing recovery was running out of steam. The
    Chartered Institute of Purchasing and Supply (CIPS) said
    its purchasing manager index (PMI) fell to 51,8 from a
    revised 53,3 in December. UK manufacturing grew at its
    slowest pace in one-and-a-half years in January,
    according to a survey. The CIPS said that the strong
    pound had dented exports while rising oil and metals
    prices had kept costs high."}
```

Documents are composed of 5 fields:

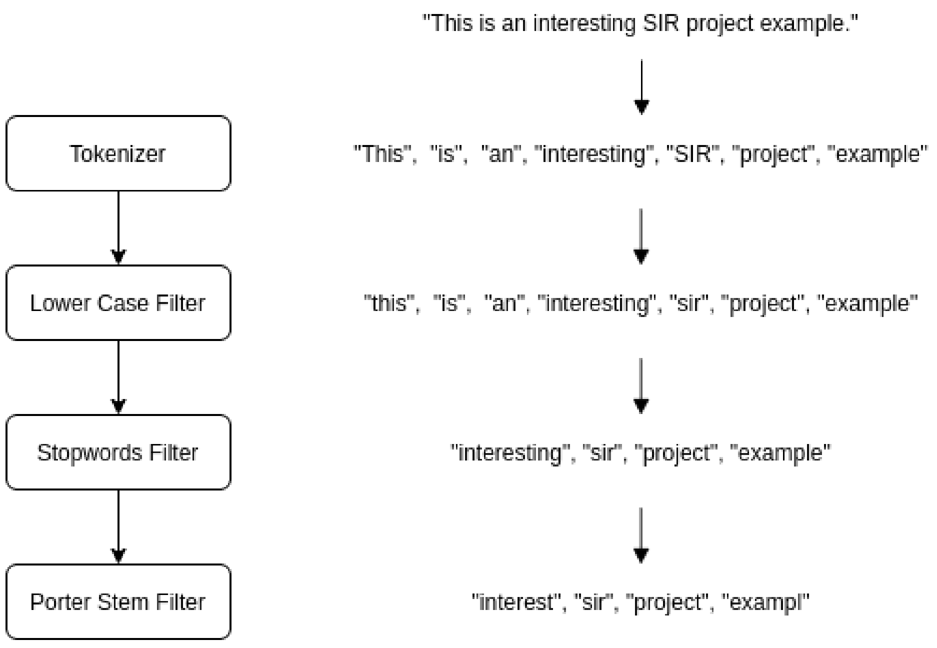
- **docId**: unique identifier of the document, integer;
- **category**: topical area of the article, string;
- **title**: title of the article, string;
- **full_text**: the article, string;
- **summary**: human-generated reference summary, string.

For the string fields it is possible to create a custom field type. Moreover, Solr provides the opportunity to include Analyzers in the definition of a type. An analyzer examines the text of fields and generates a token stream, which is composed of processed words. As an example, an analyzer can transform each word of the document in lowercase format, or it can eliminate common stopwords.

It is then possible to implement a text pre-processing pipeline by simply specifying the operations that the analyzer should apply to the text in a field. The image shows the pipeline and the type of operation chosen for this project.

According to this pipeline, the custom text field is built as follows: a new field type, belonging to the *TextField* class, is injected into the schema, and with a custom analyzer. The following shows the CURL request to create a custom field type, in agreement with Solr's documentation [7]:

```
curl -X POST -H 'Content-type:application/json' --data-binary
    '{
    "add-field-type" : {
     "name":"customTextField",
     "class":"solr.TextField",
     "analyzer" : {
        "tokenizer":{
           "class":"solr.LowerCaseTokenizerFactory" },
        "filters":[{"class":"solr.StopFilterFactory"},
                   {"class":"solr.PorterStemFilterFactory"}]},
     "similarity":{"class":"solr.ClassicSimilarityFactory"}}
}' http://localhost:8984/solr/myDocs/schema
```


"This is an interesting SIR project example."
Tokenizer
"This", "is", "an", "interesting", "SIR", "project", "example"
Lower Case Filter
"this", "is", "an", "interesting", "sir", "project", "example"
Stopwords Filter
"interesting", "sir", "project", "example"
Porter Stem Filter
"interest", "sir", "project", "exampl"

The *LowerCaseTokenizerFactory* implements both the tokenizer and lower case filter operations, while the other filters implement the rest of the pipeline. A **Similarity** child is also specified: the *ClassicSimilarityFactory* allows the retrieval of the classic IR TF-IDF values.

The only non-string field is *docId* and it does not need a custom field type. It is simply added to the schema with the Solr's default integer type, *IntPointField* (field-type: pint).

After custom field types have been defined, the final operation on the schema customization is the injection of custom fields. Field injections are performed in a similar fashion:

```
curl -X POST -H 'Content-type:application/json' --data-binary
    '{
   "add-field":{
     "name":"docId",
     "type":"pint"},
   "add-field":{
     "name":"category",
     "type":"customTextField"},
   "add-field":{
     "name":"title",
     "type":"customTextField"},
   "add-field":{
     "name":"full_text",
     "type":"customTextField"},
   "add-field":{
     "name":"summary",
     "type":"customTextField"}
}' http://localhost:8984/solr/myDocs/schema
```

The schema is now customized and ready to index the dataset's documents. The dataset is in JSON format, and Solr's **post** script for document indexing natively supports it. Therefore the operation is trivial:

```
  bin/post -p 8984 -c myDocs [dataset_dir]/*.json
```

The wildcard * allows the indexing of all the JSON files in the dataset's directory in a single command.

2.3 Solr-Python Interaction

According to Solr documentation:

> *At its heart, Solr is a Web application, but because it is built on open protocols, any type of client application can use Solr. HTTP is the fundamental protocol used between client applications and Solr. The client makes a request and Solr does some work and provides a response. Clients use requests to ask Solr to do things like perform queries or index documents.*

This means that interacting with a Solr instance is simple, and it consists trivially in composing an HTTP request. The choice of Python for the interaction is twofold: it is both simple and the most common language for machine learning algorithms. Using the *request* library, it is possible to create Python wrappers to HTTP requests, and therefore to queries, such as:

```python
import requests
import json

def simple_request():
    req = 'http://localhost:8984/solr/myDocs/select?q=*:*&wt=json'
    results = requests.get(req).json()
    return results
```

This function returns the first 10 documents of the entire collection (Solr limits the output to 10 results, unless otherwise specified in the query with the *rows* command).

2.3.1 Anatomy of a Solr Response

The inspection of the returned value allows for a thorough understanding of Solr's HTTP response:

```
{'responseHeader': {'zkConnected': True,
                     'status': 0,
                     'QTime': 191,
                     'params': {'q': '*:*', 'wt': 'json'}},

'response': {'numFound': 2225,
              'start': 0,
              'numFoundExact': True,
              'docs': [{'docId': 999,
                        'category': 'politics',
                        'title': "Jowell rejects 'Las Vegas'
                           jibe",
                        'full_text': [...],
                        'summary': [...],
                        'id': '79b635f2-5809-4c26-8eb0-64b4c45
                           57604',
                        '_version_': 1694590513119756288},

                        {'docId': 1000, [...] }

                        [...]      ]}}
```

The response is comprised of two data structures:

- **responseHeader**: the HTTP header, which carries no information of interest in the project's context;
- **response**: the body of the response, which contains the documents that meet the query demand.

It is possible to access the single fields of the response with a dictionary-like syntax. For example, the following expression extracts the docId of the first document ('docs' is an array of documents):

```
print(simple_request()['response']['docs'][0]['docId'])

> 999
```

Chapter 3

TF-IDF

The following chapters illustrate the implementation of the chosen extractive summarization techniques, namely the TF-IDF approach and the Machine Learning approach. We begin with the former in the following sections.

3.1 General Functions

Preliminary operations require the definition of support functions for a later use. These functions are general operations that query the Solr instance for specific information. Since they do not belong specifically to the summarization algorithm we define them separately.

```python
#Counts the documents in a category (in the entire
#collection if category not specified)
def count_docs(collection, category="*"):
    r = requests.get('http://localhost:8984/solr/'+collection+'/select?\
        q=category:'+category+'&wt=json').json()

    return r['response']['numFound']


#Returns a document given its docId
def get_doc(collection, docId):
    r = requests.get('http://localhost:8984/solr/'+collection+'/select?\
        q=docId:'+str(docId)+'&wt=json').json()

    return r['response']['docs'][0]
```

The function *count_ docs* trivially returns the number of documents in a category, if specified, in the entire collection if not. The query forwarded to the Solr instance simply searches for documents that belong in a category (*q=category:'+category+'*).

The function *get_ doc* returns a single document, given its docId.

Each function has the mandatory *collection* input argument, that tells Solr the collection on which to run the query on.

Here are some examples that show these functions in action:

```
print(count_docs("myDocs", "politics"))

> 417
```

```
print(get_doc("myDocs", 0))

> {'docId': 0, 'category': 'business', 'title': 'S&N extends Indian beer
↪  venture', 'full_text': [...], 'summary': [...], 'id':
↪  '8c6b6774-974b-4a29-865b-bdf48d35e9ef', '_version_':
↪  1694590517202911232}
```

The following function is an example of a **Stream Query**:

```
#Runs the custom analyzer on a sentence and returns a list of tokens
def analyze(sentence):
    sentenceT = re.sub('[\"\'&#]', ' ', sentence)
    r = requests.get('http://localhost:8984/solr/myDocs/stream?\
        expr=analyze(\"'+sentenceT+'\", full_text)').json()

    return r['result-set']['docs'][0]['return-value']
```

As mentioned before, the Stream query is a powerful tool that allows for a thorough exploration of the document's properties and characteristics. In this case we use it to extract tokens from a sentence.

Stream query responses have a different structure: to obtain the desired value, we must access the field *return-value* of the response. In this case, the desired value is an array of tokens, properly processed according to the pipeline defined in the previous chapter.

The query does in fact require the field on which to run the analyzer. That is because it implements the pre-processing pipeline associated to that field, as defined in the schema. The parameter *expr=analyze(sentenceT, full_text)* asks Solr to extract stopwords-free, lowercase and stemmed tokens from the sentence *sentenceT* (which is simply the sentence passed as an input parameter, stripped of some special characters).

The following shows an example of this function:

```
print(analyze("This is an interesting SIR project example."))

> ['interest', 'sir', 'project', 'exampl']
```

3.2 Algorithm

The goal is to extract the TF-IDF weights of each word and then compute a sentence score based on those weights. Luckily, Solr provides a particular kind of query, called *function query*, that returns different numerical properties of the collection, including TF and IDF weights. The following procedures wrap this kind of query:

```
#Returns the idf value of a term, based on the specified field
def idf(field, term, docId):
   r = requests.get('http://localhost:8984/solr/myDocs/select?\
      q=docId:'+str(docId)+'&fl=idf('+field+','+term+'),*&wt=json').json()

   return r['response']['docs'][0]['idf('+field+','+term+')']


#Returns the tf values of a term, in document with docId
def tf(field, term, docId):
   r = requests.get('http://localhost:8984/solr/myDocs/select?\
      q=docId:'+str(docId)+'&fl=tf('+field+','+term+'),*&wt=json').json()

   return r['response']['docs'][0]['tf('+field+','+term+')']
```

Therefore, given a *docId* and a *field* of the document ("full_text" in our collection), these functions return the TF and IDF values of a *term*. To elaborate a sentence score it is then necessary to extract sentences from a full text, extract processed tokens, and finally calculate TF-IDF scores on those tokens. This process is performed in the following function:

```
def generate_summary(docId):
    doc = get_doc("myDocs", docId)
    sentences = extract_sentences(doc['response']['docs'][0]['full_text'])
    scores = []
    for sentence in sentences:
        score = 0
        analyzedSentence = analyze(sentence)
        for elem in analyzedSentence:
            score+= tf("full_text", elem, docId)*
                    idf("full_text", elem, docId)
        scores.append(score/len(analyzedSentence))

    scores = np.array(scores)
    topIdx = np.sort(np.argsort(scores)[-math.ceil(len(sentences)/3):])
    topValues = [sentences[i] for i in topIdx]
    return (". ".join(topValues)+".")
```

The steps to create a summary are therefore the following:

1. Fetch a document from the collection (*get_ doc("myDocs", docId)*);
2. Extract sentences from the document's *full_ text* (*extract_ sentences* is a support function that transforms a text into an array of sentences, according to syntax rules);
3. For each sentence extracted from the *full_ text*:
   (a) Extract processed tokens with the *analyze* support function;
   (b) Evaluate the TF-IDF score for each token in the sentence;
   (c) Sum all the scores to obtain the sentence score.
4. Sort sentences in decreasing order of their scores;
5. Select the highest ranking sentences.

The number of *top* sentences to extract is an hyperparameter that depends on the document's length. It is not possible to define a predetermined optimal number for it, but it could be set to a fraction of the document's total number of sentences. In this implementation the value was set to 1/3 the total number of sentences:

- *math.ceil(len(sentences)/3)*

Chapter 4

Machine Learning

Machine learning approaches belong to the *Indicator Representation* category, and therefore they deal with the extraction of features from sentences of documents. In this project, three different kinds of features have been extracted:

- **Article-Independent Features**, such as sentence length, or the ratio of nouns to all words;
- **Article-Dependent Features**, such as the position of a sentence in an article.

Each sentence is then labeled with its class: *summary* if that sentence is also in the summary or *not summary* if not. Finally, features and classes are used to train a classifier. SVM, Decision Tree, and Naive Bayes were tested in this chapter.

4.1 Introduction

In order to extract relevant features from sentences in an easy and more immediate way (and also to test a different kind of Solr instance), it was deemed necessary to reshape the document collection. Specifically, another collection called *mySentences* was created, where each "document" represents a single sentence instead of the entire text.

As a result, the documents have the following new structure:

- **docId**: identifier of the article, integer;
- **category**: topical area of the article, string;
- **title**: title of the article, string;

- **sentence**: a sentence of the document, string;
- **summary**: whether or not the sentence is part of the summary, boolean.

The schema for the new collection must be modified accordingly, but the only difference from the earlier schema is in the summary field, which is now a boolean value. A conceptual difference is instead the *docId* field: whereas before it represented a unique identifier for each document, now it represents the identifier of the article the sentence belongs to. Different sentences can therefore have the same *docId*, meaning that they belong to the same news article.

The collection thus created is composed of documents (sentences) which already hold the **class** value needed for the training of the classifiers. Furthermore, we can easily query single sentences for feature extraction.

4.2 Feature Extraction

As a preliminary step for feature extraction, a support function that extracts information from the set of sentences that form a single article was devised. This function, called *get_article_info*, thanks to Solr's Streaming expression, extracts, given a *docId*:

- **title**: the title of the article corresponding to that *docId*;
- **sentences**: the sentences that compose an article;
- **classes**: the classes of the sentences;
- **sentenceTerms**: the processed tokens of all the sentences;
- **weightedVectors**: the weighted vectors of the sentences (TF-IDF weights);
- **dictionary**: the dictionary of the article.

Except for *dictionary* and *title*, the other values are all arrays with size equal to the number of sentences in the examined article.

```
def get_article_info(docId):
    expression= "let(echo=\"a, b, c, d\",\
    a=select(search(mySentences,\
    q=\"docId:"+str(docId)+"\", fl=\"sentence\", rows=1000), \
    analyze(sentence, sentence) as terms),\
    b=termVectors(a, minTermLength=0, minDocFreq=0, maxDocFreq=1),\
    c=getColumnLabels(b), \
    d=search(mySentences, q=\"docId:"+str(docId)+"\", fl=\"sentence,\
            title, summary\", rows=1000))"

    response = requests.get('http://localhost:8984/solr/mySentences\
               /stream?expr='+expression).json()

    dictionary = response['result-set']['docs'][0]['c']
    sentenceTerms = response['result-set']['docs'][0]['a']
    weightedVectors = response['result-set']['docs'][0]['b']
    title = response['result-set']['docs'][0]['d'][0]['title']

    sentences = []
    classes = []
    for elem in response['result-set']['docs'][0]['d']:
        sentences.append(elem['sentence'])
        classes.append(elem['summary'])

    return {"title":title, "sentences":sentences, "classes": classes,
    "sentenceTerms":sentenceTerms, "weightedVectors":weightedVectors,
    "dictionary":dictionary}
```

The Stream query expression is particularly interesting: it asks Solr for different kinds of information in a single request. Namely, it handles a single article as a mini-collection composed of sentences (in **a** we query Solr for sentences that satisfy the *docId* parameter). It evaluates TF-IDF values on this mini-collection (**b**) and extracts its vocabulary (**c**). Finally, **d** extracts the article title and summary boolean values.

In other words, we are extracting the so called **Term Frequency - Inverse Sentence Frequency** values (TF-ISF) from the sentences of a single article. These are basically TF-IDF weights but computed on a sentence/article level, rather than on a document/collection level.

TF-ISF weights are one of the Article-Dependent features used in the training process.

The following are output examples of the *get_article_info* function:

```
print(get_article_info(0)['title'])

> 'S&N extends Indian beer venture'
```

```
print(get_article_info(0)['sentences'])

> ['Kingfisher has market share of about 29\%.',
'In addition to the equity stake S&N is to invest 2,47bn rupees in\
United through non-convertible redeemable preference shares.',
"Meanwhile, United's budget airline, Kingfisher Airlines,\
is to buy 10 A320 aircraft from Airbus and has the option\
to buy 20 more aircraft in a deal worth up to \$1,8bn.", ... ]
```

```
print(get_article_info(0)['classes'])

> [False, True, True, False, False, True, True,
False, False, False, True, False, True, False]
```

4.2.1 Article-Independent Features

Features that belong in this category are:

- Sentence length (the number of its terms);
- Presence of proper nouns (boolean);
- Ratio of nouns to all words in the sentence;
- Ratio of verbs to all words in the sentence;
- Ratio of adjectives to all words in the sentence;
- Ratio of adverbs to all words in the sentence;

The functions that extract features work in batches: they operate on an entire article and therefore on a set of sentences. This means that we evaluate sentence lengths of an entire article in a single request:

```
def sentence_lengths(sentenceTerms):
    weights = []
    for terms in sentenceTerms:
        weights.append(len(terms['terms']))
    return weights
```

An example:

```
print(sentence_lengths(get_article_info(0)['sentenceTerms']))

> [5, 15, 20, 10, 12, 20, 18, 8, 15, 14, 17, 11, 16, 13]
```

The remaining Article-Independent features are all Part-Of-Speech tagging features. Luckily, Python has a library that implements POS tagging (**nltk**), therefore we can group the POS feature extractors in one function:

```
def pos_tagging_features(sentence):
    sentenceT = re.sub('[\"\'&#]', ' ', sentence)
    r = requests.get('http://localhost:8984/solr/mySentences/stream?\
                expr=analyze(\"'+sentenceT+'\", simpleTokenizer)').json()
    tagged = nltk.pos_tag(r['result-set']['docs'][0]['return-value'])

    properNouns = [word for (word, pos) in tagged if (pos == 'NNP' or pos
    ↪  == 'NNPS')]
    nouns = [word for (word, pos) in tagged if (pos == 'NN' or pos ==
    ↪  'NNS' or pos == 'NNP' or pos == 'NNPS')]
    verbs = [word for (word, pos) in tagged if (pos == 'VB' or pos ==
    ↪  'VBG' or pos == 'VBD' or pos == 'VBN' or pos == 'VBP' or pos ==
    ↪  'VBZ')]
    adjectives = [word for (word, pos) in tagged if (pos == 'JJ' or pos
    ↪  == 'JJR' or pos == 'JJS')]
    adverbs = [word for (word, pos) in tagged if (pos == 'RB' or pos ==
    ↪  'RBR' or pos == 'RBS')]

    if not properNouns:
        proper = False
    else: proper = True

    if tagged:
        nounRatio = len(nouns)/len(tagged)
        verbRatio = len(verbs)/len(tagged)
        adjectiveRatio = len(adjectives)/len(tagged)
        adverbRatio = len(adverbs)/len(tagged)
    else: nounRatio = verbRatio = adjectiveRatio = adverbRatio = 0.0

    return {"proper":proper, "nounRatio":nounRatio,
            "verbRatio":verbRatio, "adjectiveRatio":adjectiveRatio,
            "adverbRatio":adverbRatio}
```

The stream query in this function performs an analysis on the custom field *simpleTokenizer*. This field simply extracts tokens verbatim from the sentences, while still eliminating stopwords. A stemming operation was omitted since it would have caused the POS tagger to fail in its goal and uppercase words/letters were preserved since they are needed for the extraction of proper nouns.

```
print(pos_tagging_features("This is an interesting SIR project example,
↪  explaining POS features."))

> {'proper': True, 'nounRatio': 0.625, 'verbRatio': 0.125,
↪  'adjectiveRatio': 0.125, 'adverbRatio': 0.0}
```

4.2.2 Article-Dependent Features

Features belonging in this category are:

- **TF-ISF value**: the sum of the TF-ISF scores of the sentence's words, normalized by the number of total words in the sentence;
- **Sentence Position**: the position of the sentence in the article, normalized to take on values between 0 and 1;
- **Title Similarity**: number of term co-occurrences between the sentence and the article title, normalized by the number of total words in the sentence;
- **Sentence-to-Sentence Cohesion**: for each sentence s we first compute the similarity between s and each other sentence s' of the article; then we add up those similarity values, obtaining the raw value of this feature for s; the process is repeated for all sentences. The normalized value (in the range [0, 1]) of this feature for a sentence s is obtained by computing the ratio of the raw feature value for s over the largest raw feature value among all sentences in the document. Values closer to 1.0 indicate sentences with larger cohesion;

- **Sentence-to-Centroid Cohesion**: we compute the vector representing the centroid of the document, which is the arithmetic average over the corresponding coordinate values of all the sentences of the document; then we compute the similarity between the centroid and each sentence, obtaining the raw value of this feature for each sentence. The normalized value in the range [0, 1] for s is obtained by computing the ratio of the raw feature value over the largest raw feature value among all sentences in the document. Sentences with feature values closer to 1.0 have a larger degree of cohesion with respect to the centroid of the document, and so are supposed to better represent the basic ideas of the document.

The implementation of these feature extractors is the following:

```python
def sentence_positions(sentences):
    nSent = len(sentences)
    return [x/nSent for x in list(range(0, nSent))]
```

As mentioned before, sentences are parsed in batches: extracting sentence position is therefore trivial, since the *sentences* input parameter refers to the entirety of an article's full text (which is already ordered).

```python
def title_similarities(sentences, title):
    weights = []
    titleTerms = analyze(title)
    for sentence in sentences:
        sentenceTerms = analyze(sentence)
        try:
            commonTerms = len([x for x in sentenceTerms if x in
            ↪  titleTerms])/len(sentenceTerms)
        except ZeroDivisionError:
            commonTerms = 0.0
        weights.append(commonTerms)
    return weights
```

```python
def sent_to_sent_cohesion(weightedVectors):
    rawValues = []
    for i in range(0, len(weightedVectors)):
        score = 0
        for j in range(0, len(weightedVectors)):
            if j!=i:
                score += cosine_similarity(weightedVectors[i],
                ↪  weightedVectors[j])
        rawValues.append(score)
    scores = [element/max(rawValues) for element in rawValues]
    return scores
```

```
def sent_to_centroid_cohesion(weightedVectors):
    centroid = np.zeros(len(weightedVectors[0]))
    for vector in weightedVectors:
        centroid += np.array(vector)
    centroid = (centroid/len(weightedVectors)).tolist()
    rawValues = []
    for vector in weightedVectors:
        score = cosine_similarity(vector, centroid)
        rawValues.append(score)

    scores = [element/max(rawValues) for element in rawValues]
    return scores
```

4.2.3 Feature Matrix

Finally, we compute the features for each sentence of a single article:

```
def extract_features(docId):
    doc = get_doc_info(docId)
    sentences = doc['sentences']
    classes = doc['classes']
    sentPos = sentence_positions(sentences)
    titleSim = title_similarities(sentences, doc['title'])
    sentLen = sentence_lengths(doc['sentenceTerms'])
    stsCoh = sent_to_sent_cohesion(doc['weightedVectors'])
    stcCoh = sent_to_centroid_cohesion(doc['weightedVectors'])

    features = []
    for i in range(0, len(sentences)):
        generalFeatures = {"sentence":sentences[i], "docId":docId,
        ↪  "sentPos":sentPos[i],
                            "titleSim":titleSim[i], "sentLen":sentLen[i],
                            "stsCoh":stsCoh[i], "stcCoh":stcCoh[i]}
        posFeatures = pos_tagging_features(sentences[i])
        classSumm = {"class": classes[i]}
        features.append({**generalFeatures, **posFeatures, **classSumm})

    return features
```

To extract features for the entire dataset, we iterate this function over the entire collection.

4.3 Training

As mentioned before, the classifiers used for training are SVM, Decision Tree, and Naive Bayes. The Python library Scikit-Learn has all the tools needed. The feature matrix is split in training and test sets (the raw text and the *docId* are stripped since they play no discriminating role).

```
from sklearn import svm, naive_bayes, metrics
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

#Split dataset into training set and test set
X_train, X_test, y_train, y_test = train_test_split(data, target,
↪  test_size=0.3,random_state=50, stratify=target)

#Define the model
modelSVM = svm.NuSVC()
#Train the model
modelSVM.fit(X_train, y_train)
#Evaluate the model
y_predSVM = modelSVM.predict(X_test)
print("SVM Accuracy:",metrics.accuracy_score(y_test, y_predSVM))

modelNB = naive_bayes.GaussianNB()
modelNB.fit(X_train, y_train)
y_predNB = modelNB.predict(X_test)
```

```
print("NB Accuracy:",metrics.accuracy_score(y_test, y_predNB))

modelDT = DecisionTreeClassifier()
modelDT.fit(X_train, y_train)
y_predDT = modelDT.predict(X_test)
print("DT Accuracy:",metrics.accuracy_score(y_test, y_predDT))
```

The *Accuracy* measure used in this section is not final nor correct. It is only a tentative metric used to fine tune the classifiers. The following chapter, *Evaluation*, provides a deeper dive on the metrics used to evaluate the algorithms.

Chapter 5

Evaluation

The final step of the project is a formal evaluation of the proposed algorithms. As stated in chapter 1, the metrics most commonly used are variations of ROUGE.

In order to compare the two algorithms, and to estimate their effectiveness, the summaries for all the articles in the dataset are generated automatically. Then, the metrics for each article are computed. Finally, the comparison occurs on the average ROUGE metrics of all articles.

5.1 ROUGE Details

As mentioned before, ROUGE metrics come in many variants, with ROUGE-N and ROUGE-L being the most common.

ROUGE-N

ROUGE-N is an n-gram recall between a candidate summary and a reference summary and is computed as follows: a series of N-grams (N equal to 2, 3, or rarely 4) is extracted from the reference summaries and the candidate summary (automatically generated summary).

$$\text{ROUGE-N} = \frac{p}{q}$$

where p is the number of common N-grams between candidate and reference summary, and q is the number of N-grams extracted from the reference summary only.

It is a recall oriented metric because the denominator refers to the reference summary. Variations are possible, such as precision-oriented ROUGE-

N, in which the denominator is the number of N-grams extracted from the candidate summary, or F-Measure ROUGE-N that combines both.

ROUGE-L

ROUGE-L is a metric based on the Longest Common Subsequence of two text, namely candidate and reference summary. A sequence $Z = [z_1, z_2, \ldots, z_n]$ is a subsequence of another sequence $X = [x_1, x_2, \ldots, x_m]$, if there exists a strict increasing sequence $[i_1, i_2, ..., i_k]$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$. Given two sequences X and Y, the longest common subsequence (LCS) of X and Y is a common subsequence with maximum length.

Given a reference summary of u sentences containing a total of m words and a candidate summary of v sentences containing a total of n words, three LCS-based ROUGE scores (recall, precision, F-measure) can be computed as follows:

$$R_{lcs} = \frac{\sum_{i=1}^{u} LCS_U(r_i, C)}{m}$$

$$P_{lcs} = \frac{\sum_{i=1}^{u} LCS_U(r_i, C)}{n}$$

$$F_{lcs} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}}$$

where $LCS_U(r_i, C)$ is the LCS score of the *union* longest common subsequence between reference sentence r_i and candidate summary C.

The Python library *rouge-score*[1] provides the implementation of all the above metrics (and for several values of N). For example:

```
from rouge_score import rouge_scorer
scorer = rouge_scorer.RougeScorer(['rouge2', 'rougeLsum'])
scores = scorer.score('This is an interesting SIR project example.',
                      'This is an interesting example.')
print(scores)

> "{'rouge2': Score(precision=0.75, recall=0.5, fmeasure=0.6),
↪  'rougeLsum': Score(precision=1.0, recall=0.7142857142857143,
↪  fmeasure=0.8333333333333333)}"
```

[1] https://pypi.org/project/rouge-score/

5.2 Summaries Generation

Through a simple *for* loop on the *docId* field, it is possible to iterate the automatic summarization algorithm on all the documents in the collection.

However, for the sake of brevity, it was decided to limit the analysis on a subset of the documents. Therefore only the first 100 documents (docId: 0-99) will be considered.

The following code computes the summaries using the TF-IDF algorithm:

```
summaries = []
for i in range(0, 100):
    summaries.append({"docId":i, "hypotesis":generate_summary(i),
    ↪  "reference":get_doc("myDocs", i)['summary'] })
```

The generation of Machine Learning summaries is different. In order to compare the two algorithms, we must compare summaries of the same documents. Therefore we split the feature matrix in more precise training and test sets: the test set will contain sentences of the first 100 documents (the same as the TF-IDF summaries) and the training set will contain the rest. The training process remains unchanged.

The following code combines the prediction of all the sentences in the test set in a single dictionary (*results*) and then generates three summaries for each document (using *docID* to discriminate the documents):

```
results = []
for i in range(0, len(dataTest)):
    results.append({'docId':int(dataTest[i][0]),
    ↪  'sentence':sentencesTest[i],
                    'SVM':y_predSVM[i], 'NB':y_predNB[i],
                    ↪  'DT':y_predDT[i]})

summariesSVM = []
for i in range(0, 100):
    doc = [d for d in results if d['docId'] == i]
    summariesSVM.append({'docId':i, 'hypotesis':" ".join([d['sentence']
    ↪  for d in doc if d['SVM'] == 1]), 'reference': get_doc('myDocs',
    ↪  i)['summary']})

summariesNB = []
for i in range(0, 100):
    doc = [d for d in results if d['docId'] == i]
    summariesNB.append({'docId':i, 'hypotesis':" ".join([d['sentence']
    ↪  for d in doc if d['NB'] == 1]), 'reference': get_doc('myDocs',
    ↪  i)['summary']})

summariesDT = []
for i in range(0, 100):
    doc = [d for d in results if d['docId'] == i]
    summariesDT.append({'docId':i, 'hypotesis':" ".join([d['sentence']
    ↪  for d in doc if d['DT'] == 1]), 'reference': get_doc('myDocs',
    ↪  i)['summary']})
```

Then, to compare reference and candidate:

```
hyp = summariesNB[0]['hypotesis']
ref = summariesNB[0]['reference']

scorer = rouge_scorer.RougeScorer(['rouge2', 'rougeLsum'])
print(scorer.score(hyp, ref))

> "{'rouge2': Score(precision=0.9875776397515528,
↪  recall=0.7910447761194029, fmeasure=0.878453038674033), 'rougeLsum':
↪  Score(precision=0.6666666666666666, recall=0.5346534653465347,
↪  fmeasure=0.5934065934065934)}"
```

5.3 Comparisons

To appreciate the results of the algorithm, we can observe the generated summaries of each algorithm. As an example, consider the following document:

> *MG Rover's proposed tie-up with China's top carmaker has been delayed due to concerns by Chinese regulators, according to the Financial Times. The paper said Chinese officials had been irritated by Rover's disclosure of its talks with Shanghai Automotive Industry Corp in October. The proposed deal was seen as crucial to safeguarding the future of Rover's Longbridge plant in the West Midlands. However, there are growing fears that the deal could result in job losses. The Observer reported on Sunday that nearly half the workforce at Longbridge could be under threat if the deal goes ahead. Shanghai*

> *Automotive's proposed Â£1bn investment in Rover is awaiting approval by its owner, the Shanghai city government and by the National Development and Reform Commission, which oversees foreign investment by Chinese firms. According to the FT, the regulator has been annoyed by Rover's decision to talk publicly about the deal and the intense speculation which has ensued about what it will mean for Rover's future. As a result, hopes that approval of the deal may be fast-tracked have disappeared, the paper said. There has been continued speculation about the viability of Rover's Longbridge plant because of falling sales and unfashionable models. According to the Observer, 3,000 jobs - out of a total workforce of 6,500 - could be lost if the deal goes ahead. The paper said that Chinese officials believe cutbacks will be required to keep the MG Rover's costs in line with revenues. It also said that the production of new models through the joint venture would take at least eighteen months. Neither Rover nor Shanghai Automotive commented on the reports.*

and consider its **reference** summary:

> *The paper said Chinese officials had been irritated by Rover's disclosure of its talks with Shanghai Automotive Industry Corp in October. According to the FT, the regulator has been annoyed by Rover's decision to talk publicly about the deal and the intense speculation which has ensued about what it will mean for Rover's future. The proposed deal was seen as crucial to safeguarding the future of Rover's Longbridge plant in the West Midlands. According to the Observer, 3,000 jobs - out of a total workforce of 6,500 - could be lost if the deal goes ahead. The paper said that Chinese officials believe cutbacks will be required to keep the MG Rover's costs in line with revenues. As a result, hopes that approval of the deal may be fast-tracked have disappeared, the paper said.*

We can compare it to the **SVM** summary:

> *The paper said Chinese officials had been irritated by Rover's disclosure of its talks with Shanghai Automotive Industry Corp in October. According to the FT, the regulator has been annoyed by Rover's decision to talk publicly about the deal and the intense speculation which has ensued about what it will mean for Rover's future.*

to the **Naive Bayes** summary:

> *MG Rover's proposed tie-up with China's top carmaker has been delayed due to concerns by Chinese regulators, according to the Financial Times. The paper said Chinese officials had been irritated by Rover's disclosure of its talks with Shanghai Automotive Industry Corp in October. The proposed*

> *deal was seen as crucial to safeguarding the future of Rover's Longbridge plant in the West Midlands. Shanghai Automotive's proposed Â£1bn investment in Rover is awaiting approval by its owner, the Shanghai city government and by the National Development and Reform Commission, which oversees foreign investment by Chinese firms. According to the FT, the regulator has been annoyed by Rover's decision to talk publicly about the deal and the intense speculation which has ensued about what it will mean for Rover's future. There has been continued speculation about the viability of Rover's Longbridge plant because of falling sales and unfashionable models. The paper said that Chinese officials believe cutbacks will be required to keep the MG Rover's costs in line with revenues.*

to the **Decision Tree** summary:

> *MG Rover's proposed tie-up with China's top carmaker has been delayed due to concerns by Chinese regulators, according to the Financial Times. The paper said Chinese officials had been irritated by Rover's disclosure of its talks with Shanghai Automotive Industry Corp in October. The proposed deal was seen as crucial to safeguarding the future of Rover's Longbridge plant in the West Midlands. The Observer reported on Sunday that nearly half the workforce at Longbridge could be under threat if the deal goes ahead. According to the FT, the regulator has been annoyed by Rover's decision to talk publicly about the deal and the intense speculation which has ensued about what it will mean for Rover's future. According to the Observer, 3,000 jobs - out of a total workforce of 6,500 - could be lost if the deal goes ahead.*

and finally to the **TF-IDF** summary:

> *The paper said Chinese officials had been irritated by Rovers disclosure of its talks with Shanghai Automotive Industry Corp in October. The proposed deal was seen as crucial to safeguarding the future of Rovers Longbridge plant in the West Midlands. The Observer reported on Sunday that nearly half the workforce at Longbridge could be under threat if the deal goes ahead. There has been continued speculation about the viability of Rovers Longbridge plant because of falling sales and unfashionable models. Neither Rover nor Shanghai Automotive commented on the reports.*

Furthermore, values of ROUGE-1, ROUGE-2, ROUGE-3, and ROUGE-L were computed for the first 100 automatic generated summaries, both for the TF-IDF algorithm and the Machine Learning algorithms. The values were then averaged over the 100 documents, and the following tables were produced:

	ROUGE-1			**ROUGE-2**		
	P	R	F	P	R	F
TF-IDF	0.5776	0.7614	0.6537	0.4901	0.6470	0.5551
SVM	0.0507	0.2552	0.0821	0.0481	0.2519	0.0784
NB	0.8695	0.7152	0.7669	0.8158	0.6684	0.7178
DT	0.7155	0.6944	0.6831	0.6259	0.6071	0.5965

Table 5.1: ROUGE-1 and ROUGE-2 mean values of the first 100 automatic generated summaries

	ROUGE-3			**ROUGE-L**		
	P	R	F	P	R	F
TF-IDF	0.4562	0.6036	0.5172	0.4019	0.5301	0.4549
SVM	0.0463	0.2508	0.0759	0.0477	0.2476	0.0778
NB	0.7859	0.6432	0.6908	0.5597	0.4658	0.4963
DT	0.5937	0.5766	0.5658	0.4690	0.4632	0.4501

Table 5.2: ROUGE-3 and ROUGE-L mean values of the first 100 automatic generated summaries

From these result we could argue that TF-IDF and Naive Bayes have comparable results and outperform the other algorithms. This is a naive conclusion since other statistics should be extracted (such as standard deviation) to thoroughly compare the algorithms. Furthermore, we can safely assume that the SVM algorithm does non perform properly and probably needs more careful tuning.

5.4 Conclusions

The standard, and arguably simpler, approach of the TF-IDF algorithm has proven to yield interesting results, comparable to the more sophisticated Naive Bayes' classifier. Still, the classifier were not carefully tuned, and therefore a proper setting of the hyperparameters would hopefully produce better results.

However, that falls outside the goals of the project and thus it is postponed to future revisions. Other improvements that would benefit the outcomes of the algorithms are:

- Inclusion of *semantic* features in both algorithms;
- Extraction of more features for the classifiers;
- Using other algorithms.

Bibliography

[1] Allahyari, Mehdi, et al. "Text summarization techniques: a brief survey." *arXiv preprint arXiv:1707.02268* (2017).

[2] Abhijit, Roy. "Understanding Automatic Text Summarization-1: Extractive Methods." *Towards Data Science*, Medium, 5 Ago 2020, https://towardsdatascience.com/understanding-automatic-text-summarization-1-extractive-methods-8eb512b21ecc.

[3] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries" *University of Southern California* (2004).

[4] Joel Larocca, Neto Alex A. Freitas, Celso A. A. Kaestner. "Automatic Text Summarization using a Machine Learning Approach." *Pontifical Catholic University of Parana (PUCPR)* (2002).

[5] Alexander Dlikman, Mark Last. "Using Machine Learning Methods and Linguistic Features in Single-Document Extractive Summarization." *Ben-Gurion University of the Negev* (2016).

[6] Sharif, Pariza. "BBC News Summary." *Extractive Summarization of BBC News Articles*, Kaggle, 4 May 2018, https://www.kaggle.com/pariza/bbc-news-summary

[7] "Apache Solr Reference Guide," *The Apache Software Foundation*, 28 Jan 2021, https://solr.apache.org/guide/8_8/index.html

[8] Ganesan, Kavita. "What is ROUGE and how it works for evaluation of summaries?" *AI Implementation, ROUGE, Text Mining Concepts, Text Summarization*, https://kavita-ganesan.com/what-is-rouge-and-how-it-works-for-evaluation-of-summaries/#.YHFUl3UzZH6