U3 :: Procesamiento del Lenguaje Natural

3. Uso de modelos lingüísticos

Objetivos de este apartado

El objetivo de este apartado es utilizar varios métodos, incluidos modelos de lenguaje e *inserciones de palabras*, para representar un texto numéricamente.

En particular:

- Identificar aplicaciones comunes del mundo real para el modelo de lenguaje de "saco de palabras" (BoW).
- Convertir texto en una representación de BoW utilizando un diccionario de Python.
- Comprender cómo se pueden usar los "diccionarios de características" y los "vectores de características" para construir un modelo de BoW.
- Reconocer la diferencia entre datos de entrenamiento y de prueba.
- Utilizar *scikit-learn* para convertir texto en un vector BoW.
- Identificar las ventajas y desventajas de BoW en relación con otros modelos de lenguaje comunes.
- Comprender cómo *tf-idf* ayuda a determinar la importancia de las palabras en los textos.
- Implementar *tf-idf* usando *scikit-learn*.
- Replantear el significado de las palabras según el contexto utilizando "inserciones de palabras".
- Comprender cómo los vectores multidimensionales y las métricas de distancia son útiles para las representaciones numéricas del lenguaje.
- Implementar incrustaciones de palabras, incluida word2vec, con spaCy y gensim.

3.1. Saco de palabras (*Bag-of-words*)

3.1.1. Introducción

El modelo de lenguaje de "saco de palabras" (*Bag-of-words*) es una herramienta simple pero poderosa relacionada con el procesamiento del lenguaje natural.

El modelo tiene muchos casos de uso, por ejemplo:

- determinar temas en una canción,
- filtrar el correo no deseado de su bandeja de entrada,
- averiguar si un *tweet* tiene un sentimiento positivo o negativo,
- crear nubes de palabras

3.1.2. ¿Saco de qué?

Comparativamente hablando, los lenguajes son más difíciles de interpretar y analizar para los algoritmos que los datos numéricos. Esto es así por algunas razones:

1. Las oraciones no tienen una longitud fija, pero la mayoría de los algoritmos requieren un tamaño de vector de entrada estándar. Por lo tanto, se requiere relleno, correspondiente a la oración más grande en los corpus.
2. Los algoritmos ML no pueden entender las palabras como un dato de entrada válido: por lo tanto, cada palabra debe estar representada por algún valor numérico.

El modelo de *saco de palabras* permite extraer características de datos textuales. Como sabemos, un algoritmo no comprende el lenguaje. Por lo tanto, necesitamos usar una representación numérica para las palabras en el corpus. Esta representación numérica puede servir como entrada posteriormente a cualquier algoritmo para el análisis de datos.

Se llama **"saco de palabras"** porque el orden de las palabras o la estructura de la oración no importa en este modelo. Solo importa la aparición o presencia de una palabra. Podemos pensar en el modelo de esta manera: tenemos una bolsa grande, vacía al principio, y un vocabulario o un corpus. Recogemos las palabras una por una y las colocamos en la bolsa, agregando la frecuencia de su aparición, y luego seleccionamos las palabras más comunes como características para pasar a través de nuestro algoritmo de elección.

Por ejemplo, digamos que tenemos el texto:

"Five fantastic fish flew off to find faraway functions. Maybe find another five fantastic fish?"

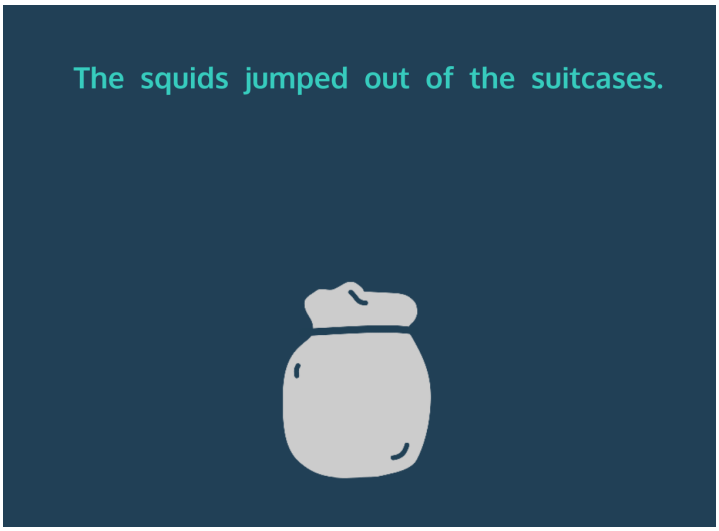
Un modelo de lenguaje probabilístico basado en la letra inicial de las palabras podría tomar este texto y predecir que es más probable que las palabras comiencen con la letra "f" porque 11 de las 15 palabras comienzan así. Un modelo estadístico diferente que preste atención al orden de las palabras podría

decirnos que la palabra "fish" tiende a seguir a la palabra "fantastic". Sin embargo, el *saco de palabras* no relaciona a *"un pez volador"* con su lugar en la oración o con el orden de las palabras; su única preocupación es el ***recuento de palabras***, es decir, con cuántas veces aparece cada palabra en un documento.

3.1.3. Diccionarios BoW

Una de las formas más comunes de implementar el modelo BoW en Python es como un diccionario donde cada clave remite a una palabra y cada valor al número de veces que aparece esa palabra.

Por ejemplo:



Las palabras de la oración van *al saco de palabras* y se formalizan como un diccionario de palabras con sus correspondientes recuentos. En los modelos estadísticos, llamamos ***datos de entrenamiento*** al texto que usamos para construir el modelo. Por lo general, necesitamos preparar nuestros datos de texto dividiéndolos en *documentos* (es decir, cadenas de texto más cortas, generalmente oraciones).

3.1.4. Introducción a los vectores BoW

A veces, los diccionarios no se ajustan a lo que necesitamos. Las aplicaciones de *identificación de temas* (en inglés, *topic modelling*), por ejemplo, requieren de una implementación de *saco de palabras* con más matemáticas "por debajo". En estos casos, se necesita manejar el lenguaje como ***vectores de características***.

Un *vector de características* es una representación numérica de las características importantes de un elemento. Cada función tiene su propia columna. Si la característica existe para el elemento, se puede representar con un 1. Si la característica no existe para ese elemento, se hace con un 0. Algunos personajes de las novelas fantásticas podrían representarse como vectores así:

	tiene_colmillos	se_derrite_en_agua	odia_luz_solar	tiene_pelo
vampiro	1	0	1	0
hombre lobo	1	0	0	1
bruja	0	1	0	0

En los *sacos de palabras*, en lugar de monstruos, tendrías documentos y las características serían palabras diferentes. En este caso, no solo nos importa si una palabra está presente en un documento; ¡queremos saber cuántas veces aparece! Convertir texto en un vector BoW se conoce como ***extracción de características*** o ***vectorización***.

Pero ..., ¿cómo sabemos qué índice vectorial corresponde a qué palabra? Al construir vectores BoW, generalmente creamos un ***diccionario de características*** con todo el vocabulario en nuestros datos de entrenamiento (generalmente varios documentos) mapeado a índices.

Por ejemplo, con *"Five fantastic fish flew off to find faraway functions. Maybe find another five fantastic fish?"* nuestro diccionario podría ser:

{'five': 0, 'fantastic': 1, 'fish': 2, 'fly': 3, 'off': 4, 'to': 5, 'find': 6, 'faraway': 7, 'function': 8, 'maybe': 9, 'another': 10}

Usando este diccionario, podemos convertir nuevos documentos en vectores usando una función de vectorización. Por ejemplo, si tomamos una oración nueva ""Another five fish find another faraway fish." - ***datos de prueba ("data test")*** - y lo convertimos a un vector, podría quedar de la siguiente forma:

[1, 0, 2, 0, 0, 0, 1, 1, 0, 0, 2]

La palabra *"another"* apareció dos veces en los datos de prueba. Si miramos el diccionario de características para *"another"*, encontramos que su índice es 10. Entonces, si miramos nuestro vector, tendremos que el número en el índice 10 es 2.

3.1.5. Creación de un diccionario de características

Ahora que sabemos cómo es un vector de *saco de palabras*, podemos crear una función que lo construya.

Primero, necesitamos una forma de generar un diccionario de características a partir de una lista de documentos de entrenamiento ("training data"). Podemos construir una función de Python que lo haga por nosotros ...

```
import nltk, re
#nltk.download('punkt')
#nltk.download('stopwords')
#nltk.download('wordnet')
from nltk.corpus import wordnet
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from collections import Counter
stop_words = stopwords.words('english')
normalizer = WordNetLemmatizer()

# Función para etiquetado morfológico
def get_part_of_speech(word):
probable_part_of_speech = wordnet.synsets(word)
pos_counts = Counter()
pos_counts["n"] = len( [ item for item in probable_part_of_speech if
item.pos()=="n"]  )
pos_counts["v"] = len( [ item for item in probable_part_of_speech if
item.pos()=="v"]  )
pos_counts["a"] = len( [ item for item in probable_part_of_speech if
item.pos()=="a"]  )
pos_counts["r"] = len( [ item for item in probable_part_of_speech if
item.pos()=="r"]  )
most_likely_part_of_speech = pos_counts.most_common(1)[0][0]
return most_likely_part_of_speech
# Función para preprocesado de texto
def preprocess_text(text):
cleaned = re.sub(r'\W+', ' ', text).lower()
tokenized = word_tokenize(cleaned)
normalized = [normalizer.lemmatize(token, get_part_of_speech(token)) for token
in tokenized]
return normalized
# Función para creación del diccionario de características
def create_features_dictionary(documents):
features_dictionary = dict()
merged = " ".join(documents)
tokens = preprocess_text(merged)
index = 0
for token in tokens:
```

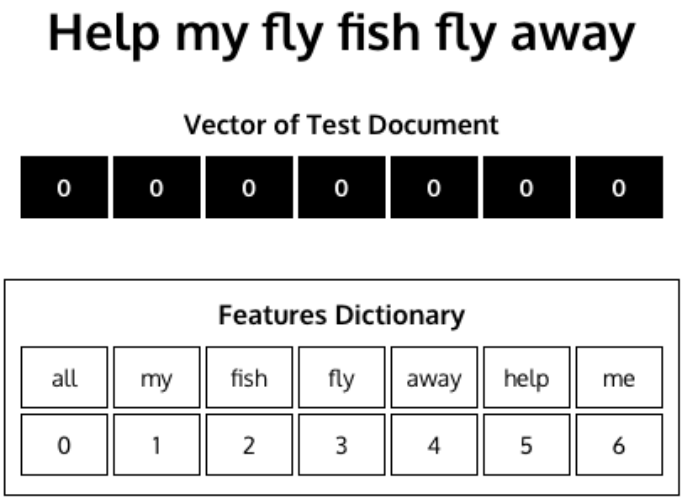
```
  if token not in features_dictionary:
    features_dictionary[token] = index
    index += 1
return features_dictionary

# Datos de entrenamiento
training_documents = ["Five fantastic fish flew off to find faraway functions.",
"Maybe find another five fantastic fish?", "Find my fish with a function
please!"]
features_dictionary = create_features_dictionary(training_documents)
print("DICTIONARY BoW:\n", features_dictionary)


---
DICTIONARY BoW:
 {'five': 0, 'fantastic': 1, 'fish': 2, 'fly': 3, 'off': 4, 'to': 5, 'find': 6, 'faraway': 7, 'function': 8, 'maybe': 9, 'another':
10, 'my': 11, 'with': 12, 'a': 13, 'please': 14}
```

3.1.6. Construyendo un vector BoW

A continuación, usaremos el diccionario de vocabulario para crear un vector de *saco de palabras* a partir de un nuevo documento. En Python, podemos usar una lista para representar un vector. Cada índice de la lista corresponderá a una palabra y se establecerá en su recuento.



```
# Creación de un vector BoW
def text_to_bow_vector(some_text, features_dictionary):
bow_vector = [ 0 for i in range(len(features_dictionary))]
tokens = preprocess_text(some_text)
for token in tokens:
  feature_index = features_dictionary[token]
  bow_vector[feature_index] += 1
return bow_vector, tokens
# La variable 'text' contiene los datos de prueba
text = "Another five fish find another faraway fish."
vector_bow, tokens = text_to_bow_vector(text, features_dictionary)

# Mostramos los resultados
print("TESTING TEXT: ", text)
print("DICTIONARY BoW: ", features_dictionary)
print("VECTOR BoW: ", vector_bow)
```

TESTING TEXT: Another five fish find another faraway fish.

DICTIONARY BoW: {'five': 0, 'fantastic': 1, 'fish': 2, 'fly': 3, 'off': 4, 'to': 5, 'find': 6, 'faraway': 7, 'function': 8, 'maybe': 9, 'another': 10, 'my': 11, 'with': 12, 'a': 13, 'please': 14}

VECTOR BoW: [1, 0, 2, 0, 0, 0, 1, 1, 0, 0, 2, 0, 0, 0, 0]

3.1.7. Usando funciones de librería

Para todo lo ilustrado anteriormente, ya existe una librería en Python que nos puede ahorrar tiempo y esfuerzo: **scikit-learn.**

La función *text_to_bow()* anterior se aproxima a la funcionalidad de la función *Counter()* del módulo *collections*:

```
from collections import Counter

tokens = ['another', 'five', 'fish', 'find', 'another', 'faraway', 'fish']

print(Counter(tokens))

# Counter({'fish': 2, 'another': 2, 'find': 1, 'five': 1, 'faraway': 1})
```

Para la parte de vectorización .. .

1. podemos utilizar *CountVectorizer* presente en la librería *scikit-learn* ...
2. ... junto al método ***fit()*** para generar y rellenar el diccionario de características y ***transform()*** para transformarlo en un vector:

Esto sería un ejemplo de cómo aplicarlo:

```
from sklearn.feature_extraction.text import CountVectorizer

training_documents = ["Five fantastic fish flew off to find faraway functions.", "Maybe find another five fantastic fish?", "Find my fish with a function please!"]

test_text = ["Another five fish find another faraway fish."]

bow_vectorizer = CountVectorizer()

# Se crea el diccionario de características

bow_vectorizer.fit(training_documents)

# Imprimimos el  conjunto de palabras contenidas en los documentos iniciales

print("Vocabulary: ")

print(bow_vectorizer.get_feature_names_out())

# Imprimimos el  diccionario de características

print("Dictionary (words + indexes): ")

print(bow_vectorizer.vocabulary_)

# Se vectoriza el diccionario de características, mostrando el resultado

bow_vector = bow_vectorizer.transform(test_text)

print("Full vector: ")

print(bow_vector.toarray())

# Imprimimos el vector para la palabra "fish"
```

```
print("Vector for 'fish': ")

print(bow_vectorizer.transform(['fish']).toarray())
```

Output:

Vocabulary:

['another', 'fantastic', 'faraway', 'find', 'fish', 'five', 'flew', 'function', 'functions', 'maybe', 'my', 'off', 'please', 'to', 'with']

Dictionary (words + indexes):

{'five': 5, 'fantastic': 1, 'fish': 4, 'flew': 6, 'off': 11, 'to': 13, 'find': 3, 'faraway': 2, 'functions': 8, 'maybe': 9, 'another': 0, 'my': 10, 'with': 14, 'function': 7, 'please': 12}

Full vector:

[[2 0 1 1 2 1 0 0 0 0 0 0 0 0 0]]

Vector for 'fish':

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

3.1.8. Ventajas de BoW

BoW también tiene varias ventajas sobre otros modelos de lenguaje. Por un lado, es un modelo más fácil de aplicar y algunas librerías de Python ya tienen soporte integrado para él. Debido a que el *saco de palabras* se basa en palabras individuales, en lugar de en secuencias de palabras, hay más ocurrencias de cada unidad de lenguaje en el corpus de entrenamiento. Más "ocurrencias" significa que el modelo tiene menos **escasez de datos** (es decir, tiene más "contenido" para aprovechar como datos de entrenamiento) que otros modelos estadísticos.

Veámoslo con un ejemplo. Imagina que quieres diseñar una camisa para venderla a la gente. Si la camisa se ajusta exactamente a la medida del cuerpo de alguien, probablemente no le quedaría bien a mucha gente. Sin embargo, si diseñas una camisa con el tamaño de un gigante, sabes que nadie la comprará. ¿qué sería mejor hacer? Seguramente tomar como referencia una camisa para un cuerpo "estándar", que tenga cierta holgura adicional para que le sirva a diferentes personas. Este fenómeno se conoce como "sobreajuste" (en inglés, "overfitting").

El sobreajuste (es decir, "forzar" un modelo a los datos de entrenamiento) es un problema común para los modelos de lenguaje estadístico. Si bien BoW todavía sufre de sobreajuste en términos de vocabulario, se

sobreajusta menos que otros modelos estadísticos, lo que permite una mayor flexibilidad gramatical y de elección de palabras.

La combinación de menor escasez de datos y menor sobreajuste hace que el modelo de bolsa de palabras sea más confiable con conjuntos de datos de entrenamiento más pequeños que otros modelos estadísticos.

Además, cabe añadir que ...

1. El **modelo de bigramas** es otro modelo estadístico que es útil para tareas como la predicción de texto. Sin embargo, no siempre es ideal cuando desea determinar el tema de un texto determinado.

Revisa el siguiente texto breve en el código y luego ejecútalo tal cual para ver los bigramas más comunes.

```
from nltk.util import ngrams
from collections import Counter

text = "It's exciting to watch flying fish after a hard day's work. I don't know
why some fish prefer flying and other fish would rather swim. It seems like the
fish just woke up one day and decided, 'hey, today is the day to fly away.'"

tokens = preprocess_text(text)

# Recuento basado en modelo Bi-grama:
bigrams_prepped = ngrams(tokens, 2)
bigrams = Counter(bigrams_prepped)
print("Three most frequent word sequences and the number of occurrences
according to Bigrams:")
print(bigrams.most_common(3))

# Recuento basado en modelo "saco de palabras":
print("\nThree most frequent words and number of occurrences according to
Bag-of-Words:")
bag_of_words = Counter(tokens)
most_common_three = bag_of_words.most_common(3)
print(most_common_three)
```

2. Debido a la escasez de datos, cada bigrama tiene una sola ocurrencia. Como resultado, el modelo de bigrama por sí solo no es bueno para hacer predicciones sobre temas o sentimientos.

Veamos ahora cómo funcionaría con el modelo de "saco de palabras" ...

3. Al final del siguiente script, imprimimos las tres palabras que aparecen con más frecuencia en el texto. Podemos encontrar las palabras más comunes llamando al método ***.most_common()*** de *Counter* y pasando un número, en este caso 3, como argumento.

3.1.9. Inconvenientes de BoW

Pero no todo en BoW son ventajas ...

BoW NO es un modelo óptimo para la predicción de texto. Si se da el caso que una "oración" no se encuentra en tu *saco de palabras*, existiría lo que se denomina ***perplejidad*** (del inglés, "perplexity"). Con este término, queremos decir que BoW no es un modelo muy preciso para la predicción del lenguaje, ya que la probabilidad de cierta palabra se relaciona sólo con aquellas utilizadas más frecuentemente.

Si cierto modelo de BoW encuentra que "bueno" aparece con frecuencia en el texto que manejamos, podríamos de entrada suponer que comunica un sentimiento positivo en ese texto . Sin embargo, si en el texto original encontramos que cada aparición de "bueno" viene precedido por un "nada" tendríamos el significado contrario. Los tokens de palabras del modelo BoW carecen de contexto, lo que puede hacer que el significado pretendido de una palabra no esté claro.

Otra cuestión podría ser: "¿Qué sucede si el modelo encuentra una palabra nueva que no estaba en los datos de entrenamiento?" Como se mencionó, BoW sufre, al igual que ocurre en cualquier modelo estadístico, de sobreajuste cuando se trata de vocabulario. Hay varias formas en que los desarrolladores de PLN han abordado este problema. Un enfoque común es mediante el ***suavizado del lenguaje*** en el que se infiere cierta probabilidad de las palabras conocidas como forma de anticipar la que se le otorga a las desconocidas.

Ejemplo

1. Ejecuta el código para un pasaje inventado de Oscar Wilde usando un texto de entrenamiento escrito por él. La herramienta utiliza el modelo estadístico de "n-gramas" con una longitud de secuencia de palabras de 3 para hacer predicciones.

2. Si cambiamos *sequence_length* a 1 para que usemos el modelo de bolsa de palabras. Para los propósitos de esta función, elegiremos cada palabra siguiente al azar de las 20 palabras más comunes.

Ejecuta el código nuevamente para ver cómo se compara el *saco de palabras* con el modelo de-n-gramas en la predicción del lenguaje. No tan bien, ¿verdad?

```
import nltk, re, random

from nltk.tokenize import word_tokenize

from collections import defaultdict, deque, Counter
```

```
from document import oscar_wilde_thoughts


# Change sequence_length:
sequence_length = 1


class MarkovChain:
 def __init__(self):
   self.lookup_dict = defaultdict(list)
   self.most_common = []
   self._seeded = False
   self.__seed_me()


 def __seed_me(self, rand_seed=None):
   if self._seeded is not True:
     try:
       if rand_seed is not None:
         random.seed(rand_seed)
       else:
         random.seed()
       self._seeded = True
     except NotImplementedError:
       self._seeded = False


 def add_document(self, str):
   preprocessed_list = self._preprocess(str)
   self.most_common = Counter(preprocessed_list).most_common(50)
   pairs = self.__generate_tuple_keys(preprocessed_list)
```

```
  for pair in pairs:
    self.lookup_dict[pair[0]].append(pair[1])
 def _preprocess(self, str):
  cleaned = re.sub(r'\W+', ' ', str).lower()
  tokenized = word_tokenize(cleaned)
  return tokenized


def __generate_tuple_keys(self, data):
  if len(data) < sequence_length:
    return


  for i in range(len(data) - 1):
    yield [ data[i], data[i + 1] ]


def generate_text(self, max_length=50):
  context = deque()
  output = []
  if len(self.lookup_dict) > 0:
    self.__seed_me(rand_seed=len(self.lookup_dict))
    chain_head = [list(self.lookup_dict)[0]]
    context.extend(chain_head)
    if sequence_length > 1:
      while len(output) < (max_length - 1):
        next_choices = self.lookup_dict[context[-1]]
        if len(next_choices) > 0:
          next_word = random.choice(next_choices)
          context.append(next_word)
```

```
            output.append(context.popleft())
          else:
            break
        output.extend(list(context))
      else:
        while len(output) < (max_length - 1):
          next_choices = [word[0] for word in self.most_common]
          next_word = random.choice(next_choices)
          output.append(next_word)
    return " ".join(output)


my_markov = MarkovChain()
my_markov.add_document(oscar_wilde_thoughts)
random_oscar_wilde = my_markov.generate_text()
print(random_oscar_wilde)
```

Resumen

Un resumen de todo lo visto podría ser:

- Bolsa de palabras (BoW), también conocido como el modelo de unigrama, es un modelo de lenguaje estadístico basado en el recuento de palabras.
- Hay un montón de aplicaciones del mundo real para BoW.
- BoW se puede implementar como un diccionario de Python con cada clave establecida en una palabra y cada valor establecido en el número de veces que esa palabra aparece en un texto.
- Para BoW, los datos de entrenamiento son el texto que se usa para construir un modelo BoW.
- Los datos de prueba de BoW son el nuevo texto que se convierte en un vector BoW utilizando un diccionario de características entrenadas.
- Un vector de características es una descripción numérica de las características más destacadas de un elemento.

- La extracción de características (o vectorización) es el proceso de convertir el texto en un vector BoW.
- Un diccionario de características es un mapeo de cada palabra única en los datos de entrenamiento a un índice único. Esto se usa para construir vectores BoW.
- BoW tiene menos escasez de datos que otros modelos estadísticos. También sufre menos de sobreajuste.
- BoW tiene una mayor perplejidad que otros modelos, lo que lo hace menos ideal para la predicción de idiomas.
- Una solución al sobreajuste es el suavizado del lenguaje, en el que se toma la probabilidad de las palabras conocidas y se asigna a las desconocidas.

Los datos de spam para esta lección se tomaron del Repositorio UCI Machine Learning.

Dua, D. y Karra Taniskidou, E. (2017). Repositorio de aprendizaje automático de la UCI [http://archive.ics.uci.edu/ml]. Irvine, CA: Universidad de California, Facultad de Información y Ciencias de la Computación.

Fuentes de consulta

3 basic approaches in Bag of Words which are better than Word Embeddings

https://towardsdatascience.com/3-basic-approaches-in-bag-of-words-which-are-better-than-word-embeddings-c2cbc7398016

Trabajar con datos de texto | scikit-learn

https://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html#extracting-features-from-text-files

Text Classification Using Naive Bayes: Theory & A Working Example

https://towardsdatascience.com/text-classification-using-naive-bayes-theory-a-working-example-2ef4b7eb7d5a

Basics of CountVectorizer

https://towardsdatascience.com/basics-of-countvectorizer-e2667790of9c

3.2. Término Frecuencia - Frecuencia inversa del documento

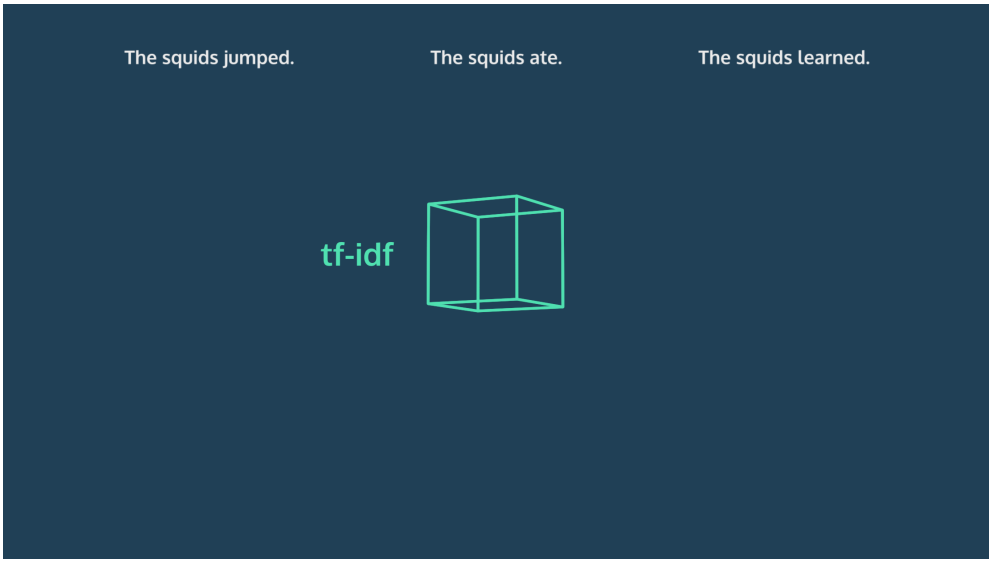
3.2.1. Introducción

Es una noche oscura en medio del invierno mientras imagina que lees poemas de la autora clásica británica Emily Dickinson, poemas que hablan de la inmortalidad. Si te fijas en la elección de palabras en cada poema que lees, intuyes que hay ciertas palabras que son comunes en diferentes poemas y que sugieren un significado particular a su obra conjunta.

Sin darnos cuenta, detrás de esa forma de pensar que guía a la forma en cómo leemos textos subyace un modelo de lenguaje similar a lo que se conoce como *término frecuencia - frecuencia del documento inverso*,en inglés como **tf-idf (“term frequency - inverse document frequency”).** Tf-idf es otra herramienta poderosa en su kit de herramientas de PNL que, por ejemplo, permite:

- clasificación de resultados en un motor de búsqueda
- resúmenes de texto
- construcción de chatbots inteligentes

La imagen a continuación muestra un ejemplo de aplicación de tf-idf a un conjunto de documentos. El resultado de la aplicación de tf-idf es la tabla que se muestra, también conocida como ***matriz término-documento***. Esta *matriz de documento de término* puede verse como una matriz de vectores de tipo *saco de palabras*. Cada columna de la tabla representa un documento único (en este caso, una oración individual). Cada fila representa un token de palabra único. El valor en cada celda representa la puntuación tf-idf para un token de palabra en ese documento en particular.



3.2.2. ¿Qué es Tf-idf?

Término frecuencia-frecuencia inversa de documentos es una operación estadística que se utiliza para indicar el grado de importancia que una palabra tiene en un documento dada una colección de documentos (corpus).

Al aplicar *tf-idf* a un corpus, a cada palabra se le asigna una puntuación *tf-idf* en cada documento, que representa la relevancia de esa palabra para el documento en particular. Una puntuación tf-idf más alta indica que un término es más importante para el documento correspondiente.

Tf-idf tiene muchas similitudes con el *lenguaje de bolsa de palabras*, modelo que, si recuerdas, está relacionado con el recuento de palabras: ambos miden las veces que cada palabra aparece en un documento. Si bien tf-idf se puede usar en cualquier situación en las que se puede usar el *saco de palabras*, ambos se diferencian en cómo se realiza el cálculo.

Tf-idf se basa en dos métricas diferentes para obtener una puntuación general:

- *la frecuencia de los términos,* o frecuencia con la que aparece una palabra en un documento. Es lo mismo que el recuento de palabras de una *bolsa de palabras.*
- *frecuencia inversa del documento*, que es una medida de la frecuencia con la que aparece una palabra en el corpus general. Al penalizar la puntuación de palabras que aparecen a lo largo de un corpus, *tf-idf* puede dar una mejor idea de la importancia de una palabra para un documento particular dentro un corpus.

Ejemplo

```
import pandas as pd
from sklearn.feature_extraction.text import TfidfVectorizer

# Oraciones de pruebas
document_1 = "This is a sample sentence!"
document_2 = "This is a beautiful dog"
document_3 = "Is this a beautiful girl?"

# Corpus con los documentos
corpus = [document_1, document_2, document_3]
print(corpus)

# Instanciamos un objeto 'TfidfVectorizer' y creamos la matriz tf-idf
vectorizer = TfidfVectorizer(norm=None)
tf_idf_scores = vectorizer.fit_transform(corpus)
```

```
# Obtenemos el diccionario de términos
feature_names = vectorizer.get_feature_names_out()
corpus_index = [n for n in corpus]

# Mostramos las puntuaciones tf-idf
pd.DataFrame(tf_idf_scores.T.todense(), index=feature_names,
columns=corpus_index)
```

3.2.3. Frecuencia de términos

El primer componente de *tf-idf* es la ***frecuencia de términos***, o la frecuencia con la que aparece una palabra en un documento dentro del corpus.

El valor del término frecuencia es el mismo que si se aplicara el modelo de lenguaje de *saco de palabras* a un documento. La frecuencia del término indica la frecuencia con la que aparece cada palabra en el documento. La intuición para incluir la frecuencia de términos en el cálculo de *tf-idf* es que cuanto más frecuentemente aparece una palabra en cierto documento, más importante es ese término para el documento.

Considera la siguiente estrofa del poema de Emily Dickinson "*I'm Nobody! Who are you?"*:

```
stanza = '''
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us — don't tell!
They'd banish us, you know.
'''
```

El término frecuencia para "you" es 3, "nobody" es 2, "are" es 2, "us" es 2, y el resto de los términos tienen una frecuencia de 1. Podemos tener una idea general de qué trata esta estrofa mirando las palabras que se usan con más frecuencia.

Como ya sabemos, la frecuencia de un término se puede calcular en Python usando *CountVectorizer* de *scikit-learn*:

```
vectorizer = CountVectorizer()
term_frequencies = vectorizer.fit_transform([stanza])
```

- Se inicializa un objeto *CountVectorizer* .
- El objeto *CountVectorizer* se ajusta (entrena) y se transforma (aplica) para el corpus de datos, devolviendo las frecuencias de término para cada par término-documento.

Ejemplo

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

poem = '''
Success is counted sweetest
By those who ne'er succeed.
To comprehend a nectar
Requires sorest need.

Not one of all the purple host
Who took the flag to-day
Can tell the definition,
So clear, of victory,

As he, defeated, dying,
On whose forbidden ear
The distant strains of triumph
Break, agonized and clear!'''

# Preparamos el texto
processed_poem = preprocess_text(poem)

# Creamos el objeto 'CountVectorizer'
vectorizer = CountVectorizer()
term_frequencies = vectorizer.fit_transform(processed_poem)

# Obtenemos el diccionario con los términos
feature_names = vectorizer.get_feature_names_out()

# Obtenemos la frecuencia de cada término
count_term_frequencies = term_frequencies.toarray().sum(axis=0)

# Mostramos el resumen de ocurrencias para cada término
pd.DataFrame(count_term_frequencies, index=feature_names, columns=["Term
```

```
frequency"])

# Mostramos la matriz de frecuencias
pd.DataFrame(term_frequencies.toarray(), columns=feature_names).head(5)
```

3.2.4. Frecuencia inversa de documentos

La ***frecuencia inversa de documentos*** de la puntuación *tf-idf* penaliza aquellos términos que aparecen con más frecuencia en un corpus. Esto se basa en la intuición de que las palabras que aparecen con más frecuencia en el corpus dan menos información sobre el tema o el significado de un documento individual y, por lo tanto, se les debe dar menor prioridad.

Por ejemplo, términos como "el" o "ir" se utilizan en todas partes, por lo que en un modelo de *saco de palabras,* se les otorgaría una alta prioridad aunque no proporcionaran mucho significado; *tf-idf*, por contra, resta prioridad a la repetición de este tipo de palabras.

Podemos calcular la **frecuencia inversa** del documento para algún término t en un corpus usando la siguiente ecuación:

$$log(\frac{Total\ number\ of\ documents}{Number\ of\ documents\ with\ term\ t})$$

La conclusión que puede extraerse de la ecuación es que a medida que aumenta el número de documentos con el término *t*, la frecuencia inversa del documento disminuye (debido a la naturaleza de la función logarítmica). Cuanto más frecuentemente aparece un término en el corpus, menos importante se vuelve para un documento individual.

La frecuencia inversa del documento se puede calcular en un grupo de documentos usando el *TfidfTransformer* de scikit-learn:

```
transformer = TfidfTransformer (norm = None)

transformer.fit (term_frequencies)

inverse_doc_frequency = transformer.idf_
```

- Se inicializa un objeto ***TfidfTransformer***.
- el *TfidfTransformer* se ajusta (entrena) en una matriz término-documento de frecuencias para los diferentes términos.

- el **atributo .idf_** del *TfidfTransformer* guarda la frecuencia inversa por documento de los términos como una matriz *NumPy.*

```
import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer

# Instanciamos 'CountVectorizer' y generamos la matriz de frecuencias
vectorizer = CountVectorizer()
term_frequencies = vectorizer.fit_transform(poems)

# Obtenermos el diccionario de términos
feature_names = vectorizer.get_feature_names_out()

# Obtenermos los índices del corpus
corpus_index = [f"Poem {i+1}" for i in range(len(poems))]

# Mostramos la matriz de frecuencias
pd.DataFrame(term_frequencies.T.todense(), index=feature_names,
columns=corpus_index)
```

3.2.5. Combinando cada parte: tf-idf

Ahora que entendemos cómo se calculan la *frecuencia de términos* y la *frecuencia inversa del documento*, juntaremos ambas partes.

Las puntuaciones de *tf-idf* se calculan sobre la base de cada término por documento. Eso significa que hay una puntuación *tf-idf* para cada palabra, en cada documento. La puntuación tf-idf para un término *t* en un documento *d* de un corpus se calcula de la siguiente manera:

*tfidf (t, d) = tf (t, d) * idf (t, corpus)*

- **tf (t, d)** es la frecuencia del término *t* en el documento *d*.
- **idf (t, corpus)** es la frecuencia inversa de un término t en todo el corpus.

Podemos calcular fácilmente los valores *tf-idf* para cada par término-documento en nuestro corpus usando el ***TfidfVectorizer*** de scikit-learn:

```
vectorizer = TfidfVectorizer (norm = None)
```

```
tfidf_vectorizer = vectorizer.fit_transform (corpus)
```

- Se inicializa el objeto ***TfidfVectorizer***. El argumento *norm* = *None* evita que scikit-learn modifique la multiplicación de la frecuencia del término y la frecuencia inversa del documento.
- El objeto *TfidfVectorizer* se **ajusta y se transforma** en el corpus de datos, devolviendo las puntuaciones ***tf-idf*** para cada par término-documento.

Ejemplo

```
from sklearn.feature_extraction.text import TfidfVectorizer

# Instanciamos 'tfidf_vectorizer'
tfidf_vectorizer = TfidfVectorizer(use_idf=True)

# Generamos la matriz 'tf-idf'
tfidf_vectorizer_vectors = tfidf_vectorizer.fit_transform(poems)

# Obtenemos los índices del corpus
corpus_index = [f"Poem {i+1}" for i in range(len(poems))]

# Mostramos los valores 'tf-idf'
pd.DataFrame(tfidf_vectorizer_vectors.T.todense(),
index=tfidf_vectorizer.get_feature_names_out(), columns=corpus_index)
```

3.2.6. Conversión de *saco de palabras* a *tf-idf*

Además de calcular directamente las puntuaciones de *tf-idf* para un conjunto de términos en un corpus, también puede convertir un modelo de *bolsa de palabras* previamente creado en puntuaciones *tf-idf*. *TfidfTransformer* de Scikit-learn tiene la capacidad de convertir un modelo de *saco de palabras* a *tf-idf*.

En este caso, empezaríamos inicializando un objeto TfidfTransformer.

tf_idf_transformer = TfidfTransformer (norm = False)

Dada una matriz de *saco de palabras* ***count_matrix***, ahora se puede multiplicar las frecuencias de términos por su frecuencia inversa de documento para obtener las puntuaciones *tf-idf*:

tf_idf_scores = tfidf_transformer.fit_transform (count_matrix)

Esto resulta similar a cómo calculamos la frecuencia inversa del documento, excepto que esta vez estamos ajustando *y* transformando el TfidfTransformer a las frecuencias del término (vectores de *saco de palabras*) en lugar de simplemente ajustar el TfidfTransformer.

Resumen

Los siguientes serían los principales aspectos cubiertos en este apartado:

- ***Frecuencia de términos inversa de documentos***, conocida como *tf-idf*, es una operación estadística que se utiliza para indicar cuán importante es una palabra para cada documento dada una colección de documentos.
- **tf-idf** consta de dos componentes:
    - *frecuencia de términos*, que es la frecuencia en la que aparece una palabra en un documento.
    - *frecuencia inversa del documento* es una medida de la frecuencia con la que aparece una palabra en todos los documentos de un corpus.
- *Tf-idf* se calcula como la frecuencia del término multiplicado por la frecuencia inversa del término en el documento
- la frecuencia de un término, la frecuencia inversa de un documento, y el *tf-idf* se pueden calcular en scikit-learn usando los objetos *CountVectorizer*, *TfidfTransformer* y *TfidfVectorizer*, respectivamente.

Fuentes de consulta

How to Use Tfidftransformer & Tfidfvectorizer

https://kavita-ganesan.com/tfidftransformer-tfidfvectorizer-usage-differences/#.Ycbye6SCHV1

Topic Model Visualization using pyLDAvis

https://towardsdatascience.com/topic-model-visualization-using-pyldavis-fecd7c18fbf6

Improving the Interpretation of Topic Models

https://towardsdatascience.com/improving-the-interpretation-of-topic-models-87fd2ee3847d

3.3. Incrustaciones de palabras ("Word Embeddings")

3.3.1. Introducción

Hay un refrán famoso del escritor alemán Johann Wolfgang von Goethe que dice:

"Dime con quién vas y te diré quién eres ..."

La suposición de Goethe es que las personas con las que pasas tiempo todos los días son un reflejo de quién eres como persona. Ahora bien, ¿qué pasa si ampliamos esa misma idea de las personas a las palabras?

El lingüista John Rupert Firth dio este paso y se le ocurrió otro refrán:

"Conocerás una palabra por aquellas que le acompañan".

Esta idea de que el significado de una palabra puede entenderse por su contexto, o por las palabras que la rodean, es la base de la idea de "incrustación de palabras". Una ***incrustación de palabras*** es una representación de una palabra como un vector numérico, lo que nos permite comparar y contrastar cómo se usan las palabras e identificar palabras que ocurren en contextos similares.

Las aplicaciones de las incrustaciones de palabras, por ejemplo, incluyen:

- reconocimiento de entidades en chatbots
- análisis de sentimiento
- análisis sintáctico

Ejemplo

```
# Importamos 'SpaCy'
import spacy
from scipy.spatial.distance import cosine

# Cargamos modelo en inglés
nlp = spacy.load('en')

# Definimos los vectores
summer_vec = nlp("summer").vector
winter_vec = nlp("winter").vector
```

```
# Comparamos similitud
print(f"The cosine distance between the word embeddings for 'summer' and
'winter' is: {cosine(summer_vec, winter_vec)}\n")

# Definimos los vectores
mustard_vec = nlp("mustard").vector
amazing_vec = nlp("amazing").vector

# Comparamos similitud
print(f"The cosine distance between the word embeddings for 'mustard' and
'amazing' is: {cosine(mustard_vec, amazing_vec)}\n")

# Mostramos las incrustaciones de los términos
print(f"'summer' in vector form: {summer_vec}")
print(f"'winter' in vector form: {winter_vec}")
print(f"'mustard' in vector form: {mustard_vec}")
print(f"'amazing' in vector form: {amazing_vec}")
```

3.3.2. Vectores

Los vectores pueden ser muchas cosas en muchos campos diferentes, pero en última instancia son contenedores de información. Según el tamaño o la dimensión de un vector, se puede contener distintas cantidades de datos.

El caso más simple es un vector unidimensional, que almacena un solo número. Digamos que queremos representar la longitud de una palabra con un vector. Podemos hacerlo de la siguiente manera:

- "cat" ----> [3]
- "scrabble" ----> [8]
- "antidisestablishmentarianism" ----> [28]

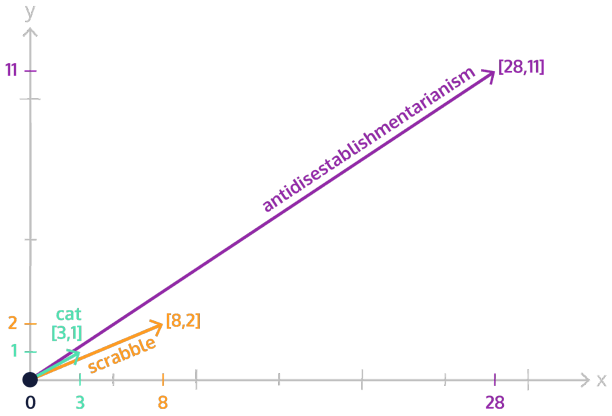
Una forma de comparar estas tres palabras es comparar los vectores que los representan ubicándolos en una recta numérica.

Podemos ver claramente que el vector "cat" es mucho más pequeño que el vector "scrabble", que es mucho más pequeño que el vector "antidisestablishmentarianism".

Ahora digamos que también queremos registrar el número de vocales en nuestras palabras, además del número de letras. Podemos hacerlo usando un vector bidimensional, donde la primera entrada es la longitud de la palabra y la segunda entrada es el número de vocales:

- "cat" ----> [3, 1]
- "scrabble" - -> [8, 2]
- "antidisestablishmentarianism" ----> [28, 11]

Para ayudar a visualizar estos vectores, podemos trazarlos en una cuadrícula bidimensional, donde el eje x es el número de letras, y el eje y es el número de vocales.



Aquí podemos ver que los vectores de "cat" y "scrabble" apuntan a un área más similar de la cuadrícula que la del vector de "antidisestablishmentarianism". Así que podríamos argumentar que "gato" y "scrabble" están más juntos.

Si bien hemos mostrado aquí solo vectores unidimensionales y bidimensionales, podemos tener vectores en cualquier número de dimensiones. La parte complicada, sin embargo, es visualizarlos.

Los vectores son útiles porque nos ayudan a resumir información sobre un objeto usando números. Luego, usando la representación numérica, podemos hacer comparaciones entre las representaciones vectoriales de diferentes objetos. Esta idea es fundamental para entender cómo las *incrustaciones de palabras* mapean palabras en vectores.

Podemos representar vectores fácilmente en Python usando matrices *NumPy*. Para crear un vector que contenga los números impares del 1 al 9, podemos usar el método .array () de NumPy:

odd_vector = np.array([1, 3, 5, 7, 9])

Ejemplo

Supongamos que estás trabajando en una escuela y deseas como profesor/a analizar las calificaciones de los estudiantes en los dos primeros exámenes de inglés del semestre.

Un estudiante, Xavier, obtuvo 88 en el primer examen y 92 en el segundo. Otro estudiante, Niko, obtuvo 94 en el primer examen y 87 en el segundo. Podríamos representar como vectores las puntuaciones de los exámenes de los estudiantes.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn

# define score vectors
scores_xavier = np.array([88,92])
scores_niko = np.array([94,87])
scores_alena = np.array([90,48])

# plot vectors
try:
 plt.arrow(0, 0, scores_xavier[0], scores_xavier[1], width=1, color='blue')
except:
 pass
try:
 plt.arrow(0, 0, scores_niko[0], scores_niko[1], width=1, color='orange')
except:
 pass
try:
 plt.arrow(0, 0, scores_alena[0], scores_alena[1], width=1, color='purple')
except:
 pass
plt.axis([0, 100, 0, 100])
```

```
plt.show()
```

3.3.3. ¿Qué es una *incrustación de palabras*?

Ahora que comprendemos los vectores, podemos volver a las *incrustaciones de palabras*. Las *incrustaciones de palabras* son representaciones vectoriales de una palabra. Nos permiten tomar toda la información que está almacenada en una palabra, ya sea su significado o su parte gramatical, y convertirla en una forma numérica más comprensible para un ordenador.

Por ejemplo, esto sería una palabra incrustada para "paz".

[5.2907305, -4.20267, 1.6989858, -1.422668, -1.500128, ...]

Aquí la "paz" se representa mediante un vector de 96 dimensiones, de las cuales hemos mostrado sólo las primeras. Cada dimensión del vector captura información sobre cómo se usa la palabra "paz". También podemos mirar una *incrustación de palabras* para la palabra "war":

[7.2966490, -0.52887750, 0.97479630, -2.9508233, -3.3934135, ...]

Al convertir las palabras "paz" y "guerra" en sus representaciones vectoriales numéricas, podemos conseguir que un ordenador compare los vectores más fácilmente y comprenda sus similitudes y diferencias.

La librería *spaCy* proporciona modelos de modelos básico de *incrustación de palabras* para varios idiomas:

```
import spacy
```

Nota: la convención es cargar modelos *spaCy* en una variable llamada nlp.

Para obtener la representación vectorial de una palabra, llamamos al modelo con la palabra deseada como argumento y podemos usar el atributo **.vector.**

```
nlp = spacy.load("en")
nlp("cat").vector
```

Output:

```
array([ 1.4532998 , 0.47115922, 0.71999407, 1.1403103 , 3.1126573 ,
    0.6275163 , 2.1985044 , 3.1114974 , 1.0527445 , 3.2556517 ,
    3.7162375 , -0.10062158, 2.56286   , -3.1346574 , 1.2517695 ,
   -0.56736076, -2.0715995 , 1.2972176 , -1.0384768 , -2.8220317 ,
    0.4024135 , 1.1475914 , -0.9666666 , -1.8358834 , 0.5618948 ,
```

```
-1.109226 , 3.1648145 , -3.0343432 , 1.5009679 , 1.8909831 ,
 2.1640768 , -0.868277 , 1.5119176 , 1.7287943 , 2.2542849 ,
-1.7297868 , 1.8519063 , 0.29412246, -3.1516666 , 2.2560802 ,
 2.8821528 , 0.9065895 , -1.4355452 , -4.937831 , 0.07267573,
-1.5819955 , 0.7202336 , -1.0646656 , -0.19294381, -0.21828318,
-0.21474707, -0.25848413, -0.41141343, -1.650431 , -1.6427782 ,
 2.0982878 , 0.8212584 , 2.458171 , -0.09555507, -0.8922238 ,
 0.7312181 , -1.4605643 , -0.46285537, -1.5074668 , -2.750762 ,
-3.9613488 , 1.5867741 , -5.1115217 , -0.21341431, 0.9157888 ,
-2.3833017 , 2.126943 , 2.4008074 , 0.91968673, -1.3888277 ,
-1.4603075 , 2.4146914 , -0.75423056, -1.8709452 , 0.6487465 ,
-0.8690753 , -1.9576063 , -1.8812385 , -0.77405465, 2.5276294 ,
 3.4152527 , -1.2771642 , -1.703851 , -0.6723095 , -2.5397158 ,
-0.1799444 , -0.87516195, -2.0097623 , 3.7264547 , 1.2048597 ,
-1.2384005 ], dtype=float32)
```

Pero, ¿cómo comparamos estos vectores? ¿Y cómo llegamos a estas representaciones numéricas?

3.3.4. Distancias

La cuestión clave en las *incrustaciones de palabras* es la idea de distancia. Antes de explicar por qué, veamos cómo se puede medir la distancia entre vectores.

Hay diferentes formas de encontrar la distancia entre vectores, por ejemplo:

1. En **la distancia de Manhattan**, también conocida como distancia entre manzanas en una ciudad, la distancia se define como la suma de las diferencias en cada dimensión individual de los vectores. Imaginemos que tenemos dos vectores [1, 2, 3] y [2, 4, 6]. Podemos calcular la distancia de Manhattan entre ellos como se muestra a continuación:

   distancia de Manhattan = $|1-2| + |2-4| + |3-6| = 1 + 2 + 3 = 6$

2. Otra métrica de distancia común se llama **distancia euclidiana**, también conocida como distancia en línea recta. En esta métrica de distancia, sacamos la raíz cuadrada de la suma de los cuadrados

   $$euclidean\ distance = \sqrt{(1-2)^2) + (2-4)^2) + (3-6)^2)} = \sqrt{14} \approx 3.74$$

   de las diferenc
   ias en cada dimensión.

3. La distancia final que consideraremos es la **distancia del coseno**. La distancia del coseno se refiere al ángulo entre dos vectores. Dos vectores que apuntan en la misma dirección no tienen ningún

ángulo entre ellos y tienen una distancia de coseno de 0. Dos vectores que apuntan en direcciones opuestas, por otro lado, tienen una distancia de coseno de 1 (puedes leer sobre el cálculo aquí).

Podemos calcular fácilmente las distancias de Manhattan, euclidiana y coseno entre vectores utilizando funciones auxiliares de la librería SciPy:

```
from scipy.spatial.distance import cityblock, euclidean, cosine

vector_a = np.array([1,2,3])
vector_b = np.array([2,4,6])

# Manhattan distance:
manhattan_d = cityblock(vector_a,vector_b) # 6

# Euclidean distance:
euclidean_d = euclidean(vector_a,vector_b) # 3.74

# Cosine distance:
cosine_d = cosine(vector_a,vector_b) # 0.0
```

Cuando se trabaja con vectores que tienen un gran número de dimensiones, como los son los asociados a las *incrustaciones de palabras*, las distancias calculadas por Manhattan y la distancia euclidiana pueden llegar a ser bastante grandes. Por lo tanto, se prefieren los cálculos que utilizan la distancia del coseno.

Ejemplo

```
import numpy as np
from scipy.spatial.distance import cityblock, euclidean, cosine
import spacy

# load word embedding model
nlp = spacy.load('en')

# define word embedding vectors
happy_vec = nlp('happy').vector
sad_vec = nlp('sad').vector
angry_vec = nlp('angry').vector

# calculate Manhattan distance
man_happy_sad = cityblock(happy_vec, sad_vec)
man_sad_angry = cityblock(sad_vec, angry_vec)
print("Happy vs Sad Manhattan distance: {}".format(man_happy_sad))
```

```
print(f"Sad vs Angry distance: {man_sad_angry}")

# calculate Euclidean distance
euc_happy_sad = euclidean(happy_vec, sad_vec)
euc_sad_angry = euclidean(sad_vec, angry_vec)
print("\nHappy vs Sad Euclidean distance: {}".format(euc_happy_sad))
print(f"Sad vs Angry Euclidean distance: {euc_sad_angry}")

# calculate cosine distance
cos_happy_sad = cosine(happy_vec, sad_vec)
cos_sad_angry = cosine(sad_vec, angry_vec)
print("\nHappy vs Sad cosine distance: {}".format(euc_happy_sad))
print(f"Sad vs Angry cosine distance: {euc_sad_angry}")
```

La idea que subyace a *las incrustaciones de palabras* es una teoría conocida como **hipótesis de distribución.** Esta hipótesis establece que las palabras que coexisten en los mismos contextos tienden a tener significados similares. Con las *incrustaciones de palabras*, asignamos palabras que existen con el mismo contexto a lugares similares en nuestro espacio vectorial.

Los valores numéricos que se asignan a la representación vectorial de una palabra no son importantes en sí mismos, pero adquieren significado a partir de cuán similares o no son las palabras entre sí. Por tanto, la distancia de coseno entre palabras con contextos similares será pequeña y aquella entre palabras que tienen contextos muy diferentes será grande.

Los valores literales de la *incrustación de una palabra* no tienen un significado real. Obtenemos cierto significado en base a las *incrustaciones de palabras* al comparar los diferentes vectores de palabras y comprobar su grado de similitud. Sin embargo, en estos vectores se codifica información latente sobre cómo se utilizan.

3.3.5. Word2vec

Lo anterior suscita la siguiente pregunta. ¿Cómo llegamos a los valores vectoriales que definen un vector de palabra? ¿Y cómo nos aseguramos de que los valores elegidos sitúen los vectores de palabras que comparten un contexto similar juntos y aquellos con diferentes usos muy separados? Una forma es usar *Word2vec*.

Word2vec es un algoritmo de aprendizaje estadístico que desarrolla *incrustaciones de palabras* a partir de un corpus de texto. Word2vec utiliza uno de los dos siguientes modelos a la hora de generar los valores que definen una colección de *incrustaciones de palabras*.

1. Un método consiste en utilizar una representación de *saco de palabras continuo* (CBOW) para cierto fragmento de texto. El modelo *Word2vec* revisa cada palabra en el corpus de entrenamiento, en orden, e intenta predecir qué palabra viene en cada posición basándose en una aplicación del *saco de palabras* a las palabras que acompañan la palabra en cuestión. En este enfoque, el orden de las palabras no importa.
2. El otro método que *Word2vec* puede utilizar para crear i*ncrustaciones de palabras* es el de *saltos-de-gramas* ("skip-grams", en inglés). Los *saltos-de-gramas* funcionan de manera similar a los n-gramas, excepto que en lugar de mirar agrupaciones de *n* palabras consecutivas en un texto, revisa secuencias de palabras que están separadas por una distancia especificada entre ellas.

Por ejemplo, si tuviéramos la frase en inglés *"Los calamares saltan de la maleta"*. El "1-salto-2-gramas" incluye todos los bigramas (2-gramas) así como las siguientes subsecuencias:

(Los, saltan), (calamares, de), (saltan, la), (de , maleta)

Cuando usamos los "saltos-de-gramas" continuos, se tiene en cuenta el contexto. Debido a esto, el tiempo que se tarda en entrenar las *incrustaciones de palabras* es más lento que cuando se usa una bolsa de palabras contínua. Sin embargo, los resultados suelen ser mucho mejores.

Con las representaciones *sacos de palabras continuas* y *saltos-de-gramas continuos* como datos de entrenamiento, *Word2vec* utiliza una red neuronal superficial de 2 capas para generar los valores que colocan palabras con un contexto similar en vectores cercanos entre sí y palabras con diferentes contextos en vectores muy separados entre sí.

Ejemplo

1. En la parte superior del siguiente tienes la primera oración de la novela de Charles Dickens *Historia de dos ciudades*, almacenada en una variable. A continuación, verás que esta oración se procesa previamente y guarda como *statement_lst*.

2. También verás una función llamada *get_cbows()*, que itera a través de una oración preprocesada palabra por palabra y devuelve una lista de las diferentes representaciones de bolsa de palabras para las palabras de contexto que rodean a cada palabra en la oración.

3. También incluye una función llamada *get_skip_grams()*, que itera a través de una oración palabra por palabra y devuelve una lista ordenada de las diferentes palabras de contexto que rodean cada palabra en la oración.

4. Además existe la variable *context_length*. Con ella, se indica que al encontrar nuestras representaciones de *saco de palabras y salto-de-grama*, solo estamos mirando 2 palabras a la izquierda y 2 palabras a la derecha de la palabra que estamos manejando.

```
from sklearn.feature_extraction.text import CountVectorizer
sentence = "It was the best of times, it was the worst of times."
print(sentence)

# preprocessing
sentence_lst = [word.lower().strip(".") for word in sentence.split()]

# set context_length
context_length = 3

# function to get cbows
def get_cbows(sentence_lst, context_length):
cbows = list()
for i, val in enumerate(sentence_lst):
  if i < context_length:
    pass
  elif i < len(sentence_lst) - context_length:
    context = sentence_lst[i-context_length:i] +
sentence_lst[i+1:i+context_length+1]
    vectorizer = CountVectorizer()
    vectorizer.fit_transform(context)
    context_no_order = vectorizer.get_feature_names_out()
    cbows.append((val,context_no_order))
return cbows

# define cbows here:
cbows = get_cbows(sentence_lst, context_length)

# function to get cbows
def get_skip_grams(sentence_lst, context_length):
skip_grams = list()
for i, val in enumerate(sentence_lst):
  if i < context_length:
    pass
  elif i < len(sentence_lst) - context_length:
    context = sentence_lst[i-context_length:i] +
sentence_lst[i+1:i+context_length+1]
    skip_grams.append((val, context))
return skip_grams

# define skip_grams here:
```

```
skip_grams = get_skip_grams(sentence_lst, context_length)
try:
print('\nContinuous Bag of Words')
for cbow in cbows:
  print(cbow)
except:
pass
try:
print('\nSkip Grams')
for skip_gram in skip_grams:
  print(skip_gram)
except:
pass
```

Dependiendo del corpus de texto que seleccionemos para entrenar un modelo de *incrustación de palabras*, tendremos diferentes tipos de incrustaciones de palabras de acuerdo con el contexto de las palabras en el corpus dado. Sin embargo, cuanto más grande y genérico es un corpus, más generalizable se vuelve el resultado del proceso.

Cuando queremos entrenar nuestro propio modelo *Word2vec* en un corpus de texto, podemos usar el **paquete gensim**.

Para entrenar un modelo de *Word2vec* en un corpus de texto propio, podemos usar la función *Word2Vec()* de *gensim*.

```
model = gensim.models.Word2Vec(corpus, size=100, window=5, min_count=1, workers=2, sg=1)
```

- ***corpus*** es una lista de listas, donde cada lista interna es un documento en el corpus y cada elemento en las listas internas hay token (palabra).
- ***tamaño*** determina cuántas dimensiones incluirán nuestras *incrustaciones de palabras*. Las *incrustaciones de palabras* suelen tener más de 1000 dimensiones. Aquí crearemos vectores de 100 dimensiones para simplificar las cosas.

Para ver todo el vocabulario usado para entrenar el modelo de *incrustación de palabras*, podemos usar el método ***.wv.vocab.items()***.

```
vocabulary_of_model = list(model.wv.vocab.items())
```

Por ejemplo, si tuviéramos los guiones de la serie televisiva *Friends* como un corpus de entrenamiento, el modelo sería capaz de identificar los usos específicos de las palabras en el programa. Si bien es posible que

los vectores generalizados en un modelo espacial no coloque juntos los vectores relativos a dos personajes que tiene una relación estrecha como "Ross" y "Rachel", un modelo de *incrustación de palabras gensim* entrenado en los guiones de *Friends* sí que podría hacerlo.

Para encontrar fácilmente qué vectores *gensim* colocó juntos en tu modelo de incrustación de palabras, podemos usar el método ***.most_similar()*.**

```
model.most_similar("my_word_here", topn=100)
```

- **"my_word_here"** es el token de la palabra de destino del cual queremos encontrar palabras más similares.
- **topn** es un argumento que indica cuántos vectores de palabras similares queremos que se devuelvan.

Un último método gensim que exploraremos es: ***.doesnt_match().***

```
model.doesnt_match(["asia", "mars", "pluto"])
```

- cuando se le da una lista de términos en el vocabulario como argumento, *.doesnt_match()* devuelve qué término está más alejado de los demás.

Resumen

Los siguientes serían los principales aspectos cubiertos en este apartado:

- *Los vectores* son contenedores de información y pueden tener desde 1 dimensión hasta cientos o miles de dimensiones.Las
- *incrustaciones de palabras* son representaciones vectoriales de una palabra, donde las palabras con contextos similares se representan con vectores que están más juntos
- *spaCy* es un paquete que nos permite para ver y usar modelos de i*ncrustación de palabras* previamente entrenados
- La distancia entre vectores se puede calcular de muchas formas, y la mejor manera de medir la distancia entre vectores de dimensiones superiores es la *distancia del coseno*
- *Word2Vec* es un modelo de red neuronal poco profunda que puede construir *incrustaciones de palabras* usando ya sea bolsa de palabras continua o saltos de gramas continuos
- *Gensim* es un paquete que nos permite crear y entrenar modelos de incrustación de palabras usando cualquier corpus de texto