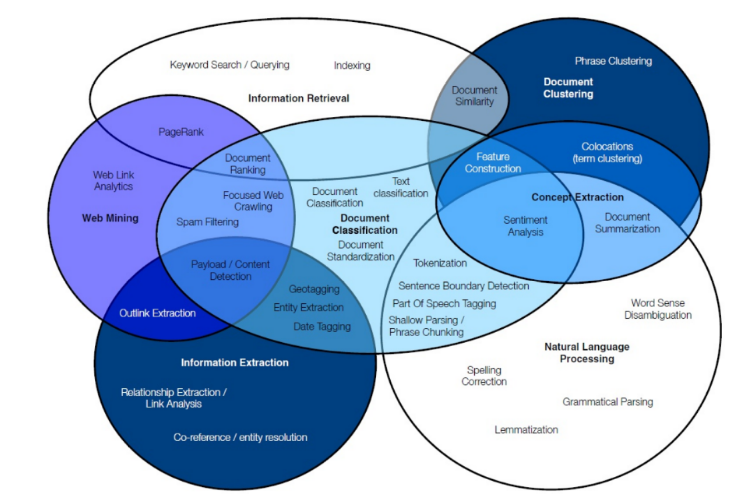
**Text Mining I**

Conjunto de **técnicas** que se utilizan **para explorar grandes cantidades de texto**, en forma **automática o semiautomática**, para **descubrir patrones repetitivos**, **tendencias** o **reglas** que explican el comportamiento del texto. Lo que se busca es **comprender el contenido de un conjunto de textos** a partir de **estadísticas y algoritmos de búsquedas**. Vamos a **analizar** **datos** **de tipo texto**. Usando **algoritmos de Machine Learning** para **hacer clasificación**, **pero con textos**. En definitiva, lo que se busca es **extraer información** sobre el contenido de **textos.**

Algunas aplicaciones iniciales fueron la **clasificación de correos spam** o **no spam**; clasificar **comentarios** de usuarios de un producto como **positivos o negativos**; identificar a qué sección de un diario pertenecen determinadas notas, entre otros.

**Técnicas de Text-Mining y su funcionalidad:**

****

* **Clasificación de documentos:** Clasificación de texto; estandarización de elementos.
* **Recuperación de Información:** Búsqueda de palabras clave; consulta e indexación.
* **Agrupación de documentos:** Agrupación de frases.
* **Procesamiento del lenguaje natural:** Corrección ortográfica; lematización; análisis gramatical; desambiguación del sentido de las palabras.
* **Extracción de información:** Extracción de relaciones; análisis de enlaces.
* **Web mining**

**Para implementar los modelos de Machine Learning**, hay que representar los datos como una **matriz de features**. Si se trata de modelos de aprendizaje supervisado; necesitamos también una etiqueta.

¿Cómo convertir un texto en una matriz de features? Contando los términos que aparecen en los textos es una forma simple de hacerlo. Cada fila va a ser un documento y cada columna, una palabra.

**Problema de la dimensionalidad**: Los textos son secuencias de palabras y signos de puntuación. El sentido de los mismos está contenido en las estructuras semánticas que forman estos elementos combinados. El sentido no lo dan las palabras en sí, sino cómo se articulan las palabras. Queremos buscar una estructura en el espacio de los textos posibles; sin embargo, este espacio **tiene una dimensionalidad enorme**. Para poder trabajar con algoritmos de Machine Learning necesitamos **definir representaciones reducidas de los textos**.

**Bag of Words:** Se descarta la mayor parte de la estructura de los textos, tales como párrafos, capítulos, y quedarnos sólo con el conjunto de palabras y la cantidad de veces que cada una aparece en el texto. Nos olvidamos del orden en el que aparecen. La cantidad de formas en las que podemos ordenar n elementos es n!



Si quisiéramos ver cuántas posibilidades de ordenamiento tenemos con 10 elementos, tenemos:



3628800 combinaciones posibles. Copn “bag of words” como no importa el orden, no importa en qué orden vengan.

**Corpus y documentos**: **Corpus** es un **conjunto de textos**. Un **documento** es cada texto que compone el corpus (libro, twit, comentario); es nuestra unidad de dato (equivalente a una fila de un dataset).

Para poder computar la representación “bag of words” de un corpus de documentos, son necesarios 3 pasos:

1. **Tokenización:** Convertir cada documento a una lista de palabras y signos de puntuación.
2. **Construcción de un vocabulario:** Colectar y ordenar todas las palabras (generalmente en orden alfabético).
3. **Encoding:** Representar los documentos como vectores en el espacio de las palabras del vocabulario.

**Preprocesamiento**: Hay una librería de Python llamada **NLTK** (Natural Language Toolkit) que sirve para el procesamiento de lenguaje natural y cuenta con un libro online gratuito para consultar.

**Tokenización:** Los **tokens** generalmente son las palabras y signos de puntuación. La tokenización es la **transformación de un texto a unidades** constitutivas llamadas **tokens**. Con NLTK podemos hacer esto, incluso interpretando si los signos de puntuación están separando oraciones o haciendo abreviaturas, por ejemplo.

**En Python:**

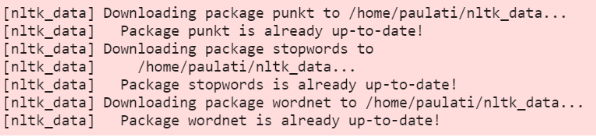
import nltk

nltk.download(‘punkt’) #esta línea debe ejecutarse una sola vez

nltk.download(‘stopword’) #esta línea debe ejecutarse una sola vez

nltk.download(‘wordnet’) #esta línea debe ejecutarse una sola vez

from nltk.tokenize import Word\_tokenize, sent\_tokenize



sentence = ‘Pythoners are very intelligent and work very pythonly and now they are pythoning their way to success.’

print(word\_tokenize(sentence))

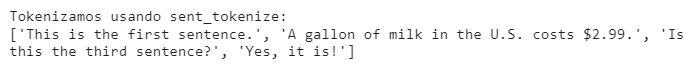


En caso de querer obtener una lista de oraciones.

text = ‘This is the first sentence. A gallon of milk in the U.S. costs $2.99 Is this the third sentence? Yes, it is!’

print(‘Tokenizamos usando sent\_tokenize:’)

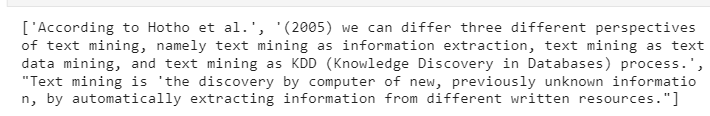
print(sent\_tokenize(text))



Puede fallar. En el siguiente caso, no interpreta que ‘al.’ es una abreviatura:

text2 = ‘According to Hotho et al. (2005) we can differ three different perspectives of text mining, namely text mining as information extraction, text mining as text data mining, and text mining as KDD (Knowledge Discovery in Databases) process. Text mining is ‘the discovery by computer of new, previously unknown information, by automatically extracting information from different written resources.’

print(sent\_tokenize(text2))



Esto se puede resolver incluyendo abreviaturas en el tokenizador. Para hacer esto, podemos instanciar los objetos PunkTrainer y PunktSentenceTokenizer.

from nltk.tokenize.punkt import PunktSentenceTokenizer, PunktTrainer

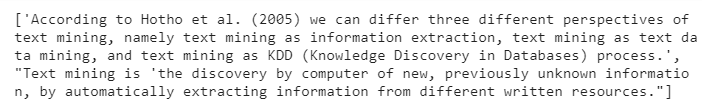
trainer = PunktTrainer()

trainer.INCLUDE\_ALL\_COLOCS = True

tokenizer = PunktSentenceTokenizer(trainer.get\_params())

tokenizer.\_params.abbrev\_types.add(‘al’)

print(tokenizer.tokenize(text2))



**Construcción de un Vocabulario:** Ahora vamos a representar los textos como una bolsa de palabras. Un camino puede ser tokenizar todos los documentos y definir el vocabulario como el set de palabras que aparecieron al menos una vez en todo el corpus. Pero si hacemos esto, el número de **palabras** va a ser muy grande y muchas de ellas **no van a aportar información** sobre el contenido del texto (preposiciones, pronombres, etc.). A estas palabras se las llama **stopwords** y **se las suele excluir** del vocabulario. Otra técnica de reducción de dimensionalidad es **agrupar palabras que comparten la misma raíz etimológica** (ie: correr, corriendo, corredor).

NLTK tiene listas de stopwords en distintos idiomas.

from nltk.corpus import stopwords

stopwords\_sp = stopwords.words(‘spanish’)

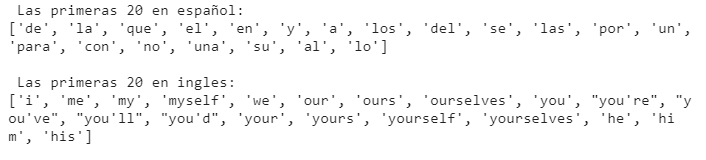
print(‘Las primeras 20 en español:’)

print(stopwords\_sp[:20])

stopwords\_en = stopwords.words(‘english’)

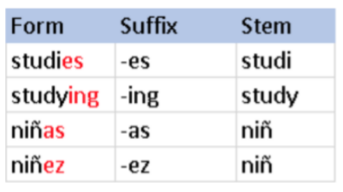
print(‘Las primeras 20 en inglés’)

print(stopwords\_en[:20])

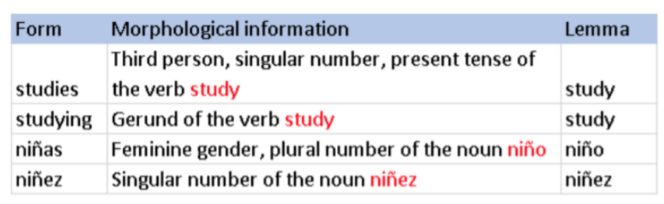


Es posible **reducir las palabras a su raíz etimológica** a través de **Stemming** y **Lemmatization**.

**Stemming** **sustituye** los **sufijos y prefijos** de las palabras. Muchas veces, la raíz que queda no es una palabra en sí. IE: “Corriendo” pasa a ser “corr”



El **Lematizador**, en cambio **devuelve una versión reducida** de la palabra (lema), pero que en sí es una **palabra de la familia**.



**En Python:**

from nltk.stem import PorterStemmer

from nltk.stem import LancasterStemmer

porter = PorterStemmer()

lancaster = LancasterStemmer()

Ahora le vamos a ir pasando palabras a los 2 algoritmos para que hagan stemming:

print(‘Porter Stemmer’)

print(porter.stem(‘cats’))

print(porter.stem(‘trouble’))

print(porter.stem(‘troubling’))

print(porter.stem(‘troubled’))

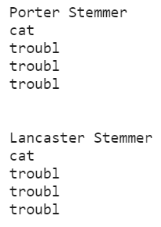
print(‘Lancaster Stemmer’)

print(Lancaster.stem(‘cats’))

print(Lancaster.stem(‘trouble’))

print(Lancaster.stem(‘troubling’))

print(Lancaster.stem(‘troubled’))



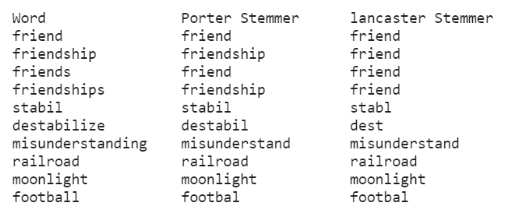
# Creamos una lista de palabras para hacer stemming con ambos algoritmos:

word\_list = [‘friend’, ‘friendship’, ‘friends’, ‘friendships’, ‘stabil’, ‘destabilize’, ‘misunderstanding’, ‘railroad’, ‘moonlight’, ‘football’]

print(‘{0:20}{1:20}{2:20}’.format(‘Word’,’Porter Stemmer’, ‘lancaster stemmer))

for word in word\_list:

print(‘{0:20}{1:20}{2:20}’.format(word,porter.stem(word),lancaster.stem(word)))



Además de PorterStemmer y LancasterStemmer, hay otros stemmers que no están en inglés como **SnowballStemmers**. Este maneja los siguientes idiomas: danés, holandés, inglés, francés, alemán, húngaro, italiano, noruego, portugués, rumano, ruso, español, sueco.

from nltk.stem.snowball import SnowballStemmer

word = ‘having’

englishStemmer=SnowballStemmer(‘english’)

print(‘{0:15}{1:10}’.format(word, englishStemmer.stem(word)))

palabra = ‘corriendo’

spanishStemmer = SnowballStemmer(‘spanish’)

print(‘{0:15}{1:10}’.format(palabra, spanishStemmer.stem(palabra)))



**Lemmatization en Python**:

from nltk.stem import WordNetLemmatizer

wordnet\_lemmatizer = WordNetLemmatizer()

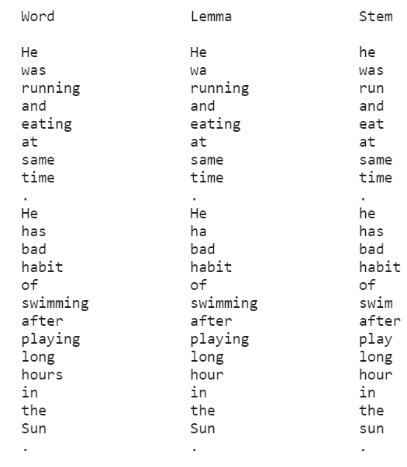
sentence = ‘He was running and eating at the same time. He has bad habit of swimming after playing long hours in the Sun.’

sentence\_words = word\_tokenize(sentence)

print(‘{0:20}{1:20}{2:20}’.format(‘Word’,’Lemma’,’Stem’))

for word in sentence\_words:

print(‘{0:20}{1:20}{2:20}’.format(word, wordnet\_lemmatizer.lemmatize(word), englishStemmer.stem(word)))



**Encoding:** Un **documento** puede representarse como un **vector en el espacio de palabras** que forman el vocabulario. Ya que podemos representar los documentos en función de las palabras que los componen, **independientemente de sus estructuras semánticas**. Hay diferentes maneras de definir estos vectores. La más intuitiva es **contar cuántas veces aparece cada palabra** en un documento y **asignarlo como la coordenada o el peso correspondiente** a dicha palabra en el vector.

Podemos hacer esto con **CountVectorizer**. Nos permite pasarle una **lista de stopwords** para que no sean tenidas en cuenta. Contiene los métodos fit y transform. Con **fit** se **genera el vocabulario** a partir de los documentos. Con **trasnform** se **vectorizan los documentos** al espacio de vocabulario. Como el vocabulario es muy grande, las matrices son muy **esparsas** (con muchos ceros). Entonces conviene almacenar sólo las entradas no nulas de las matrices en un objeto de la clase ‘sparse matrix’; y es posible hacer esto con CountVectorizer.

**En Python:**

t0 = ‘El potro y el ángel llegaron al cine por casualidad’

t1 = ‘El ángel, el tanque del cine nacional, un paso más cerca del oscar’

t2 = ‘final mes del cine nacional: ‘El Potro’, la única cinta ‘millonaria’’

t3 = ‘Juan Martín del potro volvió a Tandil: se dio el último baño de masas con los suyos.’

t4 = ‘Juan Martín del potro fue recibido por una multitud en Tandil’

t5 = ‘Juan Martin del potro fue a ver ‘El Potro’ al cine y le encantó.’

textos = [t0, t1, t2, t3, t4, t5]

from sklearn.feature\_extraction.text import CountVectorizer

from nltk.stem.snowball import SnowballStemmer

stopwords\_sp = stopwords.words(‘spanish’)

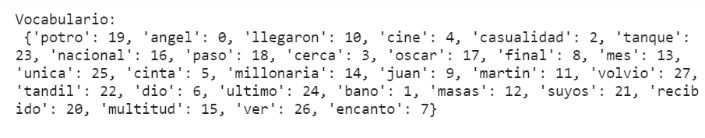
#si no hacemos esto y usamos directamente stopwords\_sp, CountVectorizer devuelve un warning.

stopwords\_sp\_stem = [spanishStemmer.stem(x) for x in stopwords\_sp]

vectorizer = CountVectorizer(stop\_words = stopwords\_sp\_stem, lowercase = True, strip\_accents = ‘unicode’)

vectorizer.fit(textos)

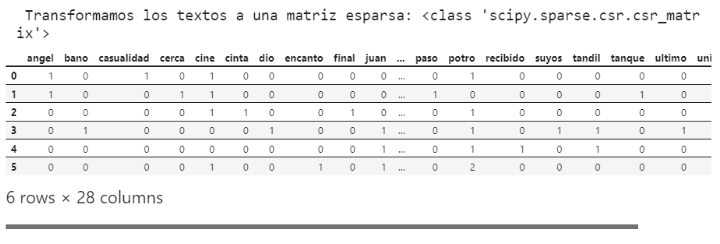
print(‘Vocabulario: ’, vectorizer.vocabulary) # vocabulario del corpus con la frecuencia de cada término.



countvectorizer\_encoding = vectorizer.transform(textos)

print(‘Transformamos los textos a una matriz esparsa’, type(countvectorizer\_encoding))

pd.DataFrame(countvectorizer\_encoding.todense(), columns = vectorizer.get\_feature\_names()) # usamos el método .todense() para ver la matriz completa



**Posibles problemas**:

1. Si un **documento** es **mucho más largo que los demás**, el conteo de palabras de dicho texto puede resultar en números mucho mayores que los demás. Para corregir esto, **habría que normalizar** el conteo de cada palabra por de tamaño de cada documento.
2. Hay palabras que aparecen en muchos documentos, entonces resultan poco informativas para distinguirlos. Si una palabra aparece muchas veces en un documento y pocas en los demás, entonces dicha palabra será importante para representarlo; a la inversa, aquellas palabras que aparezcan pocas veces o que aparezcan en muchos documentos, van a ser menos informativas.

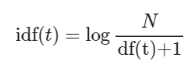
Estas cuestiones se pueden atender con la **transformación TF-IDF:**

El término t en el documento d tiene un coeficiente tf-idf(t,d) que es el producto de dos factores:



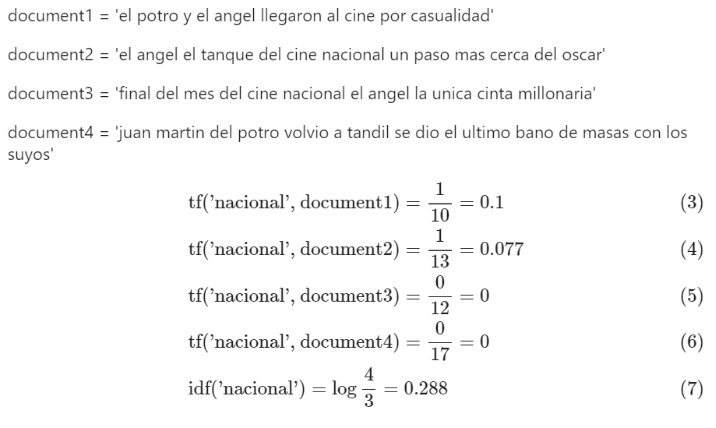
tf(t,d): la frecuencia de aparición de t dentro de d.

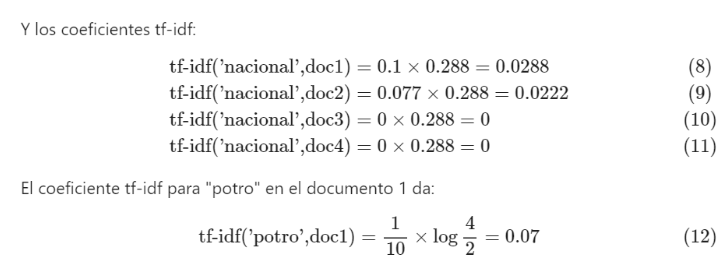
idf(t): *inverse document frequency* del término t:

****

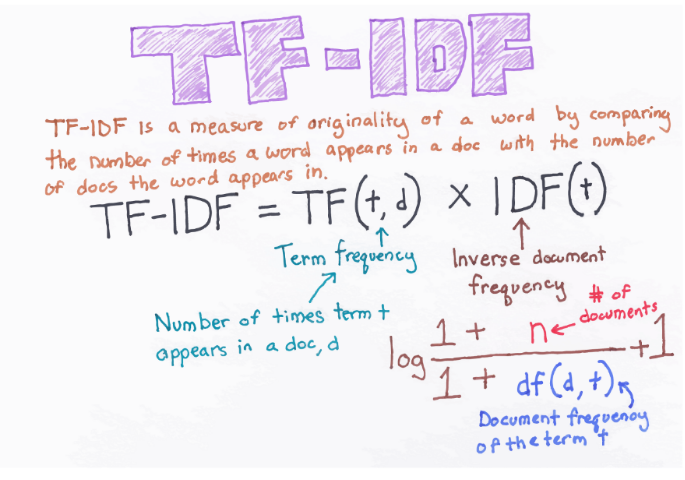
N es el número de documentos; df(t) es el número de documentos donde aparece t. Se suma un 1 en el denominador para evitar dividir por 0 en caso de que no aparezca ningún documento.

Ejemplo:

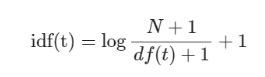




Si bien potro y nacional aparecen el mismo número de veces en el documento 1, potro tiene un coeficiente mayor porque no aparece en ningún otro documento.



Con los objetos de la clase TfidfTransformer de sklearn podemos transformar una matriz generada por CountVectorizer. Incluso podemos saltearnos el paso de usar CountVectorizer y pasar a usar TfidVectorizer directamente. A diferencia de la estimación clásica del factor idf, TfidfVectorizer y TfidfTransformer calculan una estimación suavizada del mismo:



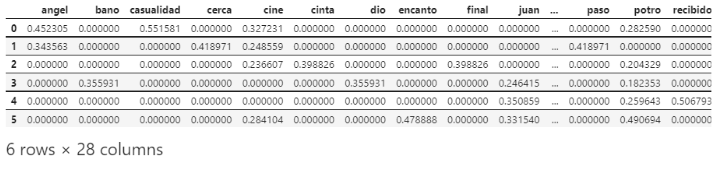
Luego, normalizan los documentos vectorizados por su norma L2.

**En Python:**

from sklearn.feature\_extraction.text import TfidfTransformer

Tfidf\_encoding = TfidfTransformer().fit\_transform(countvectorizer\_encoding)

Pd.DataFrame(Tfidf\_encoding.todense(), columns = vectorizer.get\_feature\_names())

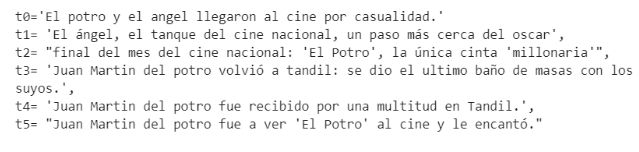


**Singular Value Decomposition (SVD):** Se trata de una técnica general de factorización de matrices. Su matemática es compleja y no es objetivo del curso. Usamos SVD para **reducir la dimensionalidad del corpus del texto**. En vez de remover elementos, lo que vamos a hacer es **encontrar combinaciones de palabras** que resulten **informativas** y **quedarnos con las “mejores”**.

Ejemplo análogo: Tenemos la base y la altura de un rectángulo; queremos reducirlo a una sola medida; tomando alguna de estas 2, damos lugar a infinidad de rectángulos posibles con la misma base o la misma altura, pero si nos quedamos con el área, entonces acotamos bastante la pérdida de información.

SVD es una transformación algebráica similar a PCA, pero aplicable a text mining. Se pueden encontrar combinaciones lineales de los términos que resulten informativos, de forma tal de describir al dataset con un número de combinaciones menores al número de términos que teníamos originalmente. Dichas combinaciones pueden considerarse como **dimensiones con sentido semántico latente** (**latent semantic dimensions**); se trata de dimensiones en las que tiene sentido proyectar el dataset por su contenido semántico. Muchas veces existe **redundancia en el conjunto de documentos**. Se dice más o menos lo mismo en muchos documentos.

Vamos a volver al ejemplo de los 6 textos con los que veníamos trabajando:



Hay 45 palabras distintas, pero los textos hablan esencialmente de 3 temas: dos películas nuevas en el cine (1); Del Potro visitó Tandil (2) y Del Potro fue al cine (3).

Con esta reducción de la dimensionalidad podría mejorarse la performance de un clasificador o de un modelo de clustering. Si reducimos a 2 o 3 dimensiones incluso podemos visualizar los datos. El problema es que en general vamos a necesitar más dimensiones para poder describir correctamente el corpus. Puede que la representación en 2 dimensiones no nos termine diciendo mucho sobre la estructura del dataset.

**En Python:**

import matplotlib.pyplot as plt

import matplotlib.patches as mpatches

from sklearn.decomposition import TruncatedSVD

svd = TruncatedSVD(n\_components = 2)

P = svd.fit\_transform(Tfidf\_encoding)

#grafico

color = [‘m’, ‘g’, ‘r’, ‘c’, ‘b’, ‘k’]

plt.figure()

patches = []

for i, texto in enumerate(textos):

plt.plot(P[i,0], P[i,1], color[i]+’o’)

patches.append(mpatches.Patch(color = color[i], label = ‘t’ + str(i)))

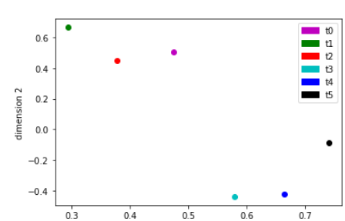
plt.legend(handles=patches)

plt.xlabel(‘dimension 1’)

plt.ylabel(‘dimension 2’)

#plt.axis([-4, 4, -4, 4])

plt.show()



Las dimensiones generadas son combinaciones lineales de los términos. En la definición de estas dimensiones podemos ver cuánto pesa cada término. Vamos a ordenar los términos en función de cuanto pesan en cada dimensión:

# coeficientes de los términos en cada una de las 2 dimensiones.

comp1, comp2 = svd.components\_

# los ordenamos de menor a mayor y tomamos los índices de sus posiciones en el array.

indices = np.argsort(comp1)

# invertimos para que queden ordenados de mayor a menor.

indices = índices[::-1]

# evaluamos los términos en estas posiciones.

print(‘Dimension 1:’)

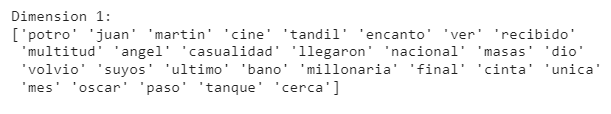
print(np.array(vectorizer.get\_feature\_names())[indices])

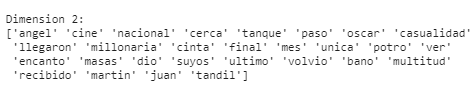
indices = np.argsort(comp2)

indices = índices[::-1]

print(‘Dimension 2:’)

print(np.array(vectorizer.get\_feature\_names())[indices])





Entonces vemos que hay 2 grupos separados: uno habla de cine y el otro de tenis. Trabajando con un algoritmo de aprendizaje no supervisado, estos 2 grupos podrían encontrarse con un algoritmo de clustering.

**Conclusiones:**

* Vimos conceptos y técnicas útiles de preprocesamiento de texto para construir features que sirvan para modelos de aprendizaje supervisado o no supervisado.
* Conocimos NTLK, librería de Python para procesamiento de lenguaje natural.
* Vimos como reducir la dimensionalidad usando Singular Value Decomposition (SVD)