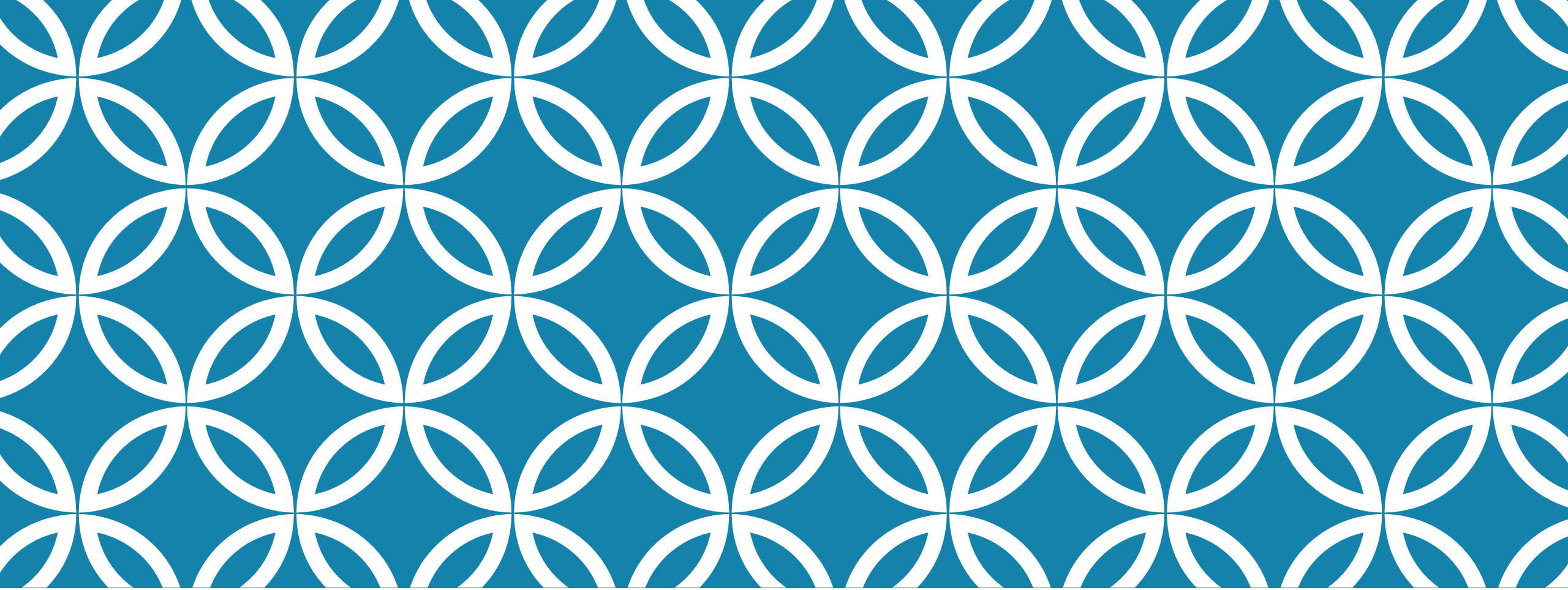


TRABAJANDO CON DATOS DE TEXTO

José Rubén Maldonado Herrera
CCM-UNAM
Curso de Machine Learning

TIPOS DE DATOS REPRESENTADOS COMO *STRINGS*

- Datos Categoricos
  - Listas de selección
- Cadenas de texto libres que pueden ser mapeados a categorías
- Datos estructurados de cadenas
  - Correos
  - Fechas
  - Direcciones
  - Teléfonos
- Datos de texto
  - Párrafos de texto sin “ninguna” restricción.

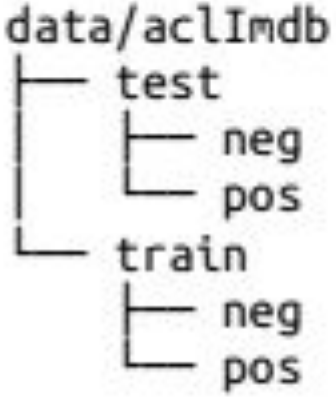
USOS DE DATOS DE TEXTO

- Decidir si un correo es legítimos o spam
- Clasificar textos en temas
- Servicio al cliente, decidir si una reseña es una queja
- Censurar tweets inapropiados

IMBD

IMBd (Internet Movie Database) es un sitio web de reseñas de películas donde se recopilan reseñas junto con una calificación del 1 al 10. Las reseñas con calificación 6 o más son consideraras positivas, mientras que el resto son negativas

Database:



LOAD_FILES()

Es una función de scikit-learn que sirve para cargar archivos que están organizados en subcarpetas, donde cada subcarpeta representa una etiqueta para el archivo.

- Carpeta
  - Subcarpeta 1
  - Subcarpeta 2
  - Subcarpeta 3
  - …
  - Subcarpeta n

Ejemplo utilizando la carpeta de IMBd

In[3]:

```
from sklearn.datasets import load_files

reviews_train = load_files("data/aclImdb/train/")
# load_files returns a bunch, containing training texts and training labels
text_train, y_train = reviews_train.data, reviews_train.target
print("type of text_train: {}".format(type(text_train)))
print("length of text_train: {}".format(len(text_train)))
print("text_train[1]:\n{}".format(text_train[1]))
```

Out[3]:

```
type of text_train:  <class 'list'>
length of text_train:  25000
text_train[1]:
b'Words can\'t describe how bad this movie is. I can\'t explain it by writing
  only. You have too see it for yourself to get at grip of how horrible a movie
  really can be. Not that I recommend you to do that. There are so many
  clich\xc3\xa9s, mistakes (and all other negative things you can imagine) here
  that will just make you cry. To start with the technical first, there are a
  LOT of mistakes regarding the airplane. I won\'t list them here, but just
  mention the coloring of the plane. They didn\'t even manage to show an
  airliner in the colors of a fictional airline, but instead used a 747
  painted in the original Boeing livery. Very bad. The plot is stupid and has
  been done many times before, only much, much better. There are so many
  ridiculous moments here that i lost count of it really early. Also, I was on
  the bad guys\' side all the time in the movie, because the good guys were so
  stupid. "Executive Decision" should without a doubt be you\'re choice over
  this one, even the "Turbulence"-movies are better. In fact, every other
  movie in the world is better than this one.'
```

BAG OF WORDS

Los para poder usar los modelos de machine learning es necesario procesar con anterioridad los datos de texto. Una de las maneras mas simples, efectivas, y comunes de codificar texto es usando *bag-of-words*.

Bag-of-words cuenta cuantas veces aparece una palabra en cada documento del corpus.

TERMINOLOGÍA:

Corpus. Es el dataset que contiene los archivos a analizar

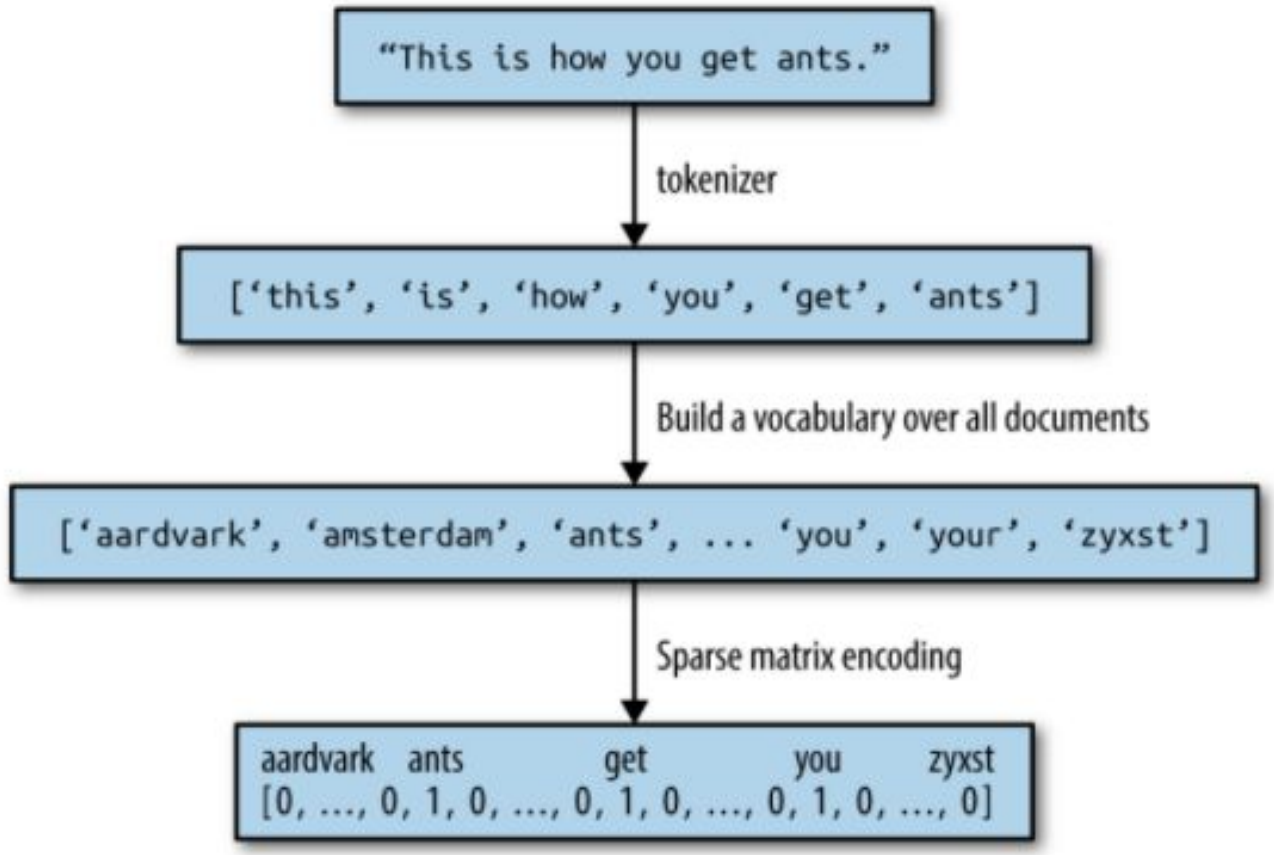
Document. Es cada elemento del corpus.

Consiste de tres pasos

- *Tokenization*. Separa las palabras (tokens) de un documento y las pone en una lista.

- *Vocabulary Building*. Junta todas las palabras del corpus y las ordena en orden alfabético

- *Encoding*. Cuenta cuantas veces aparece cada palabra en cada documento y las codifica en una matriz de enteros no negativos, donde cada renglón representa a cada documento


"This is how you get ants."
tokenizer
['this', 'is', 'how', 'you', 'get', 'ants']
Build a vocabulary over all documents
['aardvark', 'amsterdam', 'ants', ... 'you', 'your', 'zyxst']
Sparse matrix encoding
aardvark ants get you zyxst
[0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0, 1, 0, ..., 0]

COUNTVECTORIZER()

CountVectorizer() es una función de scikit-learn que implementa a bag-of-words.

- Una palabra es definida como una cadena delimitada por espacios en blanco.
  "Hola mundo" son dos palabras
  "wiki.com/blog", "tarea-final", "rub@ccm", "doesn't" forman una palabra cada una

- No considera palabras de longitud uno como "," "." "a" "y"

- Cambia todas las letras a minúsculas

Ejemplo

```
In[7]:

    bards_words =["The fool doth think he is wise,",
                  "but the wise man knows himself to be a fool"]
```

Primero creamos el vocabulario con el método **fit:**

```
In[9]:

    print("Vocabulary size: {}".format(len(vect.vocabulary_)))
    print("Vocabulary content:\n {}".format(vect.vocabulary_))

Out[9]:

    Vocabulary size: 13
    Vocabulary content:
     {'the': 9, 'himself': 5, 'wise': 12, 'he': 4, 'doth': 2, 'to': 11, 'knows': 7,
      'man': 8, 'fool': 3, 'is': 6, 'be': 0, 'think': 10, 'but': 1}
```

Después, creamos la matriz con el método **transform**:

In[10]:

```
bag_of_words = vect.transform(bards_words)
print("bag_of_words: {}".format(repr(bag_of_words)))
```

Out[10]:

```
bag_of_words: <2x13 sparse matrix of type '<class 'numpy.int64'>'
    with 16 stored elements in Compressed Sparse Row format>
```

Imprimimos la matriz, donde cada renglón representa a cada documento:

In[11]:

```
print("Dense representation of bag_of_words:\n{}".format(
    bag_of_words.toarray()))
```

Out[11]:

```
Dense representation of bag_of_words:
[[0 0 1 1 1 0 1 0 0 1 1 0 1]
 [1 1 0 1 0 1 0 1 1 1 0 1 1]]
```

BAG OF WORDS APLICADO A IMBD

In[12]:

```
vect = CountVectorizer().fit(text_train)
X_train = vect.transform(text_train)
print("X_train:\n{}".format(repr(X_train)))
```

Out[12]:

```
X_train:
<25000x74849 sparse matrix of type '<class 'numpy.int64'>'
    with 3431196 stored elements in Compressed Sparse Row format>
```

*

El método get_feature_names nos sirve para obtener una lista de todas la palabras

In[13]:

```
feature_names = vect.get_feature_names()
print("Number of features: {}".format(len(feature_names)))
print("First 20 features:\n{}".format(feature_names[:20]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 2000th feature:\n{}".format(feature_names[::2000]))
```

Out[13]:

```
Number of features: 74849
First 20 features:
['00', '000', '0000000000001', '00001', '00015', '000s', '001', '003830',
 '006', '007', '0079', '0080', '0083', '0093638', '00am', '00pm', '00s',
 '01', '01pm', '02']
Features 20010 to 20030:
['dratted', 'draub', 'draught', 'draughts', 'draughtswoman', 'draw', 'drawback',
 'drawbacks', 'drawer', 'drawers', 'drawing', 'drawings', 'drawl',
 'drawled', 'drawling', 'drawn', 'draws', 'draza', 'dre', 'drea']
Every 2000th feature:
['00', 'aesir', 'aquarian', 'barking', 'blustering', 'bête', 'chicanery',
 'condensing', 'cunning', 'detox', 'draper', 'enshrined', 'favorit', 'freezer',
 'goldman', 'hasan', 'huitieme', 'intelligible', 'kantrowitz', 'lawful',
 'maars', 'megalunged', 'mostey', 'norrland', 'padilla', 'pincher',
 'promisingly', 'receptionist', 'rivals', 'schnaas', 'shunning', 'sparse',
 'subset', 'temptations', 'treatises', 'unproven', 'walkman', 'xylophonist']
```

*

Después de codificar nuestro corpus utilizando bag-of-words podemos aplicar cualquier algoritmo de ML que nos convenga.

En este caso aplicaremos Logistic Regression

In[14]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
scores = cross_val_score(LogisticRegression(), X_train, y_train, cv=5)
print("Mean cross-validation accuracy: {:.2f}".format(np.mean(scores)))
```

Out[14]:

```
Mean cross-validation accuracy: 0.88
```

N-GRAMS

Un problema de considerar solo palabras individuales es que se pierde contexto y el orden en que estás aparecen. Por ejemplo:

“Muy buena, nada mala”

“Muy mala, nada buena”

son dos frases que tienen el mismo vector que las representa pero dicen cosas totalmente opuestas.

Una manera de evitar esto es usando **n-grams,** que son sucesiones de longitud *n* de palabras continuas.

Mediante el parámetro *ngram_range* podemos fijar las longitudes de las sucesiones que queremos usar:

ngram_range = *(i,j)*

Indica que las longitudes de las sucesión serán *i, i+1, …, j+1, j.*

En este caso los tokens son dichas sucesiones

El valor por defautl de ngram_range es (1,1)

Ejemplo, utilizando la lista bards_words

In[28]:

```
cv = CountVectorizer(ngram_range=(1, 1)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[28]:

```
Vocabulary size: 13
Vocabulary:
['be', 'but', 'doth', 'fool', 'he', 'himself', 'is', 'knows', 'man', 'the',
 'think', 'to', 'wise']
```

vocabulary resulta como antes. Ahora cambiemos ngram_range para obtener sucesiones de tamaño dos:

In[29]:

```
cv = CountVectorizer(ngram_range=(2, 2)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[29]:

```
Vocabulary size: 14
Vocabulary:
['be fool', 'but the', 'doth think', 'fool doth', 'he is', 'himself to',
 'is wise', 'knows himself', 'man knows', 'the fool', 'the wise',
 'think he', 'to be', 'wise man']
```

Esta es la matriz resultante:

In[30]:

```
print("Transformed data (dense):\n{}".format(cv.transform(bards_words).toarray()))
```

Out[30]:

```
Transformed data (dense):
[[0 0 1 1 1 0 1 0 0 1 0 1 0 0]
 [1 1 0 0 0 1 0 1 1 0 1 0 1 1]]
```

Finalmente usemos como tokens todas aquellas sucesiones de longitud a lo más tres:

In[31]:

```
cv = CountVectorizer(ngram_range=(1, 3)).fit(bards_words)
print("Vocabulary size: {}".format(len(cv.vocabulary_)))
print("Vocabulary:\n{}".format(cv.get_feature_names()))
```

Out[31]:

```
Vocabulary size: 39
Vocabulary:
['be', 'be fool', 'but', 'but the', 'but the wise', 'doth', 'doth think',
 'doth think he', 'fool', 'fool doth', 'fool doth think', 'he', 'he is',
 'he is wise', 'himself', 'himself to', 'himself to be', 'is', 'is wise',
 'knows', 'knows himself', 'knows himself to', 'man', 'man knows',
 'man knows himself', 'the', 'the fool', 'the fool doth', 'the wise',
 'the wise man', 'think', 'think he', 'think he is', 'to', 'to be',
 'to be fool', 'wise', 'wise man', 'wise man knows']
```

FORMAS DE DISCRIMINAR PALABRAS

- min_df
- Advanced Tokenization
  - Stemming
  - Lemmatization
- Stopwords
- tf-idf

MIN_DF

Otra forma de filtrar palabras es NO considerar palabras con poca frecuencia con respecto a la cantidad de documentos. Esto nos podría ayudar a eliminar errores de dedo o palabras sin significado alguno como “000aw”.

min_df es un parámetro de CountVectorizer() con el cual se indica cual es el mínimo de documentos en los que debe aparecer una palabra para ser considerada.

El valor por default de min_df es 1.

In[17]:

```
vect = CountVectorizer(min_df=5).fit(text_train)
X_train = vect.transform(text_train)
print("X_train with min_df: {}".format(repr(X_train)))
```

Out[17]:

```
X_train with min_df: <25000x27271 sparse matrix of type '<class 'numpy.int64'>'
    with 3354014 stored elements in Compressed Sparse Row format>
```

Como podemos observar, al cambiar min_df se redujo casi en una tercera parte el tamaño del vocabulario. Esto es una ventaja al momento de aplicar ML ya que se tienen menos características que procesar.

In[18]:

```
feature_names = vect.get_feature_names()

print("First 50 features:\n{}".format(feature_names[:50]))
print("Features 20010 to 20030:\n{}".format(feature_names[20010:20030]))
print("Every 700th feature:\n{}".format(feature_names[::700]))
```

Out[18]:

```
First 50 features:
['00', '000', '007', '00s', '01', '02', '03', '04', '05', '06', '07', '08',
 '09', '10', '100', '1000', '100th', '101', '102', '103', '104', '105', '107',
 '108', '10s', '10th', '11', '110', '112', '116', '117', '11th', '12', '120',
 '12th', '13', '135', '13th', '14', '140', '14th', '15', '150', '15th', '16',
 '160', '1600', '16mm', '16s', '16th']
Features 20010 to 20030:
['repentance', 'repercussions', 'repertoire', 'repetition', 'repetitions',
 'repetitious', 'repetitive', 'rephrase', 'replace', 'replaced', 'replacement',
 'replaces', 'replacing', 'replay', 'replayable', 'replayed', 'replaying',
 'replays', 'replete', 'replica']
```

ADVANCED TOKENIZATION

Hay diferentes palabras que básicamente aportan la misma información respecto al documento de texto, por ejemplo:

- Singulares y plurales como “carro” y “carros”

- Palabras con la misma raíz como “remplazar”, “remplazamiento”, “remplazado”, y “remplazando”

Considerar ese tipo de palabras como diferentes puede llevar a un overfitting.

Una solución es identificar palabras con un solo representante.

. Stemming. Identifica palabras según su raíz, esto lo hace mediante reglas básicas de heurística como eliminación de sufijos y prefijos. A la palabra representante se le llama stem

.Lemmatization. Se usa un diccionario previamente establecido. El rol de una palabra en una oración es tomado en cuenta. A la palabra representante se le llama lemma

En ambos casos todas las letras de las palabras son consideradas en minúscula.

.Stemming. Es implementado con el módulo *nltk*

.Lemmatization. Es implementado con el módulo *spacy*

In[36]:

```
import spacy
import nltk

# load spacy's English-language models
en_nlp = spacy.load('en')
# instantiate nltk's Porter stemmer
stemmer = nltk.stem.PorterStemmer()

# define function to compare lemmatization in spacy with stemming in nltk
def compare_normalization(doc):
    # tokenize document in spacy
    doc_spacy = en_nlp(doc)
    # print lemmas found by spacy
    print("Lemmatization:")
    print([token.lemma_ for token in doc_spacy])
    # print tokens found by Porter stemmer
    print("Stemming:")
    print([stemmer.stem(token.norm_.lower()) for token in doc_spacy])
```

```
In[37]:

    compare_normalization(u"Our meeting today was worse than yesterday, "
                          "I'm scared of meeting the clients tomorrow.")

Out[37]:

    Lemmatization:
    ['our', 'meeting', 'today', 'be', 'bad', 'than', 'yesterday', ',', 'i', 'be',
     'scared', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
    Stemming:
    ['our', 'meet', 'today', 'wa', 'wors', 'than', 'yesterday', ',', 'i', "'m",
     'scare', 'of', 'meet', 'the', 'client', 'tomorrow', '.']
```

.Lemmatizatio. Considera el contexto de la palabra “meeting” y las toma como diferentes en cada oración (en una es un sutantivo y en la otra un verbo). Todas las conjugaciones del verbo to be las identifica con “be”. La palabra “worse” la identifica con “bad”.

.Stemming. Identifica la palabra “meeting” con “meet”, la palabra “worse” con “wors”, la palabra “was” con “wa” y separa la palabra “I’m” es dos partes “I” y “ ‘m ”

En CountVectorizer() también podemos implementar nuestra propia forma de discriminar tokes utilizando el parámetro *tokenizer*.

In[38]:

```
# Technicality: we want to use the regexp-based tokenizer
# that is used by CountVectorizer and only use the lemmatization
# from spacy. To this end, we replace en_nlp.tokenizer (the spacy tokenizer)
# with the regexp-based tokenization.
import re
# regexp used in CountVectorizer
regexp = re.compile('(?u)\\b\\w\\w+\\b')

# load spacy language model and save old tokenizer
en_nlp = spacy.load('en')
old_tokenizer = en_nlp.tokenizer
# replace the tokenizer with the preceding regexp
en_nlp.tokenizer = lambda string: old_tokenizer.tokens_from_list(
    regexp.findall(string))

# create a custom tokenizer using the spacy document processing pipeline
# (now using our own tokenizer)
def custom_tokenizer(document):
    doc_spacy = en_nlp(document, entity=False, parse=False)
    return [token.lemma_ for token in doc_spacy]

# define a count vectorizer with the custom tokenizer
lemma_vect = CountVectorizer(tokenizer=custom_tokenizer, min_df=5)
```

STOPWORDS

Otra manera de deshacerse de palabras que no son informativas es descartando palabras que son muy frecuentes, es decir que aparecen en muchos documentos. Estás palabras por lo regular son palabras que se usan en cualquier contexto, como pronombres, artículos, adverbios, conectivos, etc.

Existe una lista predeterminada de dichas palabras es scikit-learn

In[20]:

```
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
print("Number of stop words: {}".format(len(ENGLISH_STOP_WORDS)))
print("Every 10th stopword:\n{}".format(list(ENGLISH_STOP_WORDS)[::10]))
```

Out[20]:

```
Number of stop words: 318
Every 10th stopword:
['above', 'elsewhere', 'into', 'well', 'rather', 'fifteen', 'had', 'enough',
 'herein', 'should', 'third', 'although', 'more', 'this', 'none', 'seemed',
 'nobody', 'seems', 'he', 'also', 'fill', 'anyone', 'anything', 'me', 'the',
 'yet', 'go', 'seeming', 'front', 'beforehand', 'forty', 'i']
```

Podemos descartar estas palabras mediante el parámetro *stop_words* en CountVectorizer():

In[21]:

```
# Specifying stop_words="english" uses the built-in list.
# We could also augment it and pass our own.
vect = CountVectorizer(min_df=5, stop_words="english").fit(text_train)
X_train = vect.transform(text_train)
print("X_train with stop words:\n{}".format(repr(X_train)))
```

Out[21]:

```
X_train with stop words:
<25000x26966 sparse matrix of type '<class 'numpy.int64'>'
    with 2149958 stored elements in Compressed Sparse Row format>
```

Con esto, se reduce un poco el tamaño de la matriz.

TF-IDF

También podemos considerar que tan frecuente ocurre una palabra dentro de un documento pero no en todo el corpus, ya que es muy probable que dicha palabra sea muy descriptiva para el documento en cuestión.

El metodo *term frequency–inverse document frequency* (tf–idf) es una manera de implementar esta idea. Lo que hace es asignar un peso a cada palabra.

En scikit-learn tenemos la función **TfidfTransformer()**, la cual utilizá la matriz producida por CountVectorizer()

La manera en que se asigna el peso a una palabra es:

$$\log\left(\frac{N+1}{N_w+1}\right)+1$$

Después podemos obtener el tf–idf score de una palabra *w* respecto a un documento *d* con la formula:

$$\text{tfidf}(w,d) = \text{tf}\left[\log\left(\frac{N+1}{N_w+1}\right)+1\right]$$

Aquí, *N* significa el número de documentos en el corpus, *N_w* es el número de documentos donde aparece *w,* y *tf* (termino de frecuencia) es la cantidad de veces que aparece *w* en el documento *d*.

Ver ejemplo en

https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/#.YYqcP2DMLIU

TOPIC MODELING AND DOCUMENT CLUSTERING

Una técnica que normalmente se aplica a datos de texto es *topic modeling,* que consiste en clasificar los documentos en temas (clusters). Esta clasificación no es absoluta si no que pondera a que tanto de cada tema tiene un documento. Esto se hace de manera no supervisada.

Esta técnica está implementada en sckit-learn en la función llamada Latent Dirichlet Allocation (LDA).

In[41]:

```
vect = CountVectorizer(max_features=10000, max_df=.15)
X = vect.fit_transform(text_train)
```

In[42]:

```
from sklearn.decomposition import LatentDirichletAllocation
lda = LatentDirichletAllocation(n_topics=10, learning_method="batch",
                                max_iter=25, random_state=0)
# We build the model and transform the data in one step
# Computing transform takes some time,
# and we can save time by doing both at once
document_topics = lda.fit_transform(X)
```

El parámetro *n_topics* indica el número de temas, mientras que *learning_method* indica el método aprendizaje. Para conocer más sobre estos métodos puede visitar la documentación de LatentDirichletAllocation.

In[44]:

```
# For each topic (a row in the components_), sort the features (ascending)
# Invert rows with [:, ::-1] to make sorting descending
sorting = np.argsort(lda.components_, axis=1)[:, ::-1]
# Get the feature names from the vectorizer
feature_names = np.array(vect.get_feature_names())
```

In[45]:

```
# Print out the 10 topics:
mglearn.tools.print_topics(topics=range(10), feature_names=feature_names,
                           sorting=sorting, topics_per_chunk=5, n_words=10)
```

Tabla de temas cuando n_topics = 10

Out[45]:

topic 0	topic 1	topic 2	topic 3	topic 4
between	war	funny	show	didn
young	world	worst	series	saw
family	us	comedy	episode	am
real	our	thing	tv	thought
performance	american	guy	episodes	years
beautiful	documentary	re	shows	book
work	history	stupid	season	watched
each	new	actually	new	now
both	own	nothing	television	dvd
director	point	want	years	got

topic 5	topic 6	topic 7	topic 8	topic 9
horror	kids	cast	performance	house
action	action	role	role	woman
effects	animation	john	john	gets
budget	game	version	actor	killer
nothing	fun	novel	oscar	girl
original	disney	both	cast	wife
director	children	director	plays	horror
minutes	10	played	jack	young
pretty	kid	performance	joe	goes
doesn	old	mr	performances	around

OBSERVACIONES:

- Se necesita de análisis humano para tratar de determinar los nombres de los temas

Por ejemplo, podríamos decir que el topic 1 es historia y películas de guerra, que topic 2 es malas comedias, que topic 6 es acerca de películas para niños, y el topic 8 es películas galardonadas.

- Entre más grande es n_topics los temas se vuelven más específicos

Por ejemplo tomando n_topic = 100

```
In[46]:

    lda100 = LatentDirichletAllocation(n_topics=100, learning_method="batch",
                                       max_iter=25, random_state=0)
    document_topics100 = lda100.fit_transform(X)
```

tenemos los siguientes 5 temas

topic 36	topic 37	topic 41	topic 45	topic 51
performance	excellent	war	music	earth
role	highly	american	song	space
actor	amazing	world	songs	planet
cast	wonderful	soldiers	rock	superman
play	truly	military	band	alien
actors	superb	army	soundtrack	world
performances	actors	tarzan	singing	evil
played	brilliant	soldier	voice	humans
supporting	recommend	america	singer	aliens
director	quite	country	sing	human
oscar	performance	americans	musical	creatures
roles	performances	during	roll	miike
actress	perfect	men	fan	monsters
excellent	drama	us	metal	apes
screen	without	government	concert	clark
plays	beautiful	jungle	playing	burton
award	human	vietnam	hear	tim
work	moving	ii	fans	outer
playing	world	political	prince	men
gives	recommended	against	especially	moon

Podemos decir más claramente que el topic 45 habla sobre música.

Una manera de reforzar nuestra intuición es mirando a las reseñas

In[49]:

```
# sort by weight of "music" topic 45
music = np.argsort(document_topics100[:, 45])[::-1]
# print the five documents where the topic is most important
for i in music[:10]:
    # pshow first two sentences
    print(b".".join(text_train[i].split(b".")[:2]) + b".\n")
```

Out[49]:

```
b'I love this movie and never get tired of watching. The music in it is great.\n'
b"I enjoyed Still Crazy more than any film I have seen in years. A successful
  band from the 70's decide to give it another try.\n"
b'Hollywood Hotel was the last movie musical that Busby Berkeley directed for
  Warner Bros. His directing style had changed or evolved to the point that
  this film does not contain his signature overhead shots or huge production
  numbers with thousands of extras.\n'
b"What happens to washed up rock-n-roll stars in the late 1990's?
  They launch a comeback / reunion tour. At least, that's what the members of
  Strange Fruit, a (fictional) 70's stadium rock group do.\n"
```

CONCLUSIONES

- Antes de poder usar ML para modelar datasets de datos de texto es necesario procesar los datos con la técnica bag-of-words. *

- Hay cuatro maneras de depurar los vocabularios. *

- Tareas comunes que se hacen con datos de texto son de clasificación.

GRACIAS POR SU ATENCIÓN