# Practical Work 3

## Session 1: Preprocessing and Feature Extraction using scikit-learn.

Note: This section has two labs. In the second lab we work with word-embedding representation.

*Reynier Ortega Bueno*
*rortega@prhlt.upv.es*

**Goal**

The main objective is to develop a Python functionality to preprocess on the HUrtful Humor dataset and obtain a computational representation of the tweets. For that, the scikit-learn Python library should be used. Also, Natural Language Processing tools such as FreeLing and Spacy can be used .

**Bibliography**

**https://scikit-learn.org/stable/modules/feature_extraction.html#feature-extraction**
https://arxiv.org/abs/1301.3781
https://arxiv.org/abs/1310.4546
https://nlp.stanford.edu/projects/glove/
https://fasttext.cc/

**PREPROCESSING**

The first step in creating a machine learning system is to load the dataset. Then, preprocessing is one of the essential steps in many Natural Language Processing tasks, such as humor recognition. Usually, data needs to be transformed before use.

**Some steps that can be useful in the preprocessing phase:**

- ➢ Conversion of all letters to upper or lower case.
- ➢ Sentence segmentation and tokenization.

- ➢ Noise removal: unwanted characters, such as URLs, punctuation marks, special characters, etc.
- ➢ Removal non-content words (stopwords): The NLTK library can be used to obtain a list of stopwords for each language (https://www.nltk.org/).
- ➢ Stemming: Obtain the root of a word. For that, the PorterStemmer method provided by NLTK is widely used.
- ➢ Lemmatization: Obtain the canonical form of words (lemma).

For text processing in Spanish, both the FreeLing (https://freeling-user-manual.readthedocs.io/en/latest/toc/) and Spacy (https://spacy.io/usage/spacy-101) tools can be used.

## FEATURES EXTRACTION

The input data often are raw texts (sequences of words) that cannot be used directly by the machine learning algorithms because most of them require numerical feature vectors with a fixed size. Raw texts with distinct lengths must be converted into fixed-sized numerical vectors (vectors of features). Some strategies for transforming textual raw data into a numerical representation are:

### Bag-of-Words (BoW)

BoW is an easy and effective way to transform text into a numerical representation. In this representation, each text in the dataset is tokenized. Then, a vocabulary is obtained from all the tokens (lowercase words) in the entire dataset. A vector is obtained for each text. Each axis represents the number of times that a token (in the vocabulary) appears in the text (its simplest version, a binary value, 1 if it occurs at least once, 0 otherwise).

**Please note**:
The dataset is represented by a matrix with one row per text instance and one column per token in the vocabulary. It is a high-dimensional and sparse representation (each token in the vocabulary represents a feature). The order of the words in the text is ignored.

The **scikit-learn** library provides the **CountVectorizer** class. It implements both tokenization and frequency counting in a single class. Some parameters are:

- *max_df*: Ignore terms in the vocabulary with a frequency higher than the given threshold.
- *min_df*: Ignore terms in the vocabulary with a frequency lower than the given threshold.
- *max_features*: Define the number of features to be considered in the representation.
- *vocabulary*: Define a vocabulary to be considered.
- *binary*: All non-zero counts are set to 1.

## EXAMPLE 1

```
from sklearn.feature_extraction.text import
CountVectorizer
# Given the text collection below:
texts=["Amor, este vestido hace que me vea gorda?",
"- Amor, esta corbata hace que me vea calvo?", "-
Pero si estás calvo",
      "Van dos negros, un rumano y un moro en un
coche.", "Quien conduce?", "La policia.", "En que se
parece una sirena y un transexual?", "En que los dos
son mujeres con cola", "Hijo, ¿que quieres de
regalo?...", " — ¡Una Barbie!" ,"— ¡¡Usted es
macho!!", "Pida algo de hierro!!", "— Bueno papi,
una planchita."]
vectorizer = CountVectorizer()
vectorizer.fit(texts)
X_bag_of_words = vectorizer.transform(texts)
X_bag_of_words
#Print the computed vocabulary
print(vectorizer.vocabulary_)
```

### N-grams of words or characters (N-grams)

BoW representation cannot capture multi-word phrases because it does not consider word order. Moreover, BoW does not take into account possible misspellings in the words. A partial solution to these problems is an N-grams representation.

**N-grams**
Instead of building a simple collection of unigrams (*n = 1*), it can build a collection of bigrams (*n = 2*), where the occurrences of consecutive word pairs are counted. Alternatively, one can consider a collection of n-grams of characters which is a robust representation for addressing misspelling.

For example, for the words in: *['words', 'wprds']* the following character bigrams can be obtained:

words → _w, wo, or, rd, ds,s_
wprds → _w, wp, pr, rd, ds, s_

A BoW representation considers both words as different, while a bigrams of characters representation finds 4 out of 8 possible matches.

To obtain character n-grams, the **CountVectorizer** class can be used, but this time the parameters **analyzer** and **ngram_range** need to be specified.

*Parameters:*
- *analyzer*: {'word', 'char', 'char_wb'} or callable, default='word'
- *ngram_range*: tuple (min_n, max_n), default=(1, 1)

## EXAMPLE 2

Bigrams of characters

```
ngram_vectorizer =
CountVectorizer(analyzer='char_wb', ngram_range=(2,
2))
counts = ngram_vectorizer.fit_transform(['words',
'wprds'])
print(ngram_vectorizer.vocabulary_)
for x in counts.toarray().astype(int):
    print(x)
```

```
{' w': 0, 'wo': 6, 'or': 2, 'rd': 4, 'ds': 1, 's ':
5, 'wp': 7, 'pr': 3}
[1 1 1 0 1 1 1 0]
[1 1 0 1 1 1 0 1]
```

## EXAMPLE 3

5-grams of characters

```
ngram_vectorizer = CountVectorizer(analyzer='char',
ngram_range=(5, 5))
counts=ngram_vectorizer.fit_transform(["Este
vestido?", "ver gorda."])
print(ngram_vectorizer.vocabulary_)
for x in counts.toarray().astype(int):
   print(x)
```

```
{'este ': 4, 'ste v': 9, 'te ve': 11, 'e ves': 2, '
vest': 1, 'vesti': 14, 'estid': 5, 'stido': 10,
'tido?': 12, 'ver g': 13, 'er go': 3, 'r gor': 8, '
gord': 0, 'gorda': 6, 'orda.': 7}
```

```
[0 1 1 0 1 1 0 0 0 1 1 1 1 0 1]
[1 0 0 1 0 0 1 1 1 0 0 0 0 1 0]
```

**EXAMPLE 4**
Bigrams of words

```
ngram_vectorizer = CountVectorizer(analyzer='word',
ngram_range=(2,2))
data=ngram_vectorizer.fit_transform(["Amor, este
vestido hace que me vea gorda?", "- Amor, esta
corbata hace que me vea calvo?", "- Pero si estas
calvo", "Van dos negros, un rumano y un moro en un
coche."])

print(ngram_vectorizer.vocabulary_)
for x in data.toarray().astype(int):
   print(x)
```

```
{'amor este': 1, 'este vestido': 7, 'vestido hace':
22, 'hace que': 8, 'que me': 13, 'me vea': 9, 'vea
gorda': 21, 'amor esta': 0, 'esta corbata': 5,
'corbata hace': 2, 'vea calvo': 20, 'pero si': 12,
'si estas': 15, 'estas calvo': 6, 'van dos': 19,
'dos negros': 3, 'negros un': 11, 'un rumano': 18,
'rumano un': 14, 'un moro': 17, 'moro en': 10, 'en
un': 4, 'un coche': 16}
[0 1 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 0 0 0 0 1 1]
[1 0 1 0 0 1 0 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0]
[0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 0 1 1 1 1 0 0 0]
```

## TF-IDF

Term Frequency - Inverse Document Frequency (TF-IDF)

It is a numerical measure that quantifies the importance of the terms in a text given a collection of documents. It has its origins in information retrieval systems. The idea with TF-IDF is to reduce the weight of terms proportionally to the number of documents in which they appear. In this way, a term's value increases proportionally to the number of times it appears in the text and it is compensated by its frequency in the corpus. The Scikit-learn library provides the classes TfidfTransformer and TfidfVectorizer for that purpose.

**TfidfTransformer** transforms a frequency matrix into a TF-IDF representation.

*Some parameters:*
*use_id: Enable IDF weighting.*

### EXAMPLE 5

```
from sklearn.feature_extraction.text import
TfidfTransformer
from sklearn.feature_extraction.text import
CountVectorizer
from sklearn.pipeline import Pipeline
import numpy as np
```

```
texts=["Amor, este vestido hace que me vea gorda?",
"- Amor, esta corbata hace que me vea calvo?", "-
Pero si estas calvo", "Van dos negros, un rumano y
un moro en un coche.", "Quien conduce?", "La
policia.", "En que se parece una sirena y un
transexual?",       "En que los dos son mujeres con
cola", "Hijo, ¿que quieres de regalo?...", " — ¡Una
Barbie!" ,"— ¡¡Usted es macho!!", "Pida algo de
hierro!!", "— Bueno papi, una planchita."]
pipe = Pipeline([('count', CountVectorizer()),
('tfid', TfidfTransformer())])
counts=pipe.fit(texts)
```

**TfidfVectorizer** converts a collection of raw text into a TF-IDF matrix.

```
from sklearn.feature_extraction.text import
TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
tfidf_vectorizer.fit(texts)
```

## Word Embeddings

The idea is to convert each word into an n-dimensional vector. Related words like "women" and "men" are assigned to very close n-dimensional vectors, while different words like "men" and "overweight" have distant vectors. The meaning of a word can be reflected in its vector (embedding), and a model can use this information to learn the relationship between words. Word embedding methods are based on the principle of distributional semantics, which assumes that words occurring in similar contexts have similar meanings. Using a word embedding based representation, a model trained on documents that have the word "women" will be able to match documents that have the word "men" even if this word has never been seen in the training data.

Algorithms for building word embedding:
**Word2Vec**:
https://arxiv.org/abs/1301.3781
https://arxiv.org/abs/1310.4546
**GloVe**:
https://nlp.stanford.edu/projects/glove/
**FastText**:
https://fasttext.cc/

**GEMSIM** is a library for topic modeling, document indexing and retrieval of large data collections.  The code for this is available at https://pypi.org/project/gensim/ and https://github.com/RaRe-Technologies/gensim-data.
This library also provides functionalities to work with word embeddings.

## EXAMPLE 6

```
import gensim.downloader as api
info = api.info()
for item in info["models"].items():
   print (item)
#Downloading the pre-trained word embedding matrix
model = api.load("glove-twitter-25")
rep = model.get_vector("word")
print(rep)
```

```
[ 1.3154    0.01605 -0.25464  0.19656  0.40621
-0.2143    0.88449 -0.40434  -1.2489   0.36182
-0.56618  1.1953  -4.1326  -0.28547  0.15874  1.0733
1.2445   1.1519  -0.28804  0.41741 -0.15655  0.40259
-0.44434  0.45828   0.18119]
```

The following example illustrates how to use the embedding matrix **"glove-twitter-25"**. The semantic relationship between the words "**women**", "**men**", and "**overweight**" can be captured. In this case, the expected result would be that the words "women" and "men" have a higher similarity than the rest of the word pairs that can be formed using these words.

## EXAMPLE 7

```
import gensim.downloader as api
from scipy.spatial import distance

info = api.info()
#Downloading the model
model = api.load("glove-twitter-25")
```

```
w1,w2,w3="women","men","overweight"
rep = model.get_vector(w1)
rep1 = model.get_vector(w2)
rep2 = model.get_vector(w3)

sim1 =1.0-distance.cosine(rep, rep1)
sim2 =1.0-distance.cosine(rep, rep2)
sim3 =1.0-distance.cosine(rep1, rep2)

print(f"similarity({w1},{w2})={sim1}")
print(f"similarity({w1},{w3})={sim2}")
print(f"similarity({w2},{w3})={sim3}")
```

## Pre-trained model for the Spanish language

The models available in the GENSIM API are for English. However, this library provides methods to load a pre-trained embedding matrix from a local file. For that, the **KeyedVector** class can be used. The first step is downloading the embedding matrix in the language we want to use. In the case of the Spanish language, in the following link a pre-trained embedding matrix can be found. It was created using the FastText methods.

http://dcc.uchile.cl/~jperez/word-embeddings/fasttext-sbwc.vec.gz

The example illustrates how to load a Spanish embedding matrix and obtain the word vectors using GENSIM.

**EJEMPLO 8**

```
!wget
http://dcc.uchile.cl/~jperez/word-embeddings/fasttex
t-sbwc.vec.gz
from gensim.models.keyedvectors import KeyedVectors
file_vec = 'fasttext-sbwc.vec.gz'
cantidad = 100000
wordv = KeyedVectors.load_word2vec_format(file_vec,
limit=cantidad)

w1,w2,w3="mujer", hombre, 'sobrepeso'
rep, rep1, rep2 = wordv[w1], wordv[w2], wordv[w3]

sim1 =1.0-distance.cosine(rep, rep1)
sim2 =1.0-distance.cosine(rep, rep2)
sim3 =1.0-distance.cosine(rep1, rep2)
```

```
similarity(mujer,hombre)=0.5953878164291382
similarity(mujer,sobrepeso)=0.27200281620025635
similarity(hombre,sobrepeso)=0.23534655570983887
```

Once you obtain the words in a text and have an embedding matrix, it is possible to represent a sentence using the vectors of the words in it.  A simple way to obtain the representation of the sentence is by averaging the vector of all words in the sentence.

## EXAMPLE 9

```
import numpy as np
def get_sentence_rep(text, keyedvec):
#Obtaining the words or terms in the sentence.
This time we      considered a simple whitespace
splitting.
  words= text.split()
#Notice that, after split the texts, the stopwords
can be removed
#300 is the dimension of the vectors in our
embedding matrix.
  zero_vec=np.zeros(300)
  avg_vec=np.zeros(300)
  total_w=0
  for w in words:
      try:
          avg_vec+=wordv[w]
      except:
          print(f"The word {w} is not found in the
embedding matrix")
          pass
      total_w+=1
  if total_w==0:
      return zero_vec
  return avg_vec/total_w

text= "Van dos negros un rumano y un moro en un
coche"
get_sentence_rep(text, wordv)
```

**ACTIVITY**

Implement the preprocessing and feature extraction for the HUHU training data. For the representation of the texts, two of the studied techniques have to be used, where one of them must be based on word embedding representation. Then, in the second session, the quality of each representation will be evaluated with machine learning models. The proposal's code and report must be delivered before the next session.

**Note:** The training data of the HUHU task (train.csv) for the development of the practical activity is available in the folder PW3 in PoliformaT. It is crucial that once you have the team members and before sending the results, you register as participants in the task and officially request access to the data. https://forms.office.com/e/vyZ5aVuUWC