FACULTÉ DES SCIENCES DHAR EL MAHRAZ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH

Conceptual Foundations of Text Mining and Preprocessing Steps

El Habib NFAOUI (elhabib.nfaoui@usmba.ac.ma)
LIIAN Laboratory, Faculty of Sciences Dhar Al Mahraz, Fes
Sidi Mohamed Ben Abdellah University, Fes

2020-2021

Outline

1. Introduction
2. Syntax versus Semantics
3. A General Framework for Text Analytics
4. n-grams
5. Deep Learning in NLP: an overview

1. Introduction

Before starting a text mining project, it is first necessary to understand the conceptual foundations of text mining and then to understand how to get started leveraging the power of your data to drive decision making in your organization. This chapter highlights many of the theoretical foundations of text mining algorithms and describes the basic preprocessing steps used to prepare data for text mining algorithms.

The majority of text data encountered on a daily basis is *unstructured*, or free, text. This is the most familiar type of text, appearing in everyday sources such as books, newspapers, emails, and web pages. By unstructured, we mean that this text exists “in the wild” and has not been processed into a structured format, such as a spreadsheet or relational database. Often, text may occur as a field in a spreadsheet or database alongside more traditional structured data. This type of text is called *semistructured*. Unstructured and semistructured text composes the majority of the world’s data.

Given the sheer volume of text available, it is necessary to turn to **automated means** to assist humans in understanding and exploiting available text documents. This chapter provides a brief introduction to the statistical and linguistic foundations of text mining and walks through the process of preparing unstructured and semistructured text for use with text mining algorithms and software.

2. Syntax versus Semantics

The purpose of text mining algorithms is to provide some understanding of how the text is processed without having a human read it.

Syntax **pertains to the structure of language and how individual words are composed to make well-formed sentences and paragraphs**. This is an ordered process. Specific grammar rules and language conventions govern how language is used, leading to statistical patterns appearing frequently in large amounts of text. This structure is relatively easy for a computer to process. On its own, **however, syntax is insufficient for fully understanding meaning**.

Semantics **refers to the meaning of the individual words within the surrounding context**. Common idioms are useful to illustrate the differences between syntax and semantics. Idioms are words or phrases with a figurative meaning that is different from the literal meaning of the words. For example, the sentence “Mary had butterflies in her stomach before the show” is syntactically correct and has two potential semantic meanings: a **literal interpretation** where Mary’s preshow ritual includes eating butterflies and an **idiomatic interpretation** that Mary was feeling nervous before the show. **Clearly, the complete semantic meaning of the text can be difficult to determine automatically without extensive understanding of the language being used.**

2. Syntax versus Semantics

Fortunately, syntax alone can be used to extract practical value from the text without a full semantic understanding. Many text mining tasks, such as document classification and clustering, are concerned with finding specific types of documents in a large document database. Though relying on syntactic information alone, these approaches work because documents that share many keywords are often on the same topic. In other instances, **the goal is really about semantics**. For example, **concept extraction** is about automatically identifying words and phrases that have the same meaning.

3. A General Framework for Text Analytics

- A traditional text analytics framework consists of three consecutive phases: Text Preprocessing, Text Representation and Knowledge Discovery, shown in Figure below.

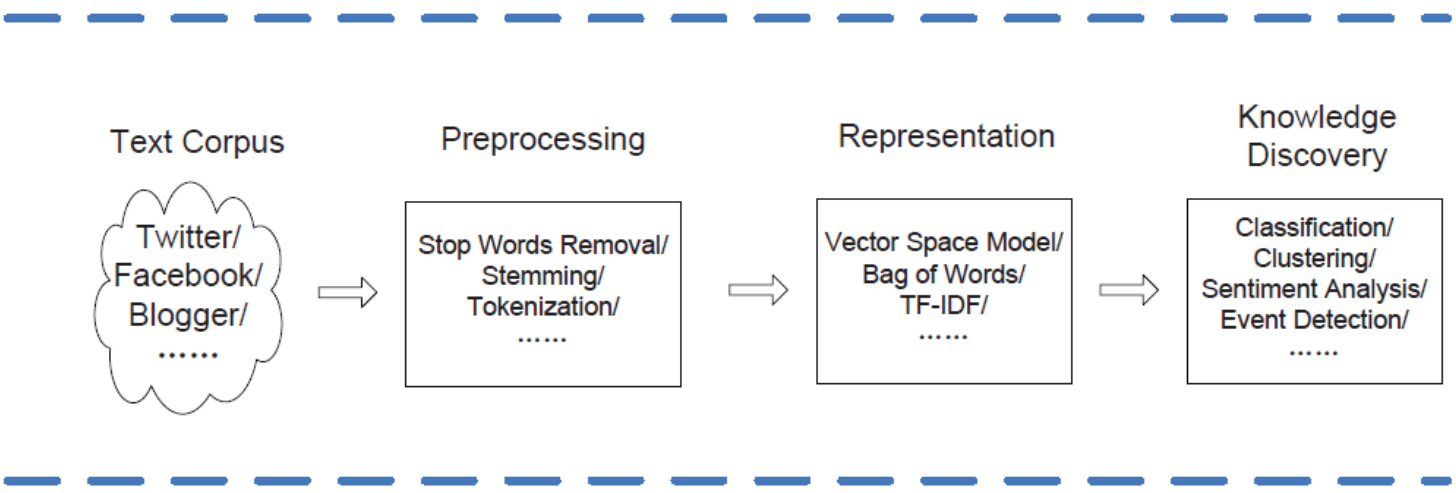

Figure 1. A Traditional Framework for Text Analytics (Xia Hu and Huan Liu, 2012)

3. A General Framework for Text Analytics

- **Text Preprocessing**

  Text preprocessing aims to make the input documents more consistent to facilitate text representation, which is necessary for most text analytics tasks. The text data is usually preprocessed to remove parts that do not bring any relevant information.

- **Text Representation**

  After text preprocessing has been completed, the individual word tokens must be transformed into a vector representation suitable for input into text mining algorithms.

- **Knowledge Discovery**

  When we successfully transform the text corpus into numeric vectors, we can apply the existing machine learning or data mining methods like classification or clustering.

By conducting text preprocessing, text representation and knowledge discovery methods, we can mine **latent**, **useful information** from the input text corpus, like similarity between two messages from social media.

3.1 Text Preprocessing

- The possible steps of text preprocessing are the same for all text mining tasks, though which processing steps are chosen depends on the task. The ways to process documents are so varied and application- and language-dependent. The basic steps are as follows:

1. Choose the scope of the text to be processed (documents, paragraphs, etc.).

   Choosing the proper scope depends on the goals of the text mining task: for *classification* or *clustering* tasks, often the entire document is the proper scope; for *sentiment analysis*, *document summarization*, or *information retrieval*, smaller units of text such as paragraphs or sections might be more appropriate.

2. Tokenize: Break text into discrete words called tokens → Transform text into a list of words (tokens).

3. Remove stopwords ("stopping"): remove all the stopwords, that is, all the words used to construct the syntax of a sentence but not containing text information (such as conjunctions, articles, and prepositions) such as a, about, an, are, as, at, be, by, for, from, how, will, with, and many others.

3.1 Text Preprocessing

4. Stem: Remove prefixes and suffixes to normalize words - for example, *run, running,* and *runs* would all be stemmed to run. So the words with variant forms can be regarded as same feature. Many algorithms have been invented to do stemming (Porter, Snowball, and Lancaster). It also depends on the language. Notice that **lemmatization** can be used instead stemming, it depends on the text mining subtasks and corpus language.

5. Normalize spelling: Unify misspellings and other spelling variations into a single token.

6. Detect sentence boundaries: Mark the ends of sentences.

7. Normalize case: Convert the text to either all lower or all upper case.

3.1 Text Preprocessing

Difference between Stemming and Lemmatization

- Stemming and lemmatization both of these concepts are used to normalize the given word by removing infixes and consider its meaning. The major difference between these is as shown:
  - Stemming:
  1. Stemming usually operates on single word without knowledge of the context.
  2. In stemming, we do not consider POS (Part-of-speech) tags.
  3. Stemming is used to group words with a similar basic meaning together.

  - Lemmatization :
  1. Lemmatization usually considers words and the context of the word in the sentence.
  2. In lemmatization, we consider POS tags.

3.1 Text Preprocessing

- Preprocessing methods depend on specific application. In many applications, such as Opinion Mining or Natural Language Processing (NLP), they need to analyze the message from a syntactical point of view, which requires that the method retains the original sentence structure. Without this information, it is difficult to distinguish "Which university did the president graduate from?" and "Which president is a graduate of Harvard University?", which have overlapping vocabularies. In this case, we need to avoid removing the syntax-containing words.

3.1 Text Preprocessing

As an example, the following Python code preprocesses a sample text using the techniques described previously.

```
## Example of corpus-raw text preprocessing
import nltk
nltk.download('stopwords') # Downloading and updating package stopwords
from nltk.corpus import stopwords
from nltk.tokenize import WordPunctTokenizer
from nltk.stem.porter import PorterStemmer

# RegexpTokenizer class: A RegexpTokenizer splits a string into substrings using a regular expression.
from nltk.tokenize import RegexpTokenizer

stemmer = PorterStemmer()

raw_text = """ The next preprocessing step is breaking up the units of text into individual words or tokens. This process
            can take many forms, depending on the language being analyzed. For English, a straightforward and effective
            tokenization strategy is to use white space and punctuation as token delimiters."""
#Lowercase conversion
text = raw_text.lower()
print("text :", text)
print("\n")

# Tokenization using RegexpTokenizer
tokenizer = RegexpTokenizer(r'\w+') # remove also punctuation mark
tokens = tokenizer.tokenize(text)
print("tokens :", tokens)
print("\n")
print("tokens[0] : ", tokens[0])

# Stop word removal
tokens_clean = [t for t in tokens if t not in stopwords_en]
print("\n tokens_clean : ",tokens_clean)

# Stemming using PorterStemmer()
stems = [stemmer.stem(t) for t in tokens_clean]
print("\n stems : ", stems)
```

3.1 Text Preprocessing

The output of the preceding code snippet is as :

```
text :  the next preprocessing step is breaking up the units of text into individual words or tokens. this process
            can take many forms, depending on the language being analyzed. for english, a straightforward and effective
            tokenization strategy is to use white space and punctuation as token delimiters.


tokens : ['the', 'next', 'preprocessing', 'step', 'is', 'breaking', 'up', 'the', 'units', 'of', 'text', 'into', 'individual',
'words', 'or', 'tokens', 'this', 'process', 'can', 'take', 'many', 'forms', 'depending', 'on', 'the', 'language', 'being', 'ana
lyzed', 'for', 'english', 'a', 'straightforward', 'and', 'effective', 'tokenization', 'strategy', 'is', 'to', 'use', 'white',
'space', 'and', 'punctuation', 'as', 'token', 'delimiters']


tokens[0] :  the

 tokens_clean :  ['next', 'preprocessing', 'step', 'breaking', 'units', 'text', 'individual', 'words', 'tokens', 'process', 'ta
ke', 'many', 'forms', 'depending', 'language', 'analyzed', 'english', 'straightforward', 'effective', 'tokenization', 'strateg
y', 'use', 'white', 'space', 'punctuation', 'token', 'delimiters']

 stems :  ['next', 'preprocess', 'step', 'break', 'unit', 'text', 'individu', 'word', 'token', 'process', 'take', 'mani', 'for
m', 'depend', 'languag', 'analyz', 'english', 'straightforward', 'effect', 'token', 'strategi', 'use', 'white', 'space', 'punct
uat', 'token', 'delimit']
```

3.2 Text Representation: Bag of Words and Vector Space Models

The most popular structured representation of text is the **vector-space model**, which represents every document (text) from the corpus as a vector whose length is equal to the vocabulary of the corpus. This results in an extremely high-dimensional space; typically, every distinct string of characters occurring in the collection of text documents has a dimension. This includes dimensions for common English words and other strings such as email addresses and URLs. For a collection of text documents of reasonable size, the vectors can easily contain hundreds of thousands of elements. For those readers who are familiar with data mining or machine learning, the **vector-space model can be viewed as a traditional feature vector** where **words and strings substitute for more traditional numerical features**. Therefore, it is not surprising that many text mining solutions consist of applying data mining or machine learning algorithms to text stored in a vector-space representation, provided these algorithms can be adapted or extended to deal efficiently with the large dimensional space encountered in text situations.

3.2 Text Representation: Bag of Words and Vector Space Models

The vector-space model makes an implicit assumption (called the **bag-of words assumption**) that the order of the words in the document does not matter. This may seem like a big assumption, since text must be read in a specific order to be understood. For many text mining tasks, such as *document classification* or *clustering*, however, this assumption is usually not a problem. The collection of words appearing in the document (in any order) is usually sufficient to differentiate between semantic concepts. The main strength of text mining algorithms is their ability to use all of the words in the document-primary keywords and the remaining general text. Often, keywords alone do not differentiate a document, but instead the usage patterns of the secondary words provide the differentiating characteristics.

Though the bag-of-words assumption works well for many tasks, it is not a universal solution. For some tasks, such as *information extraction* and *natural language processing*, the **order of words is critical** for solving the task successfully. Prominent features in both entity extraction and natural language processing include both preceding and following words and the decision (e.g., the part of speech) for those words. Specialized algorithms and models for handling sequences such as finite state machines or conditional random fields are used in these cases.

Another challenge for using the vector-space model is the presence of homographs. These are words that are spelled the same but have different meanings.

P.S. Bag of Words model is also known as Vector Space Model.

3.3 Understanding BOW

Bag of words (**BOW**) model represents the text as the bag or multiset of its words, disregarding grammar and word order and just keeping words (**after text preprocessing has been completed**). BOW is often used to generate **features**; after generating BOW, we can derive the **term-frequency** of each word in the document, which can later be fed to a machine learning algorithm.

To vectorize a corpus with a bag-of-words (BOW) approach, we represent every document from the corpus as a vector whose length is equal to the vocabulary of the corpus. We can simplify the computation by sorting token positions of the vector into alphabetical order, as shown in the following Figure. Alternatively, we can keep a dictionary that maps tokens to vector positions. Either way, we arrive at a vector mapping of the corpus that enables us to uniquely represent every document.

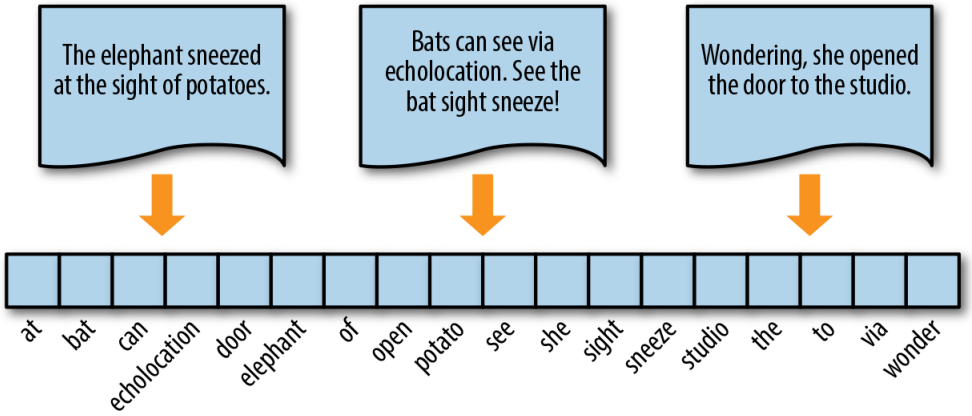

Figure 2. Encoding documents as vectors (Tony Ojeda et al., 2018)

3.3 Understanding BOW

What should **each element** in the document vector be? In the next sections, We will explore three types of **vector encoding** :

- Frequency vector,
- One-hot vector (a binary representation),
- TF–IDF vector (a float-valued weighted vector).

There are many available APIs (such as Scikit-Learn, Gensim, and NLTK) that make the implementations of those vectors encoding easier.

3.4 Frequency vector

- In this representation, each document is represented by one vector where a vector element i represents the number of times (frequency) the i^{th} word appears in the document. This representation can either be a straight count (integer) encoding as shown in the following figure or a normalized encoding where each word is weighted by the total number of words in the document.

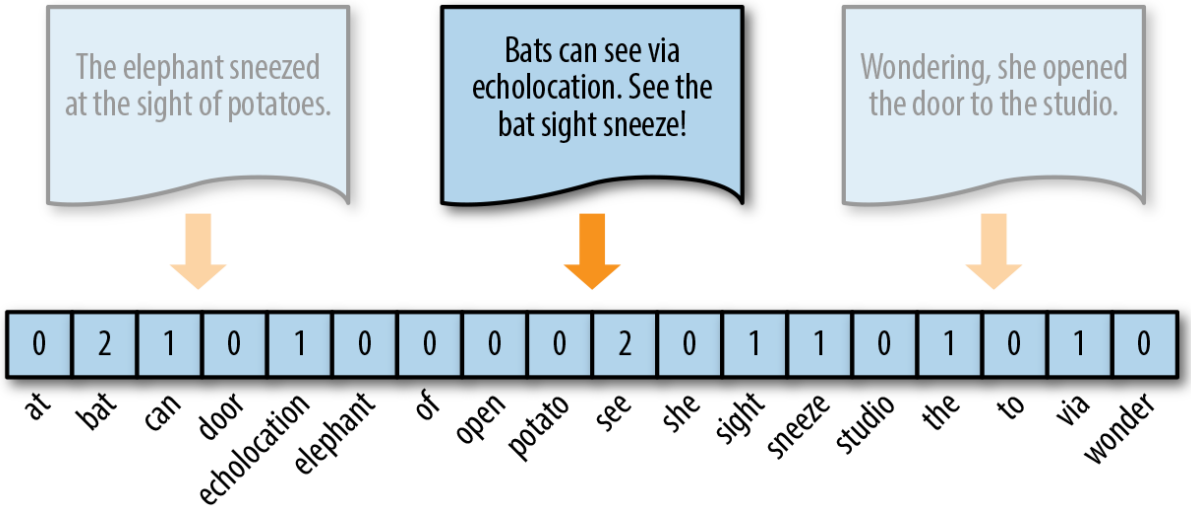

Figure 3. Token frequency as vector encoding (Tony Ojeda et al., 2018)

3.5 One-hot encoding

Because they disregard grammar and the relative position of words in documents, frequency-based encoding methods suffer from the long tail, or Zipfian distribution, that characterizes natural language. As a result, tokens that occur very frequently are orders of magnitude more “significant” than other, less frequent ones. This can have a significant impact on some models (e.g., generalized linear models) that expect normally distributed features.

A solution to this problem is one-hot encoding, a boolean vector encoding method that marks a particular vector index with a value of true (1) if the token exists in the document and false (0) if it does not. In other words, each element of a one-hot encoded vector reflects either the presence or absence of the token in the described text as shown in the following Figure.

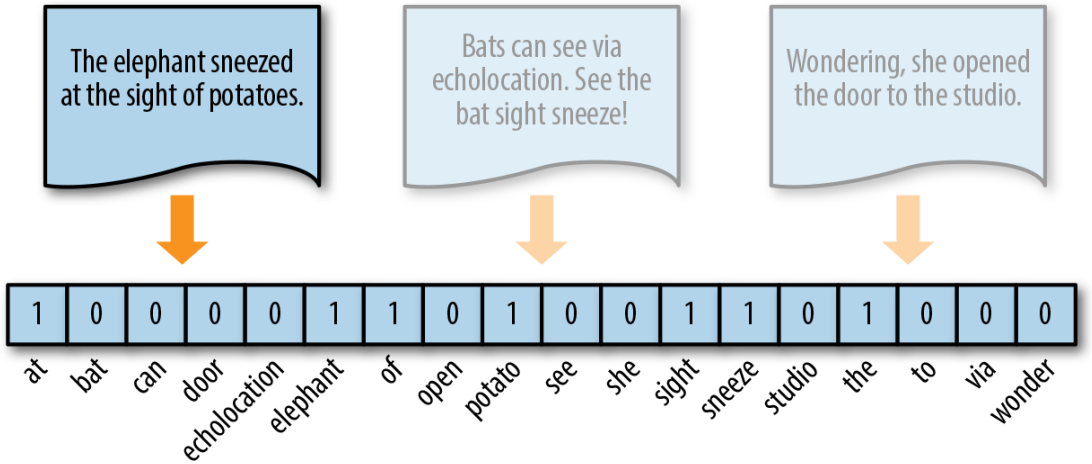

Figure 4. One-hot encoding (Tony Ojeda et al., 2018)

3.5 One-hot encoding

- One-hot encoding is very useful. Some of the **basic applications for one-hot encoding** format are:
  - Many artificial neural networks accept input data in the one-hot encoding format and generate output vectors that carry the sematic representation as well.
  - The word2vec algorithm accepts input data in the form of words and these words are in the form of vectors that are generated by one-hot encoding.
  - …..

3.6 TF-IDF

- Storing text as weighted vectors first requires choosing a weighting scheme. The most popular scheme is the TF-IDF weighting approach.

- The concept TF-IDF stands for **term frequency-inverse document frequency**. This is in the field of numerical statistics. With this concept, we will be able to decide how important a word is to a given document in the present dataset or corpus (collection).

3.6 TF-IDF

- **Term Frequency (TF):**

  The **term frequency** for a term t_i in document $\mathbf{d}_j$ is the number of times that t_i appears in document $\mathbf{d}_j$, denoted by f_{ij}. It can be absolute or relative (normalization may also be applied).

  The **normalized term frequency** (denoted by tf_{ij}) of t_i in $\mathbf{d}_j$ is given by the equation below:

$$tf_{ij} = \frac{f_{ij}}{\max\{f_{1j}, f_{2j}, \ldots, f_{|V|j}\}}$$

where the **maximum** is computed over all terms that appear in **document $\mathbf{d}_j$**. If term t_i does not appear in $\mathbf{d}_j$ then $tf_{ij} = 0$.

|V| *:* is the vocabulary size of the collection (v words).

3.6 TF-IDF

- **Document Frequency (DF)**
  - df_i = **document frequency** of term t_i
    = number of documents containing term t_i
  - The **inverse document frequency** (denoted by idf_i) of term t_i is given by:

$$idf_i = \log \frac{N}{df_i}$$

- Where N: total number of documents
- There is **one IDF value** for **each term t_i** in a collection.
- Log used to dampen the effect relative to tf.

The intuition here is that if a term appears in a large number of documents in the collection, it is probably not important or not discriminative.

3.6 TF-IDF

- The final TF-IDF term **weight** is given by:

$$TF\text{-}IDF\ (t_i, d_j) = w_{ij} = tf_{ij} \times idf_i$$

Postscript:
tf-idf weighting has many variants. Here we only gave the most basic one.

- The assumption behind TF-IDF is that words with high term frequency should receive high weight **unless** they also have high document frequency. The word “the” is one of the most commonly occurring words in the English language. “The” often occurs many times within a single document, but it also occurs in nearly every document. These two competing effects cancel out to give “the” a low weight.

3.6.1 Computing IDF: an example

- Suppose N = 1 million (***Total number of documents)***

- *Use the* log_{10}

term	df_t	idf_t
calpurnia	1	6
animal	100	4
sunday	1,000	3
fly	10,000	2
under	100,000	1
the	1,000,000	0

There is **one idf** value for each term *t* in a collection.

3.6.2 Computing TF-IDF: An Example

Here is a simplified example of the vector space retrieval model. Consider a very small collection C that consists in the following three documents:

d1: “new york times”
d2: “new york post”
d3: “los angeles times”

Some terms appear in two documents, some appear only in one document. The total number of documents is N=3. Therefore, the *idf* values for the terms are:

angles	$log_2(3/1)=1.584$
los	$log_2(3/1)=1.584$
new	$log_2(3/2)=0.584$
post	$log_2(3/1)=1.584$
times	$log_2(3/2)=0.584$
york	$log_2(3/2)=0.584$

3.6.2 Computing TF-IDF: An Example

For all the documents, we calculate the *tf* scores for all the terms in C. We assume the words in the vectors are ordered alphabetically.

	angeles	los	new	post	times	york
d1	0	0	1	0	1	1
d2	0	0	1	1	0	1
d3	1	1	0	0	1	0

Now we multiply the *tf* scores by the *idf* values of each term, obtaining the following matrix of documents-by-terms: (All the terms appeared only once in each document in our small collection, so the maximum value for normalization is 1.)

	angeles	los	new	post	times	york
d1	0	0	0.584	0	0.584	0.584
d2	0	0	0.584	1.584	0	0.584
d3	1.584	1.584	0	0	0.584	0

3.6.3 TF-IDF based applications

Some applications that use TF-IDF:

- In general, text data analysis can be performed by TF-IDF easily. You can get information about the most accurate keywords for your dataset.
- If you are developing a text summarization application where you have a selected statistical approach, then TF-IDF is the most important feature for generating a summary for the document.
- Variations of the TF-IDF weighting scheme are often used by search engines to find out the scoring and ranking of a document's relevance for a given user query.
- Document classification applications.

3.6.4 Implementation: gensim Python Library

models.tfidfmodel – TF-IDF model

https://radimrehurek.com/gensim/models/tfidfmodel.html

class `gensim.models.tfidfmodel.TfidfModel`(*corpus=None*, *id2word=None*, *dictionary=None*, *wlocal=<function identity>*, *wglobal=<function df2idf>*, *normalize=True*, *smartirs=None*)

Bases: **`gensim.interfaces.TransformationABC`**

Objects of this class realize the transformation between word-document co-occurrence matrix (int) into a locally/globally weighted TF_IDF matrix (positive floats).

4. n-grams

- **n-gram** is a very popular and widely used technique in the NLP domain. If you are dealing with text data or speech data, you can use this concept.
- Formal definition of n-grams:

  An n-gram is a continuous sequence of **n items** from the given sequence of text data or speech data. Here, items can be phonemes, syllables, letters, words, or base pairs according to the application that you are trying to solve.

  There are some versions of n-grams that you will find very useful. If we put n=1, then that particular n-gram is referred to as a unigram. If we put n=2, then we get the bigram. If we put n=3, then that particular n-gram is referred to as a trigram, and if you put n=4 or n=5, then these versions of n-grams are referred to as four gram and five gram, respectively.

4.1 Examples

- The following tables shows examples from NLP and computational biology to understand n-gram technique:

  - **Unigram example sequence**

			1-gram
Name of domain	**items**	**Sample sequence of the data**	**unigram**
Computational biology (DNA sequence)	base pair	...AGCTTCGA...	..., A,G,C,T,T,C,G,A ,...
Computational biology (Protine sequence)	Amino acid	...Cys-Gly-Leu-Ser-Trp ...	..., Cys, Gly, Leu, Ser, Trp, ...
NLP	character	...this_is_a_pen...	..., t,h,i,s,_,i,s,_,a,p,e,n ,...
NLP	words	...This is a pen...	..., this,is,a,pen ,...

  - **Bigram example sequence**

With bigrams, we are considering overlapped pairs, as you can see in the following example.

			2-gram
Name of domain	**items**	**Sample sequence of the data**	**bigram**
Computational biology (DNA sequence)	base pair	...AGCTTCGA...	..., AG,GC,CT,TC,CG,GA ,...
Computational biology (Protine sequence)	Amino acid	...Cys-Gly-Leu-Ser-Trp ...	..., Cys-Gly, Gly-Leu, Leu-Ser, Ser-Trp, ...
NLP	character	...this_is_a_pen...	..., th,hi,is,s_,_i,is,s_,_a,a_,_p,pe,en ,...
NLP	words	...This is a pen...	..., this is, is a, a pen ,...

4.1 Examples

- **Trigram example sequence**

The preceding examples are very much self-explanatory. You can figure out how we are taking up the sequencing from the number of n. Here, we are taking the overlapped sequences, which means that if you are taking a trigram and taking the words this, is, and a as a single pair, then next time, you are considering is, a, and pen. Here, the word is overlaps, but these kind of overlapped sequences help store context. If we are using large values for n-five-gram or six-gram, we can store large contexts but we still need more space and more time to process the dataset.

			3-gram
Name of domain	items	Sample sequence of the data	trigram
Computational biology (DNA sequence)	base pair	...AGCTTCGA...	..., AGC,GCT,CTT,TTC,TCG,CGA ,...
Computational biology (Protine sequence)	Amino acid	...Cys-Gly-Leu-Ser-Trp ...	..., Cys-Gly-Leu, Gly-Leu-Ser, Leu-Ser-Trp ,...
NLP	character	...this_is_a_pen...	..., thi,his,is_,s_i,_is,is_,s_a,_a_,a_p,_pe,pen ,...
NLP	words	...This is a pen...	..., this is a, is a pen ,...

4.2 Application

- In this section, we will see what kinds of applications n-gram has been used in:
  - If you are making a plagiarism tool, you can use n-gram to extract the patterns that are copied, because that's what other plagiarism tools do to provide basic features
  - Computational biology has been using n-grams to identify various DNA patterns in order to recognize any unusual DNA pattern; based on this, biologists decide what kind of genetic disease a person may have

4.3 Comparing n-grams and Bag of Words

- We have looked at the concepts of n-grams and Bag of Words (BOW). So, let's now see how n-grams and BOW are different or related to each other.

  - **Difference:**

  The difference is in terms of their usage in NLP applications. In n-grams, word order is important, whereas in BOW it is not important to maintain word order. During the NLP application, n-gram is used to consider words in their real order so we can get an idea about the context of the particular word; BOW is used to build vocabulary for your text dataset.

  - **Relationship**

  If you are considering n-gram as a feature, then BOW is the text representation derived using a unigram. So, in that case, an n-gram is equal to a feature and BOW is equal to a representation of text using a unigram (one-gram) contained within.

5. Deep learning in NLP: an overview

The early era of NLP is based on the rule-based system, and for many applications, an early prototype is based on the rule-based system because we did not have huge amounts of data. Now, we are applying ML techniques to process natural language, using statistical and probability-based approaches where we are representing words in form of one-hot encoded format or co-occurrence matrix.

In this approach, we are getting mostly syntactic representations instead of semantic representations. When we are trying out lexical-based approaches such as bag of words, ngrams, and so on, we cannot differentiate certain context.

Deep Learning (DL) is more useful to solve these issues and other NLP domain-related problems because nowadays, we have huge amounts of data that we can use. We have developed good algorithms such as word2vec, GloVe, and so on in order to capture the semantic aspect of natural language. Apart from this, deep neural networks (DNN) and DL provide some cool capabilities that are listed as follows:

- **Expressibility** : This capability expresses how well the machine can do approximation for a universal function.
- **Trainability** : This capability is very important for NLP applications and indicates how well and fast a DL system can learn about the given problem and start generating significant output.

5. Deep learning in NLP: an overview

- **Generalizability** : This indicates how well the machine can generalize the given task so that it can predict or generate an accurate result for unseen data.

Apart from the preceding three capabilities, there are other capabilities that DL provides us with, such as interpretability, modularity, transferability, latency, adversarial stability, and security.

We know languages are complex things to deal with and sometimes we also don't know how to solve certain NLP problems. The reason behind this is that there are so many languages in the world that have their own syntactic structure and word usages and meanings that you can't express in other languages in the same manner. So we need some techniques that help us **generalize the problem** and give us good results. All these reasons and factors lead us in the direction of the usage of **DNN and DL for NLP applications**.

5.1 Difference between classical NLP and deep learning NLP techniques

In this section, we will compare the classical NLP techniques and DL techniques for NLP.

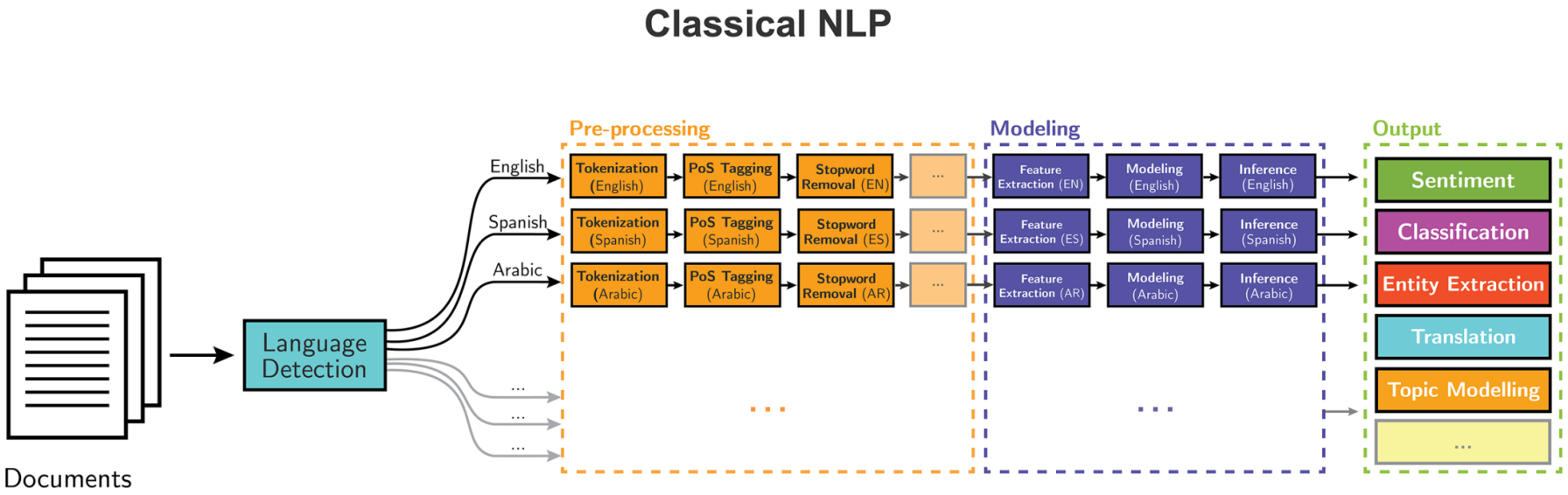

Figure 5: Classical NLP approach (Image credit: https://s3.amazonaws.com/aylien-main/misc/blog/images/nlp-language-dependence-small.png)

5.1 Difference between classical NLP and deep learning NLP techniques

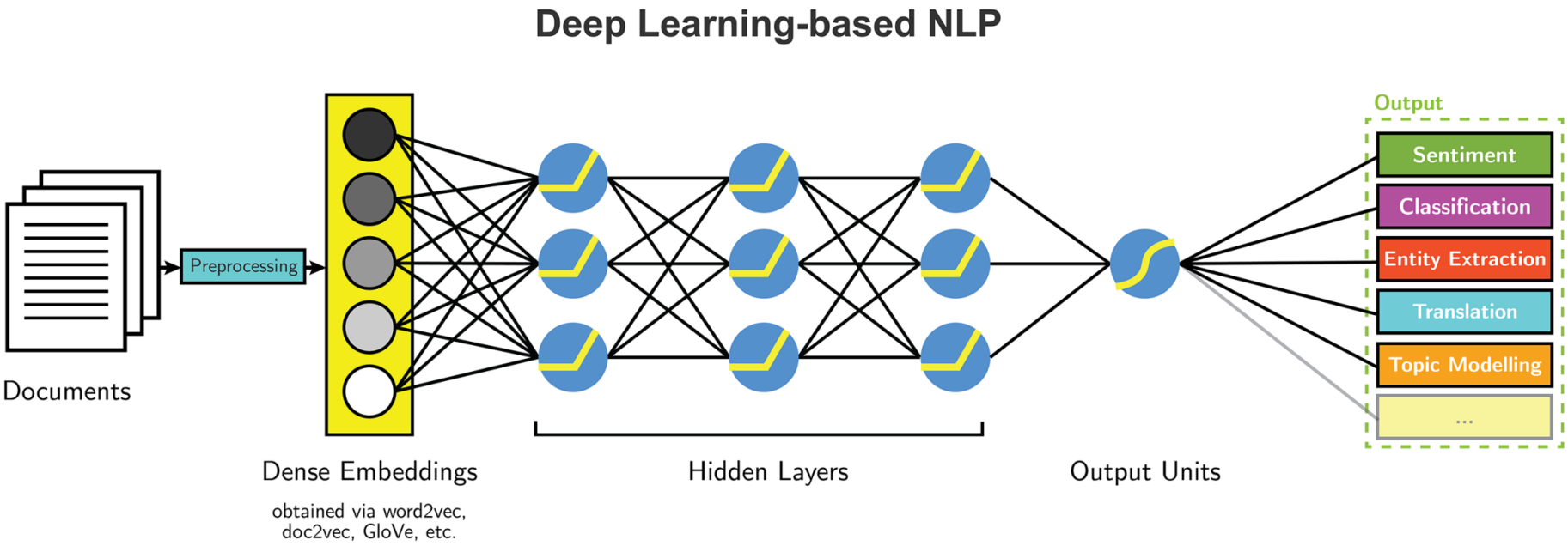

Figure 6: Deep learning approach for NLP (Image credit: https://s3.amazonaws.com/aylien-main/misc/blog/images/nlp-language-dependence-small.png)

5.1 Difference between classical NLP and deep learning NLP techniques

In classical NLP techniques, we preprocessed the data in the early stages before generating features out of the data. In the next phase, we use hand-crafted features that are generated using NER tools, POS taggers, and parsers. We feed these features as input to the machine learning (ML) algorithm and train the model. We will check the accuracy, and if the accuracy is not good, we will optimize some of the parameters of the algorithm and try to generate a more accurate result. Depending on the NLP application, you can include the module that detects the language and then generates features.

In Deep Learning techniques for NLP, we do some basic preprocessing on the data that we have. Then we convert our text input data to a form of dense vectors. To generate the dense vectors, we will use word-embedding techniques such as word2vec, GloVe, doc2vec, and so on, and feed these dense vector embedding to the DNN. Here, we are not using hand-crafted features but different types of DNN as per the NLP application, such as for machine translation, we are using a variant of DNN called **sequence-to-sequence model**. For summarization, we are using another variant, that is, **Long short-term memory units (LSTMs)**. The multiple layers of DNNs generalize the goal and learn the steps to achieve the defined goal. In this process, the machine learns the hierarchical representation and gives us the result that we validate and tune the model as per the necessity.

References

Ashok N. Srivastava and Mehran Sahami. Text Mining Classification, Clustering, and Applications. Pub. Date: 2009, ISBN: 978-1-4200-5940-3. Taylor and Francis Group, LLC.

Diana Inkpen, Information Retrieval and the Internet. Lecture slides. http://www.site.uottawa.ca/~diana/csi4107/

Jalaj Thanaki. Python Natural Language Processing. Pub. Date: 2017, ISBN 978-1-78712-142-3. Packt Publishing Ltd.

John Elder, Dursun Delen, Thomas Hill, Gary Miner and Bob Nisbet. Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications. Pub. Date: 2012, pages: 1093, ISBN: 978-0-12-386979-1. Publisher: Elsevier Science

Tony Ojeda, Rebecca Bilbro, Benjamin Bengfort. Applied Text Analysis with Python. Release Date: June 2018, ISBN: 9781491963036. Publisher: O'Reilly Media, Inc.

Xia Hu and Huan Liu. TEXT ANALYTICS IN SOCIAL MEDIA, Chapter 12. Book title: MINING TEXT DATA. ISBN: 9781461432234 1461432235. Springer US, 2012.