UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA MATEMÁTICA

ESTUDIANDO OBRAS LITERARIAS CON HERRAMIENTAS DE PROCESAMIENTO DE LENGUAJE NATURAL

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL MATEMÁTICO

ROMAIN VICTOR OLIVIER GOURON

PROFESOR GUÍA:
FELIPE TOBAR HENRÍQUEZ

MIEMBROS DE LA COMISIÓN:
IVÁN RAPAPORT ZIMERMANN
JORGE PÉREZ ROJAS

SANTIAGO DE CHILE
2017

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL MATEMÁTICO
POR: ROMAIN VICTOR OLIVIER GOURON
FECHA: 2017
PROF. GUÍA: SR. FELIPE TOBAR HENRÍQUEZ

ESTUDIANDO OBRAS LITERARIAS CON HERRAMIENTAS DE PROCESAMIENTO DE LENGUAJE NATURAL

En los últimos años, el procesamiento de lenguaje natural (*Natural Language Processing*, o NLP) ha experimentado importantes avances. Específicamente, en 2013 *Google* lanzó "*word2vec*", un algoritmo que propone, a partir de un corpus dado, una representación vectorial de las palabras que lo componen. Dicho algoritmo ha tenido un gran éxito principalmente por dos razones: La primera es el bajo costo computacional de su entrenamiento que permitió un uso masivo, mientras que la segunda es la intuitiva topología inducida por la representación vectorial ilustrada por el popular ejemplo:

```
word2vec("king") - word2vec("man") + word2vec("woman")  = word2vec("queen")
```

En esta memoria, presentamos en un primer lugar un ejemplo ilustrativo del algoritmo "word2vec"mediante su implementación para determinar preguntas duplicadas en *Quora*, una competencia propuesta por el sitio *Kaggle.com*. Una vez familiarizados con el algoritmo, nos enfocamos en un problema más abierto que considera el análisis de 45 obras de literatura francesa. En particular, queremos atacar la siguiente pregunta: ¿cómo se puede definir una distancia entre dos libros? Después de haber preparado los libros con el propósito de poder usar el algoritmo, propondremos varios métodos originales para comparar pares de libros. Luego, nos interesará representar estas obras en un espacio, y determinar si dicha representación revela propiedades literarias de las obras consideradas tales como la paternidad o el estilo literario.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

El procesamiento de lenguaje natural (*Natural Language Processing*, o NLP), es una área de *Machine Learning* que aspira a extraer información de un texto escrito por un humano. Esa tecnología sirve, entre otros, para detectar *spam* en correos electrónicos, o para traducir textos automáticamente.

1.1. Historia

Las tentativas de concepción de máquinas de traducción preceden de muchos años la invención del primer computador, sin embargo esa área de interés ganó mucha importancia durante los años de la Segunda Guerra Mundial (WW2), cuando la interpretación de mensajes encriptados era de importancia capital. El estadounidense Warren Weaver, científico considerado como uno de los pioneros del dominio, y implicado en desciframiento durante WW2, lideó después la investigación para fines pacíficos. Su punto de vista era que, dado que todos los humanos son bastante parecidos, un documento en un idioma puede ser considerado como haber sido escrito en código: por ejemplo, en este sentido, Alemán no sería más que Inglés encriptado.

Un primer hito fue la publicación del artículo *Syntactic Structures* por Noam Chomsky (1957), que marcó la separación entre sintaxis y semántica con la frase ahora famosa *"Colourless green ideas sleep furiously"*: esa frase no tiene sentido semántico, pero es gramaticalmente correcta. Más generalmente, ese artículo introduce un método de reconocimiento de estructura en una frase – independientemente del lenguaje usado – que influenció todo el trabajo en NLP desde entonces.

La voluntad de hacer de la traducción automática un dominio real de investigación se difundió, y grupos de investigadores se formaron en el Reino Unido, Francia, EE.UU. y Unión Soviética. Los sistemas estadounidenses se enfocaron en el idioma alemán, con el fin de terminar la traducción de documentos técnicos sobrantes de la guerra. Sin embargo, después de algunos años, la investigación se orientó hacía los pares de lenguaje Ruso-Inglés y Ruso-Francés: la Guerra Fría había dado mucha inspiración a la investigación. Sin embargo, a pesar

de esa emulación internacional, el área no conoció el desarrollo esperado: los investigadores se dieron cuenta de la inmensa complejidad de la tarea.

Por eso, se propuso limitar, en un primer tiempo, la investigación a traducciones automáticas asistidas por humanos, donde un ayudante tenía que marcar las partes ambiguas que un solo computador no podía entender. De manera más realista, se aceptó luego la idea que el rol de los computadores se limitaría a diccionarios extendidos: la mayoría de los proyectos de entonces consistía en pares de palabras, así que se admitió que *Fully-Automatic-High-Quality-Translation* (FAHQT) quedaba fuera de alcance, y en 1960, un reporte del *Automatic Language Processing Advisory Committee* (ALPAC) se pronunció negativamente sobre los progresos en el dominio, comentando que no se había logrado ninguna traducción automática de documentos científicos, ni una en próxima perspectiva. Así se congelaron los fundos en los Estados Unidos, y la investigación no conoció grandes avances durante años.

Fue no antes que en los años 1980 que el aprendizaje de máquinas pudo abordar el problema desde un punto de vista estadístico, y modelos markovianos, entre otros, surgieron. Años después, la primera tentativa de representación distribuida de palabras fue publicado por Bengio et al. (2003), y fue democratizada 10 años después por un equipo de *Google* (Mikolov, Sutskever, et al., 2013), bajo el nombre de *word2vec*.

Haciendo eco al artículo *Computing Machinery and Intelligence* de Turing (1950) que introduce el famoso "test de Turing" como criterio de inteligencia artificial, Mikolov et al. (2015) exponen en "*A Roadmap towards Machine Intelligence*" su punto de vista sobre el desarrollo de la inteligencia artificial, abordando las propiedades fundamentales que deben tener las máquinas. Según ellos, esas deberían enfocarse en los dos puntos siguientes: comunicación y aprendizaje. En efecto, "*una máquina capaz de desempeñar tareas complejas sin necesidad de programación laboriosa sería tremendamente útil en cualquier esfuerzo humano*". Por eso, en los últimos años, NLP ha ocupado un lugar cada vez más importante en aprendizaje de máquinas.

1.2. Estado del arte y ejemplos

1.2.1. Estado del arte

Lo que ya pueden hacer los algoritmos:

1. Reconocer nombres de entidad, por ejemplo *ONU, USA, Einstein, ...* Eso se llama *name-entity recognition*

2. Reconocer, en la mayoría de los casos, la estructura de una frase: sujeto, verbo, complemento. Esto se llama *part-of-speech tagging*

Lo que queda bastante difícil:

1. Reconocer la equivalencia entre paráfrasis. Por ejemplo: *"ABC compró XYZ"* y *"XYZ*

fue adquirido por ABC"

2. Despejar las ambigüedades, como en el ejemplo siguiente:

   "Fed raises interest rates"

   Aquí, el algoritmo tiene dificultades para saber si el verbo es *raises* – y el sujeto *Fed* – o *interest* – y el sujeto *Fed raises*

3. El análisis de sentimiento, que aspira a determinar si el sentimiento llevado por un texto es positivo, neutro o negativo, y posiblemente etiquetarlo con más precisión (tristeza, hostilidad, incomprensión, etc). Este campo del NLP es muy usado esos días, entre otros por las empresas que quieren analizar la opinión de sus clientes (en *Twitter*, por ejemplo).

Lo que no logran hacer todavía:

1. Entender y contestar preguntas precisas. Eso sería el último paso antes de llegar a la comunicación que menciona Mikolov.

1.2.2. Ejemplos de aplicación

Mencionemos algunas aplicaciones interesantes de los recientes años:

Musicalización de libros : mediante un análisis de sentimiento, el programa *TransProse* (Davis, 2014) pretende crear un ambiente musical para un texto dado, variando el ritmo, la intensidad, y los intervalos musicales usados según el sentimiento que predomina.

Resumen de textos : sacar las ideas importantes de un texto, que sea un contrato u otro documento legal, o un artículo de prensa sería una gran ganancia de tiempo. Hay varios laboratorios que se dedican a crear algoritmos que abordan esa tarea, entre los cuales *Alchemy API* de IBM (2005).

Ofertas laborales : la empresa *Textio* (2014) propone una herramienta que busca mejorar la relevancia de las ofertas de trabajo. El entrenamiento se hace supervisando el análisis de contenido de ofertas con el tiempo medio entre la publicación de la oferta y la contratación de un candidato. Así, se propone evaluar la calidad de la redacción de ofertas de trabajo de las empresas.

Bot conversacional (o *chatbot*): con el uso creciente de herramientas de conversación (como *WhatsApp* (2009) o *Messenger* (2011), ambas de *Facebook*), la interacción entre humanos y robots es posible mediante varios canales. Ahora, varias empresas proponen un servicio adaptado a ese tipo de comunicación, como *Citron* (2017) que busca dar recomendaciones de restaurantes en Paris, o *Poncho* (2016), un experto en pronósticos meteorológicos.

En 2016, hubieron más de 34.000 bots creados en la plataforma dedicada de *Messenger*.

Ese aumento se puede explicar por la facilidad de utilización y la ausencia de instalación en comparación con una aplicación clásica.

1.3. Contribución

En lo que sigue, después de haber presentado unos métodos que se pueden usar en NLP, introduciremos la herramienta muy útil que es *word2vec* con un ejemplo ilustrativo. Por fin, se tratara de atacar un problema más abierto, que considera el análisis de 45 obras de literatura francesa, y que constituye el núcleo de este trabajo de memoria. En eso realizamos:

- una selección de libros de literatura francesa del siglo XIX
- una "limpieza" de esos libros para obtener listas de frases
- una implementación de método de definición de distancia entre dos libros
- un análisis de grupos (o *clusters*)
- un análisis del sentido que podemos dar a esos grupos
- unas propuestas de mejora del modelo

1.4. Recursos disponibles en NLP

Hay muchos recursos disponibles en Internet sobre *Natural Language Processing*. Listamos los curso en linea masivos abiertos (*MOOC*), los autores relevantes, los artículos y las presentaciones más mencionadas.

1.4.1. *MOOC*

- el ramo de Stanford *CS 224N: NLP*. 14h horas de videos disponibles en Youtube.
- el ramo de Stanford *CS 224D: Deep Learning for NLP*. 15 conferencias disponibles en Youtube.

1.4.2. Personas de importancia en el dominio

- Yoshua Bengio. Director del *MILA (Montreal Institute for Learning Algorithms)* y co-director del proyecto *Learning in Machines & Brains* del *Canadian Institute for Advanced Research*. Ha escrito unos de los primeros *papers* sobre NLP.
- Dan Jurafsky. Profesor en Stanford del ramo CS 224N.
- Chris Manning. Segundo profesor en CS 224N.
- Richard Socher. Profesor de CS 224D.

- Tomas Mikolov: uno de los autores del algoritmo *word2vec*, y co-autor del paper más generalista sobre inteligencia artificial: *A Roadmap towards Machine Intelligence*. Trabajó por Google, y ahora por Facebook.

1.4.3. Artículos

- El primer modelo de representación vectorial de palabras: *A Neural Probabilistic Language Model*, Bengio et al. (2003)
- Los tres artículos que publicó el equipo de *Google* para presentar *word2vec*:
  1. *Efficient Estimation of Word Representations in Vector Space*, Mikolov, Chen, et al. (2013)
  2. *Distributed Representations of Words and Phrases and their Compositionality*, Mikolov, Sutskever, et al. (2013)
  3. *Linguistic regularities in continuous space word representations*, Mikolov, Yih, y Zweig (2013)
- Una generalización de *word2vec* para representación vectorial de documentos: *doc2vec. Distributed Representations of sentences and documents* Le y Mikolov (2014)

1.4.4. Presentaciones

- *Deep Learning for NLP (without Magic)* Socher (2013)
- *Distributed Representations* Hinton (2011)
- *Learning Representations of Text using Neural Networks* Mikolov (2013)

1.4.5. Otros

- The Stanford NLP Group
- NLTK: libreria en Python para NLP. Publicaron un libro también.

Capítulo 2

Preliminares

En esta sección, introducimos algunos métodos ocupados en NLP, empezando por las más básicas, siguiendo con el algoritmo *word2vec* y terminando por otros métodos, más específicos.

2.1. Herramientas básicas de NLP

Aquí proponemos una introducción a la caja de herramientas que se ocupan en NLP:

2.1.1. *Regular expression*

Al analizar un texto, el humano hace suposiciones que no son obvias para el computador. Entre otros, podemos citar la marca de forma plural o la mayuscula de principio de frase: ¿cómo explicar simplemente al computador que buscamos, sin distinción, uno de los siguientes?:

Lámpara, lámpara, Lámparas, lámparas

Para abordar ese problema, se creó la nomenclatura *RegExp* (o *Regular Expression*, Mitkov (2003)), de la cual damos algunos ejemplos:

Patrón	**Detecta**	**Ejemplo**
[A-Z]	Une letra en mayúscula	Hola Diego !
[a-z]	Una letra en minúscula	¿por qué?
[0-9]	Un número	Día 5: mucho viento todavía

Tabla 2.1: Reglas generales en *Regexp*

Detallemos el ejemplo anterior: para detectar las cuatro ocurrencias de *lámpara*, ocupamos el código *Regexp* siguiente:

```
[Ll]ámpara[s]*
```

donde el $[Ll]$ detecta el carácter L o l, y $[s]*$ detecta n veces la ocurrencia de la letra s, para $n \in [0, +\infty[$. Precisamos también que existen muchas reglas – no las vamos a detallar aquí –, que permiten detectar secuencias más elaboradas.

2.1.2. Tokenización de palabras

Para analizar un texto, una idea muy intuitiva es descomponerlo en las palabras que lo componen: se llama "tokenizar". Pero tan intuitiva que sea, esa idea no es tan simple en la práctica: puede pasar que una entidad esté compuesta por dos palabras, como la cuidad de *San Francisco* por ejemplo. Por lo tanto, tenemos que relacionar las reglas de reconocimiento de entidades con las de tokenización, para no obtener *"San"* y *"Francico"* como dos palabras distintas. Al revés, hay casos con los cuales sería más relevante separar la palabra. En Alemán, "*Lebensversicherungsgesellschaftsangestellter*" quiere decir "empleado de empresa de seguro de vida". Podemos imaginar que esa palabra no existe en el diccionario como tal, así que ganaríamos a separarla en sus partes.

2.1.3. Identificación de las palabras de menor sentido: *stopwords*

Podemos separar los elementos de una frase en dos grupos: en el primero serían los verbos, nombres y complementes, mientras que en el segundo guardaríamos las palabras de conexión, que no llevan mucho sentido. Algunos ejemplos son "el", "de", "por", "a", etc. Ese grupo tiene un nombre: las "palabras vacías" o *stopwords* (Rajaraman y Ullman, 2011).

Los idiomas más comunes tienen un diccionario que los contiene en la librerías de paquetes de NLP: es el caso por el Inglés, el Castellano, y el Francés por lo menos.

2.1.4. Normalización de palabras

De la misma manera, todavía con la idea de simplificar el acceso a un diccionario, nos conviene simplificar una palabra en función de su raíz. Por ejemplo, nos gustaría que las distintas conjugaciones de un verbo hagan referencia a su forma infinitiva:

$$\textit{tengo}, \textit{tienes}, \text{etc} \rightarrow \textit{tener}$$

Más bien, que el adverbio haga referencia al sustantivo, por ejemplo:

$$\textit{alternativamente} \rightarrow \textit{alternativo}$$

Eso se llama *lematización* (Muller et al., 2014): la acción de reducir una palabra a su lema.

Otra manera de proceder es ocupar el *stemming* (Beth, 1968), que se refiere a la búsqueda del radical de una palabra. Es el proceso que transforma por ejemplo "autómata", "automáti-

co" y "automatización" en la raíz "*automat*". Eso es un poco más grueso que la lematización, pero las librerías en *Python* de lematización se encuentran difícilmente en otros idiomas que el inglés.

2.1.5. Segmentación de frases

En una sección anterior mencionamos la tokenización. El paso siguiente es la segmentación de un texto en frases. En esa área, si los puntos de exclamación y de interrogación son bastante claros, no siempre es el caso de los puntos. Pueden servir en abreviaciones, como en "Sr." o "Sra.", o pueden ser un separador de parte entera y parte decimal en números.

En la práctica, el clasificador se hace usando el tipo de árbol siguiente.

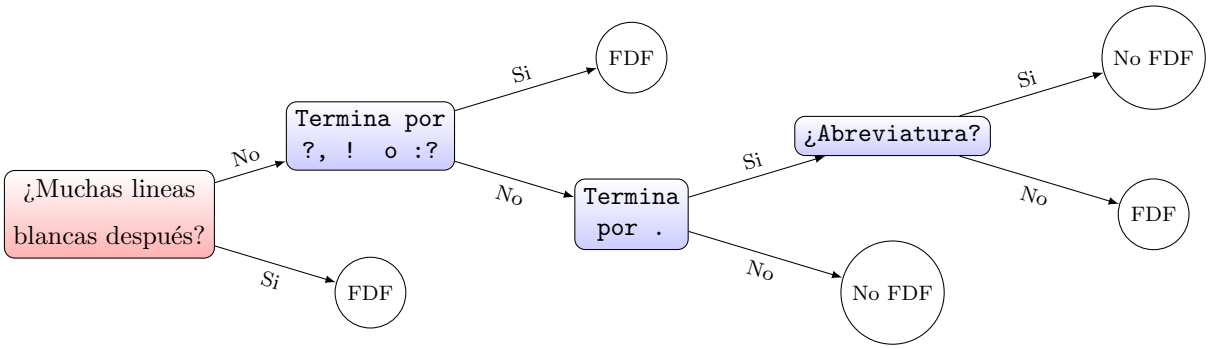

Fig. 2.1: Árbol para decidir si un punto constituye un fin de frase (FDF). Extraído y traducido de la primera lectura del ramo *CS 224N* de Stanford

2.1.6. Distancia de edición mínima

Con el fin de poder corregir los errores tipográficos en un texto, nos interesa definir una distancia entre dos cadenas de caracteres (o *strings*). Cuando se habla de distancia de edición, se refiere generalmente al número mínimo de operaciones necesarias para pasar de un *string* a otro, definiendo operación como uno de los siguientes: inserción, supresión, o substitución de carácter.

Consideremos el ejemplo siguiente:

O	C	U	R	R	E	N	C	I	A
|	|	|	|	|	|	|	|	|	|
R	E	F	E	R	E	N	C	I	A
s	*s*	*s*	*s*						

Tabla 2.2: Operaciones necesarias para pasar de la palabra "ocurrencia" a "referencia". Aquí notamos *s* por "substitución".

Según el modelo de distancia elegido, se puede definir varios costo por las substituciones: 1 si consideramos que es una operación como las otras, o 2 si lo vemos como una inserción seguida por una supresión. En el presente caso, tendríamos una distancia de 4 o 8 entre las dos palabras.

Mencionemos un último concepto en esta sección, que usamos implícitamente en el ejemplo anterior: el alineamiento. Otra forma de ver el problema de ir de una palabra a otra en el mínimo número de ediciones, es encontrar el mejor alineamiento. Veamos varios ejemplos de alineamientos entre dos palabras:

Palabra inicial	I	N	T	E	R	R	O	G	A	C	I	Ó	N
Alin. 1	I	N	T	E	N	C	I	Ó	N	*	*	*	*
					s	*s*	*s*	*s*	*s*	*d*	*d*	*d*	*d*
Alin. 2	*	*	*	*	I	N	T	E	N	C	I	Ó	N
	d	*d*	*d*	*d*	*s*	*s*	*s*	*s*	*s*				
Alin. 3	I	N	T	E	N	*	*	*	*	C	I	Ó	N
					s	*d*	*d*	*d*	*d*				

Tabla 2.3: Varios alineamientos entre dos palabras. Aquí notamos *s* por "substitución" y d por "supresión".

Aquí el mejor alineamiento es el tercero, pues permite pasar de una palabra a la otra en un número mínimo de operaciones. Ese concepto se usa en particular en Biología cuando se trata de comparar dos secuencias de ADN, como en el ejemplo de la figura 2.2:

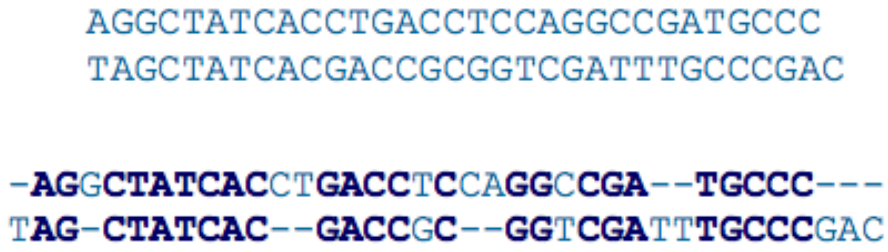

Fig. 2.2: Dos secuencias de ADN, cuya distancia está considerada desde el punto de vista de alineamiento. Extraído de la primera lectura del ramo *CS 224N* de Stanford

Mencionamos para concluir dos distancias conocidas: la de *Levenshtein* (Navarro, 2001) y la de *Jaro-Winkler* (Cohen et al., 2003). La primera es la más conocida: corresponde al número mínimo de ediciones (en el sentido de inserción, supresión o substitución) necesarias para pasar de una palabra a la otra. La segunda es una variación de la primera que da más importancia a las primeras letras de una palabra. Eso viene del hecho que los errores tipográficos son más raros en principio de palabras.

2.1.7. Estudio de grupos de palabras sucesivas: *n-grams*

Seguimos esa introducción a las herramientas de NLP por el análisis de n-gramas, o n palabras sucesivas (o *n-grams*, (Broder et al., 1997)). Eso tiene una importancia particular cuando uno quiere definir un modelo probabilístico del contenido de un corpus. Específicamente, definiendo la probabilidad de una palabra como su frecuencia, se hace la modelización siguiente:

$$\mathbb{P}(w_1 w_2 ... w_n) = \prod_{\mathrm{i}} \mathbb{P}(w_{\mathrm{i}} | w_1 w_2 ... w_{\mathrm{i}-1}) \tag{2.1}$$

donde $\mathbb{P}(w_j|w_{\mathrm{i}})$ se refiere a la probabilidad de que la palabra w_j siga w_{i}.

Desarrollemos un ejemplo:

$$\begin{aligned}\mathbb{P}(\text{“rumor de olas quebrándose”}) = \mathbb{P}(\text{“rumor”}) \times \mathbb{P}(\text{“de”}|\text{“rumor”}) \\ \times \mathbb{P}(\text{“olas”}|\text{“rumor de”}) \times \mathbb{P}(\text{“quebrándose”}|\text{“rumor de olas”})\end{aligned} \tag{2.2}$$

Ahora bien, al calcular $\mathbb{P}(\text{“quebrándose”}|\text{“rumor de olas”})$, sería bastante laborioso computar:

$$\frac{\text{cuenta(“rumor de olas quebrándose”)}}{\text{cuenta(“rumor de olas”)}} \tag{2.3}$$

por lo que se hace la hipótesis markoviana siguiente:

$$\mathbb{P}(\text{“quebrándose”}|\text{“rumor de olas”}) \approx \mathbb{P}(\text{“quebrándose”}|\text{“olas”}) \tag{2.4}$$

A continuación mostramos los modelos según su nivel de complejidad:

Modelo	**Hipótesis**
Unigram	$\mathbb{P}(w_1 w_2 ... w_n) \approx \prod_{\mathrm{i}} \mathbb{P}(w_{\mathrm{i}})$
Bigram	$\mathbb{P}(w_{\mathrm{i}} \mid w_1 w_2 ... w_{\mathrm{i}-1}) \approx \prod_{\mathrm{i}} \mathbb{P}(w_{\mathrm{i}} \mid w_{\mathrm{i}-1})$
...	...

Tabla 2.4: Ejemplos de modelos e hipótesis correspondientes

Este método se usa mucho en reconocimiento del habla (*speech recognition*), o traducción automática. Para terminar, precisemos lo siguiente: una debilidad de este método es su dependencia al conjunto de entrenamiento. En efecto, así hecho, el modelo asigna una probabilidad nula a palabras que nunca ha visto. Por lo tanto, se hace en general una interpolación: por ejemplo, se puede definir, para la n_x palabras $(w_{x_{\mathrm{i}}})_{\mathrm{i}=1..n}$ fueras del vocabulario:

$$\mathbb{P}(w_{x_{\mathrm{i}}}) = \frac{1}{|Vocabulario| + n_x} \tag{2.5}$$

2.1.8. El análisis de sentimiento con *Bag-of-Words*

Terminamos con un método que busca dar el tono general de un texto: en particular, determinar si hay un sentimiento que predomina, como la felicidad, la cólera, etc. Para hacer eso, tenemos que darnos una manera de representar las palabras.

La representación matemática de palabras más ingenua es la llamada *Bag-of-Words* (Ko, 2012), que corresponde a un *one-hot encoding* (Harris y Harris, 2007) – i.e. una representación binaria. Eso consiste en lo siguiente:

1. Reducir el corpus considerado a un *set* de *vocabulario* (en el sentido de que cada palabra aparezca una sola vez)

2. Ordenar ese conjunto alfabéticamente. Notamos N_{voc} su tamaño

3. Atribuir a la primera palabra el vector de tamaño N_{voc} siguiente: (1, 0, 0, ..., 0). La segunda palabra tendrá (0, 1, 0, ..., 0), etc

Obtenemos así una representación numérica de todas las palabras.

Una vez proveídos de esa representación, podemos ocupar un clasificador, para determinar si el documento considerado es positivo o negativo, o un método de *clustering* para agrupar los datos en las categorías clásicas de sentimiento: anticipación, alegría, confianza, miedo, sorpresa, tristeza, repugnancia, ira.

Podemos enumerar dos problemas con esa representación. El primero es la necesidad de acumular vectores de tamaño N_{voc} (que muchas veces puede ser superior a 10^6). El segundo es la ausencia de sentido reservada a la distancia: todas las palabras tienen entre ellas una distancia de $\sqrt{2}$, independientemente de la proximidad de sentido. Por ejemplo, hay la misma distancia entre *fuerte* y *potente* que entre *fuerte* y *París*. El algoritmo que presentamos en la siguiente sección intenta resolver esos dos problemas.

2.2. El algoritmo *word2vec*

Ahora presentamos un método llamado *word2vec* (Mikolov, Chen, et al., 2013). Ese método es un algoritmo creado por un equipo de *Google*, que toma como entrada un corpus de textos, y entrega vectores de palabras. El modelo, cuando abastecido con suficientes datos, produce una representación vectorial de palabras con propiedades notables: las palabras de sentido cercano se ubican próximas entre ellas, y eso da lugar al ejemplo famoso siguiente:

`vect(“Paris”) - vect(“Francia”) + vect(“Italia”) = vect(“Roma”)`

A continuación se muestran los resultados que se obtuvieron en la publicación original.

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

Tabla 2.5: Pares de palabras obtenidas en *word2vec* (extraído de Mikolov, Chen, et al. (2013), tabla 8)

Vemos que la asociación de palabras de la tabla 2.5 no es perfecta, pero la mayoría de los resultados son sorprendentemente relevantes. Tenemos que insistir sobre el hecho siguiente: esas relaciones no eran un objetivo del algoritmo. Nunca se buscó, en la manera de diseñar el algoritmo, obtener relaciones particulares entre palabras. Son únicamente un efecto secundario de la optimización realizada, la cual consiste simplemente en predecir, dada una palabra, la palabra anterior o siguiente.

Precisamos también que la proeza de *Google* no reside en la creación de ese algoritmo, que ya existía (Bengio et al., 2003), sino en la rapidez de su implementación: el entrenamiento de un corpus de 100 billones de palabras con un computador se puede hacer en un día, mientras que demoraba varias semanas con algoritmos anteriores.

En lo que sigue, describimos el modo de funcionamiento del algoritmo.

2.2.1. ¿Cómo funciona *word2vec*?

Queremos entonces un algoritmo que, dado un conjunto de textos, nos entregue una tabla de correspondencia entre cada palabra distinta y un vector que la represente en $\mathbb{R}^n$ – donde n es un parámetro que elegimos, habitualmente entre 100 y 500.

El modelo inicial

Por eso, vamos a usar un modelo llamado *Skip-Gram* (Huang et al., 1992), que es derivado del modelo del artículo *"A Neural Probabilistic Language Model"* (Bengio et al., 2003) y que busca predecir, dado una palabra, las palabras vecinas (que llamamos el "contexto"). En la práctica, dado un texto – de entrenamiento – compuesto por las palabras $w_1, w_2, ..., w_T$, esa

predicción equivale a encontrar los $(w_{t+\mathrm{i}})_{-c\leq \mathrm{i}\leq c,\mathrm{i}\neq 0}$ que maximizan la función siguiente:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{2.6}$$

donde c es el tamaño del contexto de entrenamiento (un parámetro del algoritmo).

Por otra parte, definimos la distribución condicional del modelo *Skip-Gram* $p(w_{t+j}|w_t)$ de la manera siguiente (llamada *softmax*):

$$p(w_{t+j}|w_t) = \frac{\exp(v_{w_{t+j}}^T v_{w_t})}{\sum_{w=1}^{W}\exp(v_w^T v_{w_t})} \tag{2.7}$$

donde W es el tamaño del vocabulario, y v_w es el vector asociado a la palabra w..

Entonces al principio inicializamos una matriz C de manera aleatoria, que contiene en cada columna las coordenadas de cada palabra:

$$C = \begin{pmatrix} \theta w_1^1 & \cdots & \theta w_T^1 \\ \vdots & & \vdots \\ \theta_{w_1^n} & \cdots & \theta w_T^n \end{pmatrix}$$

Ahora tenemos una función para minimizar, y $n \times T$ parámetros, lo que hace teóricamente factible el proceso. Sin embargo, la formula 2.7 es computacionalmente muy costosa, ya que depende de todos los elementos del conjunto de vocabulario, y que se tiene que computar por cada palabra, y por cada paso de la optimización. Más aún, no se necesita un modelo probabilístico completo para nuestra utilización.

Negative sampling

Entonces ahí interviene el truco del equipo de *Google*, llamado el *negative-sampling* (o *noise-contrastive estimation*) (Mikolov, Sutskever, et al. (2013)), que pretende simplificar la función de costo. En la práctica, remplazamos esta última por:

$$J_{NEG} = \log Q_\theta(D=1|w_t,h) + k \mathop{\mathbb{E}}_{\tilde{w}\sim P_{ruido}} [\log Q_\theta(D=0|\tilde{w},h)] \tag{2.8}$$

donde $Q_\theta(D=1|w_t,h)$ es la probabilidad (calculada con regresión logística) de ver la palabra w con el contexto h en el corpus D.

Describamos lo que pasa aquí: en vez de calcular la probabilidad de un contexto dada una palabra (conformemente al modelo *Skip-Gram*) por cada contexto en el corpus, y elegir el más probable de todos, entrenamos un clasificador para reconocer si el ensamblaje de la palabra w con el contexto h es plausible o no. Entrenamos el clasificador con un par (w,h) verdadero y k pares falsos, extraídos de una distribución de ruido P_{ruido}.

Desarrollemos un ejemplo para ilustrar esa idea. Tenemos por ejemplo como corpus la frase:

the cat is sat on the mat

y decidimos para simplificar que el contexto es de 1 (el contexto es un parámetro del algoritmo). Con la nomenclatura (contexto, palabra), tenemos el conjunto de datos siguiente:

([the, is], cat), ([cat,sat], is), ([is,on],sat), ...

La tarea del modelo *Skip-Gram* es de predecir "*the*" e "*is*" dado "*cat*", "*sat*" y "*sat*" dado "*is*", etc, así que el conjunto queda, con la nomenclatura (entrada, salida):

(cat, the), (cat, is), (is, cat), (is, sat), ...

Ahora, supongamos que estemos al paso t de la optimización, y que queramos predecir "*the*" dado "*cat*". Seleccionamos k ejemplos de la distribución de ruido (con $k = 2$), y la función de costo queda:

$$\begin{aligned} J_{NEG}^{(t)} = \log Q_\theta(D = 1|\text{the,cat}) + \log Q_\theta(D = 0|\text{nuclear,cat}) \\ + \log Q_\theta(D = 0|\text{ecoresponsability,cat}) \end{aligned} \tag{2.9}$$

Ahora, sola falta aplicar un método de optimización, por ejemplo el *gradient descent*, y por eso calcular $\frac{\partial}{\partial\theta} J_{NEG}$.

Ahora, hagámonos la pregunta siguiente: por qué era necesario usar pares malos extraídos de un conjunto de ruido? ¿Podríamos haber definido J_{NEG} como sigue?:

$$J'_{NEG} = \log Q_\theta(D = 1|w_t, h) \tag{2.10}$$

En ese caso, definiendo $Q_\theta(D = 1|w_t, h)$ como un *softmax*, es decir:

$$Q_\theta(D = 1|w_t, h) = \frac{1}{1 + \mathrm{e}^{-v_{w_t} \cdot v_h}} \tag{2.11}$$

podríamos maximizar de manera obvia J'_{NEG} con $v_{w_t} = v_h$ (con $v_{w_t} \cdot v_h$ suficientemente grande). Por eso, necesitamos un mecanismo que lo nos impida, y una solución es generar pares aleatoria de una distribución P_{ruido} (independiente del corpus elegido), asumiendo que los pares generados son incorrectos. Por otra parte, precisamos que ese método se base en resultados matemáticos de convergencia. En particular:

$$\lim_{k\to\infty} \frac{\partial}{\partial\theta} J_{NEG}(k) = \frac{\partial}{\partial\theta} J_{\text{softmax}} \tag{2.12}$$

Para concluir esas explicaciones sobre *negative-sampling*, añadimos que la complejidad, que estaba en $O(|V|)$ antes, está ahora en $O(k)$, donde k típicamente está entre 2 y 20, según el tamaño del corpus.

2.2.2. Parámetros del algoritmo

Tenemos varios parámetros que podemos ajustar en el algoritmo. Primero daremos detalles sobre los más importantes, y después se explicará como ajustarlos

Parámetros principales

- **Arquitectura**: puede ser *continuous bag of words (CBOW)* o *skip-gram*. Con *CBOW*, se busca predecir una palabra dado un contexto (por ejemplo: *the cat is sat on the* $\rightarrow$ *mat*), y con *skip-gram*, queremos predecir un contexto dado una palabra.

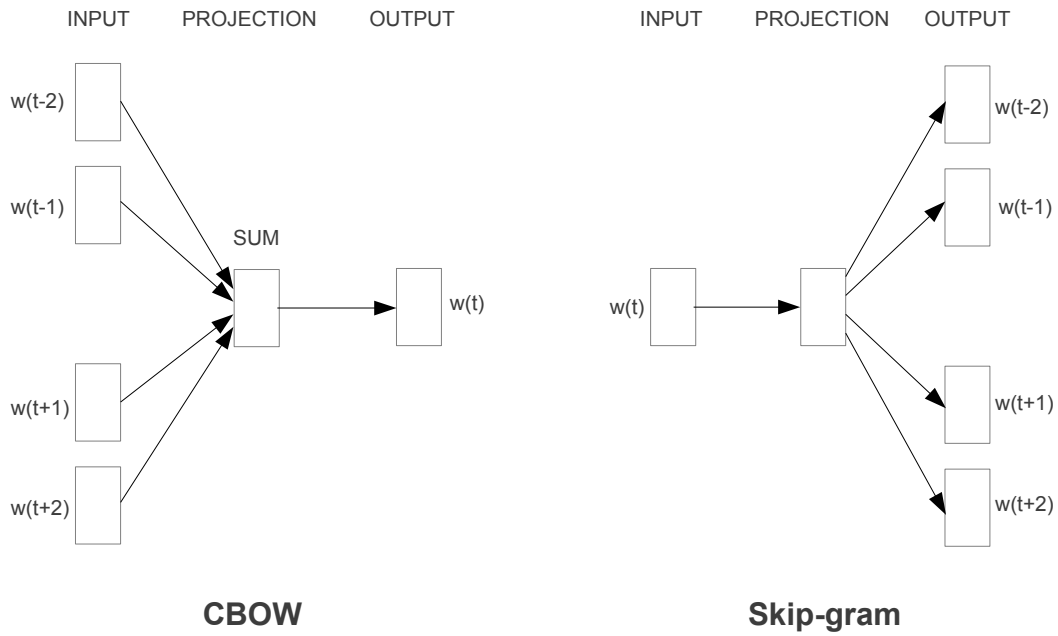

Fig. 2.3: Modelos *CBOW* y *Skip-Gram* (extraído de Mikolov, Chen, et al. (2013), figura 1)

- **Algoritmo de entrenamiento**: *negative sampling* o *hierarchical softmax*.
  El *hierarchical softmax* es un método alternativo al que usamos. En pocas palabras, la idea es de ordenar las palabras por orden de frecuencia, y organizarlas en un árbol, con las más frecuentes en la raíz y las menos en las ramas. Eso permite obtener un *softmax* que se calcule en $O(\log(|W|))$ en vez de $O(|W|)$. En práctica, se obtuvieron mejores resultados con *negative sampling*.
- **Downsampling of frequent words**: no queremos contaminar nuestro modelo por palabras privadas de sentido. Esas suelen ser las palabras las más frecuentes, como *el, la, de, etc.*. Por eso, con ese parámetros, fijamos la cantidad de palabras más frecuentes que no consideramos. La documentación de *Google* recomienda valores entre .00001 y .001.
- **Dimensión de los vectores de palabras**: se recomienda usar valores entre 1 y 500, según el tamaño del corpus
- **Contexto**: número de palabras que tomamos en cuenta alrededor de la palabra de interés
- **Cuenta mínima de palabras**: número de veces que debe aparecer una palabra para estar tomada en cuenta por el algoritmo. Eso permite evitar los errores ortográficos.

¿Cómo ajustarlos?

Aquí nos hacemos la pregunta de cómo encontrar los mejores parámetros en un modelo *word2vec*. Esa se inscribe en un cuadro más largo: como elegir los parámetros de un modelo en caso de aprendizaje no supervisado? En el caso del idioma inglés, *Google* ha publicado un conjunto de test que usa la propiedad famosa de *word2vec*: "A es para B lo que C es para D", donde el algoritmo está evaluado sobre su capacidad para encontrar D.

A continuación se muestran el tipo de pares que se propuso en el conjunto de test proveído por Mikolov, Yih, y Zweig (2013):

Columna 1	**Columna 2**	**Columna 3**	**Columna 4**
Cambodia	Cambodian	Belarus	Belorussian
Cambodia	Cambodian	Brazil	Brazilian
Chile	Chilean	China	Chinese
Chile	Chilean	Colombia	Colombian
Chile	Chilean	Croatia	Croatian
Chile	Chilean	Denmark	Danish

Tabla 2.6: Extracto del documento de test. Pares de vocabulario: nacionalidad y gentilicio

Columna 1	**Columna 2**	**Columna 3**	**Columna 4**
going	went	dancing	danced
going	went	decreasing	decreased
going	went	enhancing	enhanced
going	went	falling	fell
going	went	feeding	fed
going	went	flying	flew

Tabla 2.7: Extracto del documento de test. Pares de verbos: gerundio y pretérito

En ese documento, además de preguntas sobre la nacionalidad y los verbos, se encuentran también pares sobre la familia (por ejemplo: *[man-woman]* vs *[brother-sister]*), así que sobre reglas gramáticas (singular *vs* plural, etc). Al final, podemos obtener un puntaje de las pares exitosamente relacionadas en cada sección, y nos permite evaluar la calidad de nuestro modelo. Sin embargo, podemos observar algunos límites a este método:

1. existe solamente en inglés

2. está muy anclado en la época actual, tan al nivel del vocabulario elegido que del tipo de preguntas: un corpus de textos de Shakespeare, tan completo que sea, no tendrá muy buenos resultados sobre las pares que involucran Chile por ejemplo.

3. En un sentido más general, los vectores de palabras se hacen para un propósito definido, y es según ese propósito que hay que evaluarlos

2.3. De la palabra al documento

Hasta aquí, el enfoque se ha hecho al nivel de la palabra. Tomemos un punto de vista más global e interesémonos al nivel superior, el del párrafo, o del documento entero.

2.3.1. Otras posibilidades: más allá de *word2vec*

Un de los limites de *word2vec* es que no toma en cuenta el orden de las palabras. Para mejorar eso, existe el algoritmo *doc2vec* (Le y Mikolov (2014)), que crea *paragraph vectors*. Sin embargo, el algoritmo es bastante reciente y le falta documentación todavía.

2.3.2. *Latent Semantic Analysis*

El análisis latente semántico es una técnica que pretende establecer relaciones entre un conjunto de documentos y las palabras que contienen, con el fin de reducir un documento a algunas "palabras claves". Un de los objetos más usados en esa técnica es la matriz *Time frecuency - Inverse document frecuency* (o *TF-IDF*, Rajaraman y Ullman (2011)), que presentamos ahora.

Esa matriz pretende analizar el contenido de los documentos con respecto al conjunto entero de los documentos. Eso se base en el hecho que una palabra poco común tiene más probabilidad de ocupar un rol importante en una frase. En más detalles, el peso dado a una palabra aumenta proporcionalmente al número de veces que ocurre en una pregunta, y baja proporcionalmente al número de pregunta en que aparece.

Por ejemplo, en la pregunta siguiente (considerada en un conjunto de preguntas más amplio):

"¿Cuáles son las mejores actividades en Japón?"

la palabra *Japón* es más característica de la pregunta que la palabra *mejores*, que se puede encontrar de manera arbitraria, y por lo tanto la primera tendrá un peso más grande. De manera más calculatoria, definimos el peso de una palabra i en una pregunta j como sigue:

$$\mathrm{tfidf}_{\mathrm{i},j} = \mathrm{tf}_{\mathrm{i},j} \cdot \mathrm{idf}_{\mathrm{i}} \tag{2.13}$$

donde:

- $\mathrm{tf}_{\mathrm{i},j}$ es la frecuencia de la palabra i en la pregunta j
- $\mathrm{idf}_{\mathrm{i}} = \log \frac{|D|}{|\{\mathrm{d}_j : t_{\mathrm{i}} \in \mathrm{d}_j\}|}$

(con $|D|$ el número total de preguntas en el conjunto, y $|\mathrm{d}_j : t_{\mathrm{i}} \in \mathrm{d}_j|$ el número de preguntas donde aparece la palabra t_{i})

2.3.3. *Latent Dirichlet Allocation*

Para terminar, presentamos el resultado de un artículo (Blei et al., 2002) que pretende clasificar documentos por tema. Ese técnica se llama *Latent Dirichlet Allocation*, y es derivada de *LSA*. Demos un ejemplo ilustrativo.

Supongamos que tengamos las frases siguientes:

1. Como postre, voy a pedir la *crème brulée*
2. Me gustan el chocolate y los plátanos
3. Que tierno ese perro !
4. Yo tengo dos gatos.
5. El mono se comió toda mi comida !

Ejecutando ese algoritmo con el objetivo de tener dos temas, lo que podemos obtener es lo siguiente:

- Frases 1 y 2: 100 % tema A
- Frases 3 y 4: 100 % tema B
- Frase 5: 65 % tema A, 35 % tema B
- Tema A: 15 % "postre", 15 % *"crème brulée"*, 13 % "chocolate", 13 % "'platanos"
- Tema B: 20 % "perro", 20 % "gatos", 15 % "mono", ...

Con esas informaciones, se puede deducir fácilmente que el tema A está relacionado con la comida, y el tema B con los animales. Eso es muy útil para clasificar documentos en un corpus, tratando de ver en que medida son parecidos.

Para terminar, mencionemos otra aplicación derivada, interesante en visión artificial: clasificación de imágenes. A continuación se muestra un extracto del artículo *"Describing visual scenes"* (Sudderth et al. (2013))

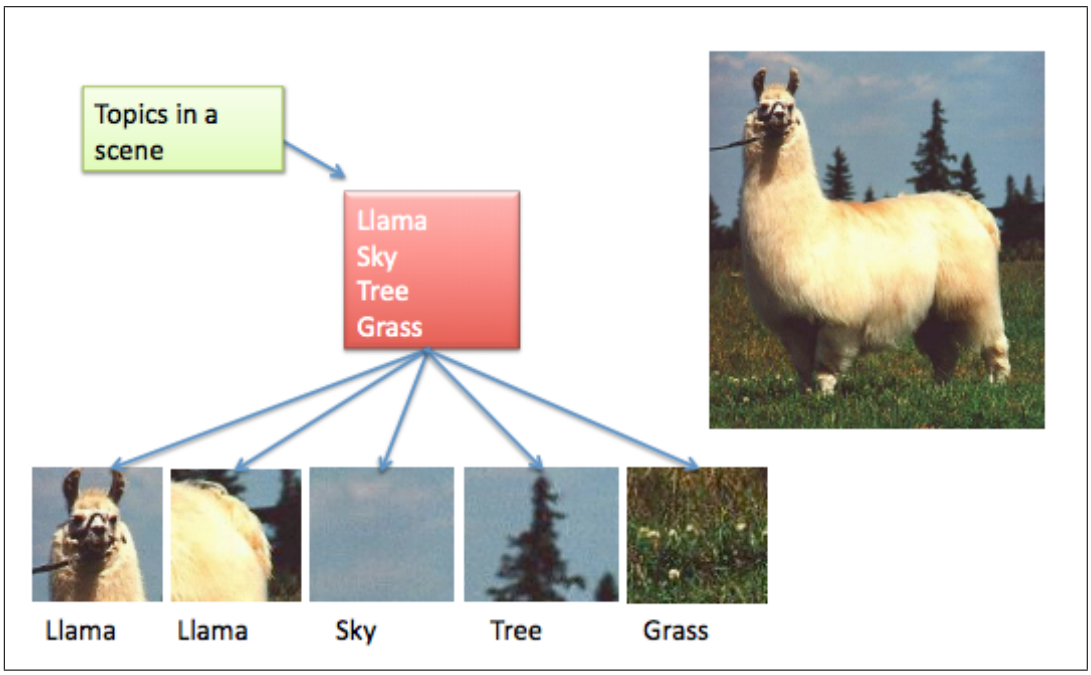

Fig. 2.4: Clasificación de subpartes de imágenes. Extracto del *paper "Describing visual scenes"*

2.4. Otros métodos usados en esta memoria

2.4.1. Random Forests

Entre los numerosos clasificadores que existen en aprendizaje de máquina, elegimos – por simplicidad de uso – el llamado "bosques aleatorios" (o *random forests*, Breiman (2001)). Su funcionamiento consiste en tomar n muestras, constituidas de una fracción de los datos y de los atributos. Luego, se entrenan n árboles de clasificación(Rokach y Maimon, 2008), y se promedian los resultados de todos.

2.4.2. Cross-Validation

En aprendizaje de máquina supervisado, cuando uno arma un modelo de clasificación, necesita evaluar sus resultados. Para poder medirlos, eso se debe hacer en el conjunto de entrenamiento, pues está etiquetado. La validación cruzada (o *cross-validation*, Geisser (1993)) se encarga de realizar esa tarea de manera insesgada, procediendo como sigue:

1. Separar aleatoriamente el conjunto en n_{folds} partes $(C(\text{i}))_{\text{i}=1..n_{\text{folds}}}$
2. Para $j = 1..n_{\text{folds}}$, entrenar el clasificador sobre los conjuntos $(C(\text{i}))_{\text{i}\neq j}$ y evaluarlo sobre $C(j)$ con la medida elegida
3. Promediar el resultado sobre los n_{folds}

Usaremos este método en el siguiente capitulo.

2.4.3. *t-distributed Stochastic Neighbor Embedding*

t-SNE es una técnica clásica de reducción de dimensionalidad que funciona bien con espacios de alta dimensión. Funciona de la manera siguiente :

1. se construye una distribución de probabilidad, que asigna alta probabilidad a los pares de elementos cercanos en el espacio original, y baja probabilidad a los lejanos (con distancia euclidiana).

2. se construye otra distribución de probabilidad en el espacio de dimensión reducida, y trata de minimizar la divergencia Kullback-Leibler (Kullback y Leibler, 1951) entre esas dos distribuciones.

Usaremos este método para representar los vectores de palabras en el siguiente capitulo.

2.4.4. *Clustering*

El análisis de grupos consiste en agrupar elementos de un conjunto de datos según propiedades que los caracterizan. Existen muchos métodos de *clustering*: en nuestro caso, nos vamos a enfocar sobre *hierarchical clustering* (Rokach y Maimon, 2005) que tiene la particularidad de tomar como entrada no las coordenadas de puntos sino una matriz de distancia.

La idea de *hierarchical clustering* es de partir con un grupo por elemento, y ampliar el tamaño de esos grupos, hasta alcanzar el número de grupos deseado.

2.5. Implementaciones en Python

Mencionemos el hecho siguiente, sobre la filosofía en *Python* con conjuntos de datos de tamaño importante: mientras parecería natural iterar sobre las filas de nuestras tablas de datos, hay que cuidar con el hecho que ese lenguaje es muy poco eficiente con bucles *for*. Eso viene del hecho que *Python* es un lenguaje interpretado, y que necesita, para cada operación, hacer operaciones adicionales, como verificar el tipo de los variables.

Por lo tanto, les preferiremos el método `.apply` de la librería *pandas* (ver pandas (2017)), que usa código optimizado en *C* que le asegura más velocidad para ese tipo de tareas. En más detalles, una tarea que involucra un bucle *for* va a ser delegada hacia código compilado – como *C* o *Fortran* – donde se va a hacer más rápidamente.

Capítulo 3

Ejemplo de uso de *word2vec*: detección de preguntas duplicadas en *Quora*

Quora es una plataforma parecida a un foro, con un sistema de *upvotes*. Personas hacen preguntas sobre temas diversos, y las respuestas aparecen en el orden del número de votos. Por su simplicidad de utilización, sumada a un crecimiento rápido, *Quora* ha atraído como usuarios muchos expertos en su dominio, así que personalidades famosas: Barack Obama, Mark Zuckerberg, Noam Chomsky, o Hans Zimmer por ejemplo.

Eso ha estimulado el desarrollo del sitio, agregando mucho valor a la calidad de las respuestas. Sin embargo, ahora con más de 100 millones de visitantes cada mes, el sitio tiene el problema donde muchas preguntas existen en varias ocurrencias. Eso tiene como consecuencia que los lectores toman más tiempo para encontrar la información que buscan, y los escritores contestan varias veces la misma pregunta.

Por eso, publicaron un extracto de sus datos para el sitio de competencias en aprendizaje de máquinas *Kaggle.com*, contando con la inventiva de los integrantes para diseñar un algoritmo capaz de reconocer las preguntas duplicadas.

Los datos se presentan como sigue:

id	qid1	qid2	question1	question2	is_duplicate
0	1	2	What is the step by step guide to invest in share market in india?	What is the step by step guide to invest in share market?	0
1	3	4	What is the story of Kohinoor (Koh-i-Noor) Diamond?	What would happen if the Indian government stole the Kohinoor (Koh-i-Noor) diamond back?	0
...	...	...	...	...	...
7	15	16	How can I be a good geologist?	What should I do to be a great geologist?	1

Tabla 3.1: Organización de los datos proveídos por *Quora*. Dos pares son distintos, un par es duplicado

3.1. Presentación de la competencia

El objetivo de la competencia es etiquetar los pares del conjunto de test con probabilidades de ser preguntas duplicadas. Se evalúa después el *logarithmic loss*, es decir:

$$-\frac{1}{N}\sum_{\mathrm{i}=1}^{N}[y_{\mathrm{i}}\log(p_{\mathrm{i}})+(1-y_{\mathrm{i}})\log(1-p_{\mathrm{i}})] \tag{3.1}$$

donde y_{i} es la etiqueta del par (notamos 1 los duplicados), p_{i} es la probabilidad que el par si sea un duplicado.

Intuitivamente, podemos comentar que esa medida penaliza los pares mal etiquetados relativamente a la certeza (en términos de probabilidad) con la cual fueron etiquetados.

Precisemos por terminar que, según la organizadores, dos preguntas se consideran duplicadas cuando se pueden contestar por la misma respuesta.

3.2. Exploración de los datos

Los datos disponibles para la competencia están divididos entre un conjunto de entrenamiento de 400.000 pares de preguntas etiquetadas, y 2.350.000 pares en el conjunto de test.

3.2.1. Repartición de las clases

Notemos que los dos conjuntos son desbalanceados: hay 36.9 % de duplicados en el de entrenamiento, y 16.7 % en el de test. Esa última información se deduce del hecho siguiente:

si asignamos a cada par en el conjunto de test la probabilidad 36.9 % (probabilidad de ser un par duplicado), y hacemos una sumisión en la plataforma de *Kaggle*, obtenemos un *logloss* de 0.55. Con eso, podemos simplificar la última formula:

$$LL = -\tau_P \log p - (1 - \tau_P) \log (1 - p) \tag{3.2}$$

Luego:

$$\tau_P = \frac{\log (1 - p) + LL}{\log \frac{1-p}{p}} \tag{3.3}$$

Calculando con $p = 0{,}369$ y $LL = 0{,}55$, obtenemos $\tau_P = 16{,}7\,\%$. Esa diferencia entre los dos conjuntos tiene que ver con el hecho de que hay muchas preguntas en el conjunto de test que son generadas automáticamente, para desanimar el etiquetado "a mano".

La tasa de pares positivos nos va a guiar en la construcción del clasificador: tendremos que abastecerle con extractos del conjunto de datos adecuadamente balanceados.

3.2.2. Análisis preliminar de las preguntas

Exploramos ahora los datos que tenemos. A continuación se muestra el histograma de la cuenta de caracteres en preguntas:

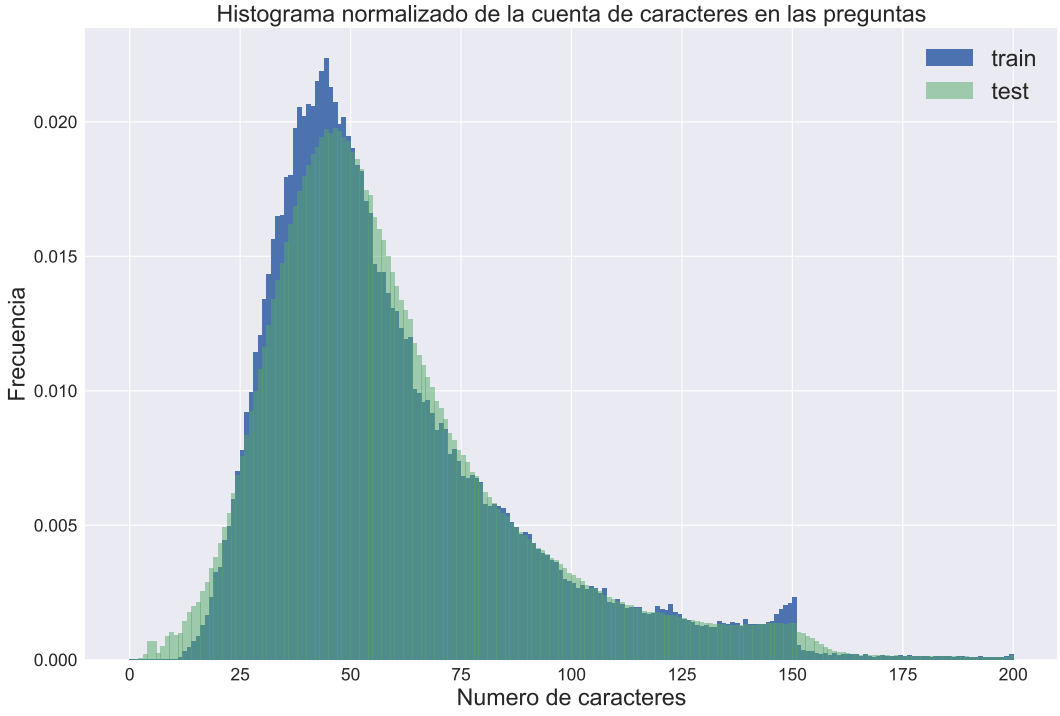

Fig. 3.1: Número de caracteres en preguntas *vs* frecuencia

Podemos ver que la repartición es muy similar en los dos conjuntos. La mayoría de las preguntas tiene entre 30 y 65 caracteres.

Miremos ahora el histograma de la cuenta de palabras:

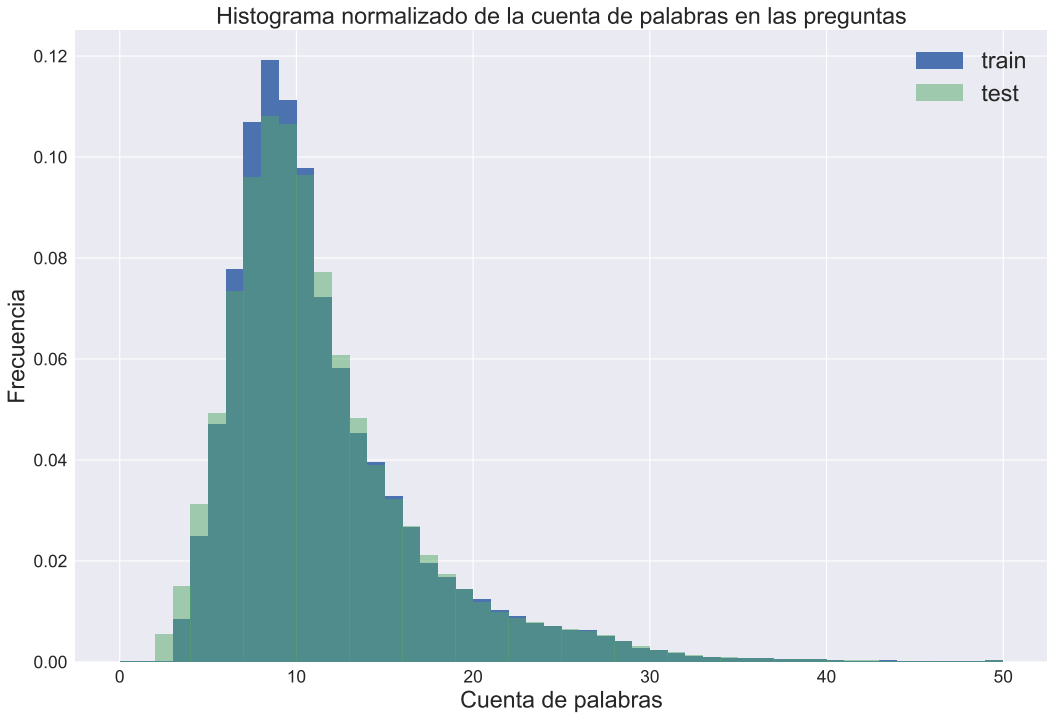

Fig. 3.2: Número de palabras en preguntas *vs* frecuencia

Como anteriormente, la repartición es muy similar en los dos conjuntos. La mayoría de las preguntas tiene entre 5 y 15 palabras.

Para terminar con la exploración estadística, demos en la figura 3.3 un ejemplo más visual de lo que contiene nuestro conjunto de entrenamiento: gracias al paquete *WordCloud*, podemos representar la frecuencia de las expresiones según el tamaño de la fuente, i.e. las palabras más grandes son las que aparecen más (en el conjunto de entrenamiento). Precisemos que los *stopwords* no están representados.

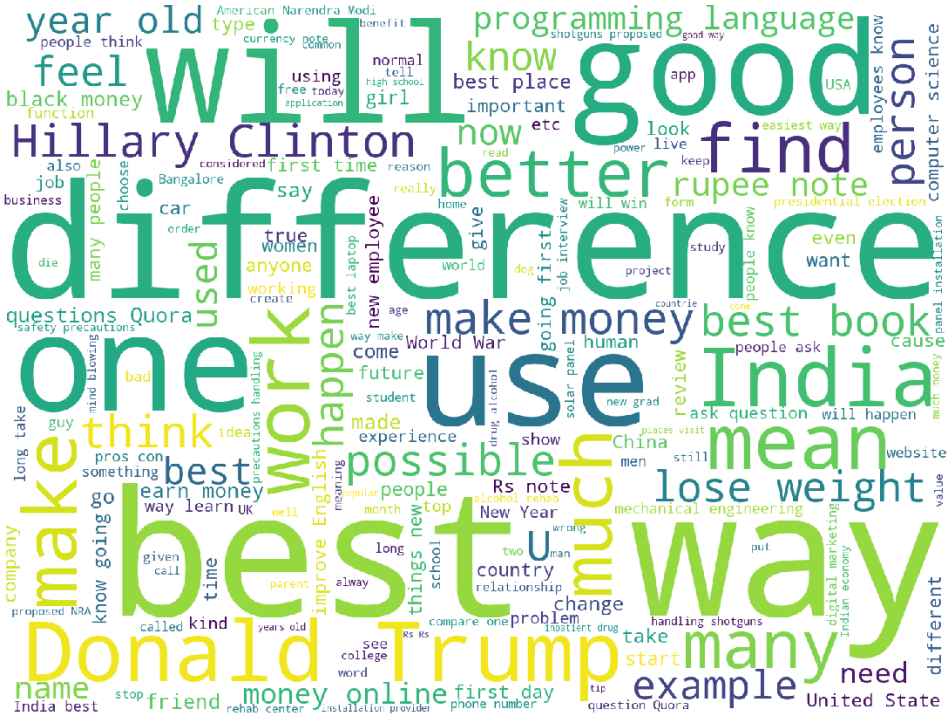

Fig. 3.3: Representación de los términos más frecuentes en las palabras, vía el paquete *WordCloud*

Ese gráfico da cuenta del hecho que hay una gran parte de las preguntas en *Quora* que empieza por *'What is the best way ...?"* o por *"What is the difference between ...?"*. Notemos que *"What"* no aparece por ser un *stopword*.

Ahora bien, si esta última representación es eficiente en términos de frecuencia de utilización de grupos de palabras, no se preocupa mucho del sentido de esos grupos. Antes de seguir con el planteamiento del problema, veamos si podemos abordar ese problema. mediante una representación gráfica. Para hacer eso, vamos a usar *t-SNE* (van der Maaten y Hinton, 2008), un método de reducción de dimensionalidad.

Primero limpiemos los datos, y reduzcamos las preguntas a sus palabras importantes, como se ve en lo que sigue:

Par inicial	Why do some Americans leave the US and go to live in Israel?	Which scenario would make an average American Jew to live in Israel?
Par modificado	[americans, leave, us, go, live, israel]	[scenario, would, make, average, american, jew, live, israel]

Tabla 3.2: Ejemplo de par de preguntas extraído del conjunto de entrenamiento, antes y después de transformarlo para *word2vec*

Con esta modificación, obtenemos un corpus, que consiste en una lista de preguntas modificadas, es decir una lista de lista de palabras:

```
corpus = [[americans, leave, us,go, live, israel], [scenario, would,
          make,average, american,jew, live, israel], ... ]
```

Ahora podemos entrenar el algoritmo:

```
model = word2vec.Word2Vec(corpus, size=100, window=10, min_count=250,
                          workers=4)
```

Aquí tenemos:

- una dimensionalidad de 100
- una ventana de contexto de 10 palabras
- una ocurrencia mínima de 250 veces por palabra para estar tomada en cuenta
- 4 *cores* del procesador activos

Finalmente, podemos representar gráficamente los resultados mediante *t-SNE*.

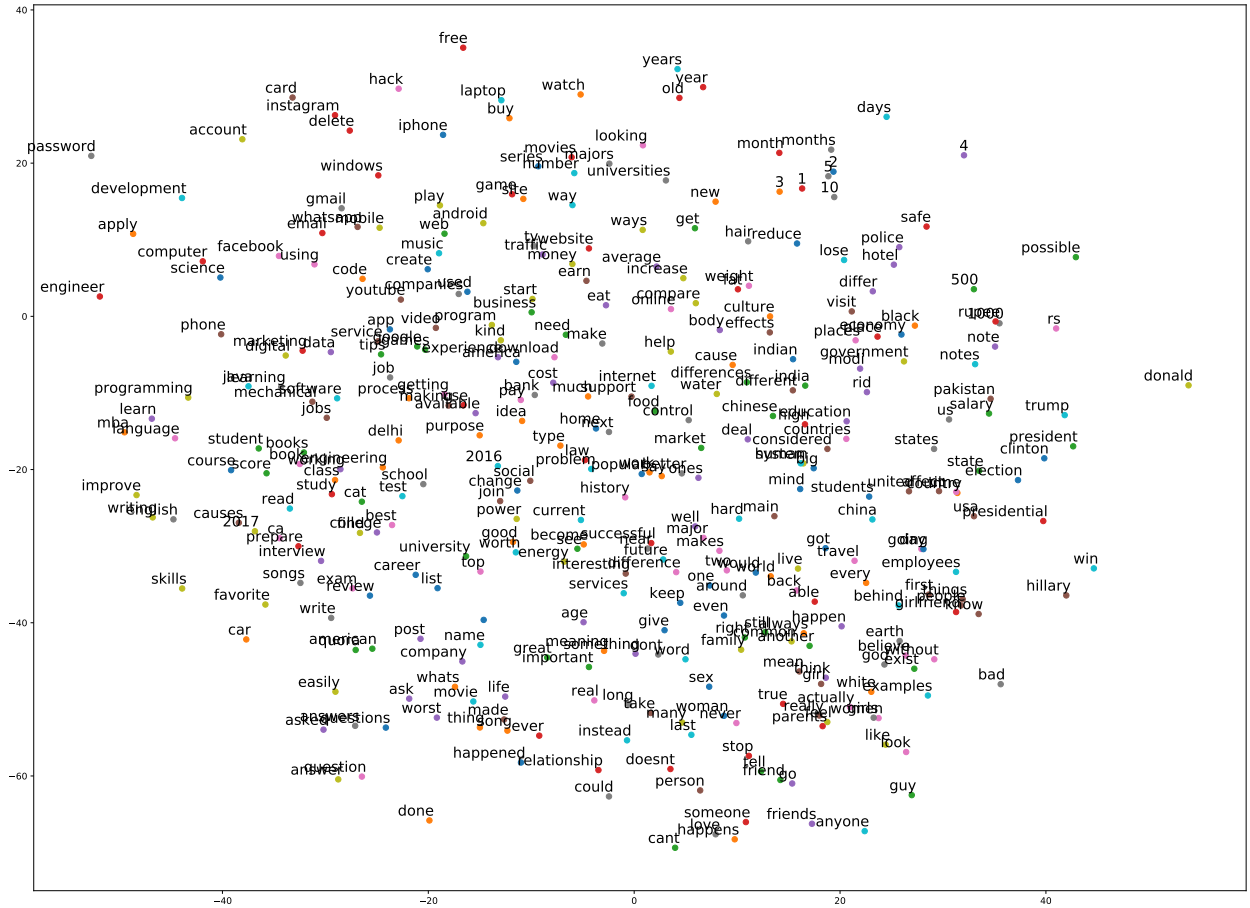

Fig. 3.4: Representación mediante *t-SNE* de los vectores de palabras

Eso nos da una idea de la palabras más comunes en nuestro extracto del conjunto de entrenamiento. Para terminar ese estudio, mostramos con más precisión las relaciones entre algunas palabras "claves".

quora	**sim.**	**facebook**	**sim.**	**government**	**sim.**
answers	0.94	instagram	0.94	modi	0.88
questions	0.92	delete	0.93	economy	0.88
ask	0.92	password	0.92	indian	0.87
answer	0.90	whatsapp	0.92	current	0.87
asked	0.89	gmail	0.91	affect	0.85
easily	0.88	account	0.88	500	0.82
question	0.87	hack	0.87	1000	0.80
post	0.79	email	0.85	notes	0.79
google	0.71	list	0.83	rupee	0.79
instead	0.70	someone	0.80	note	0.76

Tabla 3.3: Palabras más parecidas a *"Quora"*, *"Facebook"*, y *"government"*

La tabla 3.3 da cuenta de algunas de las asociaciones de palabras que se suelen hacer en

Quora.

Ahora, con un mejor entendimiento de la composición de nuestros conjuntos de datos, podemos seguir avanzando con el análisis en si mismo.

3.3. El rol de *word2vec*

El algoritmo de *Google* va a jugar un rol determinante en este problema: nos va a permitir "matematizar" el problema, transformando los datos en vectores, con los cuales podremos abastecer nuestro clasificador.

Una pregunta que uno se puede hacer consiste en la manera de entrenarlo. Para hacer eso, tenemos varias opciones:

- entrenar un modelo con la lista de preguntas en Quora. Eso tiene la desventaja de un tipo de *overfitting*
- usar un modelo pre-entrenado:
  - el primer millón de palabras de *Wikipedia* (i.e. los primeros artículos en el orden alfabético hasta alcanzar un millón de palabras).
  - el conjunto dispuesto por el equipo de *word2vec* que consiste en 100 billón de palabras extraídas de *Google News*. Eso tiene la ventaja de corresponder al vocabulario usado en Quora de manera más adecuada, pero el modelo es muy pesado (3 Gigas).

Por un tema de rapidez, lo hacemos con el conjunto de *Wikipedia*, que nos da resulta coherentes sobre palabras comunes:

man	**sim.**	**woman**	**sim.**	**house**	**sim.**
woman	0.73	child	0.80	commons	0.71
girl	0.67	girl	0.77	lords	0.64
creature	0.62	man	0.73	usher	0.58
boy	0.59	lover	0.69	rooming	0.57
gentleman	0.58	mother	0.69	parliament	0.56
evil	0.58	herself	0.67	houses	0.56
stranger	0.57	husband	0.67	palace	0.56
person	0.56	lady	0.66	representatives	0.55
demon	0.55	person	0.66	manor	0.55
beast	0.55	baby	0.65	chamber	0.54

Tabla 3.4: Palabras más parecidas a *"man"*, *"woman"*, y *"house"* en el modelo *Wikipedia*

Las asociaciones de palabras de la tabla 3.4 parecen relevantes. Notamos que la palabra *"house"* está tomada aquí en su sentido político.

3.4. Primer planteamiento del problema: un vector promedio por pregunta

Para empezar, una manera ingenua de proceder consiste en lo siguiente:

1. tomar los vectores asociados a las palabras de una pregunta (que no son *stopwords*)

2. promediarlos y observar la distancia (producto punto, norma de la diferencia, etc) entre ese y la otra pregunta del par

3. clasificar según esa distancia, mediante un algoritmo de *Random Forest* (Breiman (2001)) por ejemplo.

Ese método es poco sutil, en el sentido de que perdimos mucha información al promediar todos los vectores de palabras que componen una pregunta. Nos da resultados buenos en casos simples como los siguientes (notemos que consideramos no la distancia sino la similitud (1 − producto punto)):

question1	**question2**	**is_duplicate**	**sim**
Is using WiFi during a thunderstorm dangerous?	Why do jellyfish end up on the beach during a thunderstorm?	0	-0.099
Is there American flag on lunar surface which Apollo 16 astronots carried with them?	Why is Edward Said's Orientalism highly appraised?	0	-0.093
How can you replace a power button on a iPhone 4S?	Why are Arab businessmen so rich? Does it have to do with the Old Testament?	0	-0.064

Tabla 3.5: Extracto de pares ingenuos

Aquí vemos que la preguntas en cada par son bien distintas: ni siquiera tienen una palabra en común. De hecho, nuestra medida les da una similitud muy baja.

Veamos ahora unos casos mas sutiles:

question1	question2	is_duplicate	sim
How do I integrate x^3/(x+1) ^2?	How do I Integrate: 1/ (1-x^3)?	0	1.0
What are the best English songs of 2016?	What are the best English songs of 2015?	0	1.0
How is the word “gloat” used in a sentence?	How is the word ’temerity’ used in a sentence?	0	1.0
From where can I download software for free?	Which antivirus software can I download for free?	0	1.0

Tabla 3.6: Pares más sutiles: se diferencian por pocas palabras

Podemos ver que nuestra medida de similitud atribuye un score muy alto a esos pares, mientras que no son preguntas duplicadas. Eso viene del hecho que las preguntas en los pares tienen muchas palabras en común. Para mejorar eso, vamos a introducir otro *feature* en la siguiente sección.

3.4.1. De la necesidad ponderar las palabras por importancia: *TF-IDF*

Una vez sacados los *stopwords*, una pregunta queda con palabras que tienen varios niveles de importancia. Sin embargo, al promediar los vectores de palabra, no se considera esa variación. Para tomar en cuenta ese hecho, usamos el método llamado $TF-IDF$ que presentamos en la sección introductiva.

Ahora, una idea más elaborada que la anterior sería normalizar los vectores obtenidos en *word2vec*, y, en cada pregunta, darle a cada uno su peso $TF-IDF$. Miremos como mejoran los resultados con ese método, observando pares que tienen similitud alta pero que no son duplicados:

question1	question2	sim	sim pond
What are some of the best kept secrets of the rich?	What is the best kept secret?	0.80	0.70
What can I do to make myself more attractive to women?	What can I do to make myself more attractive?	0.80	0.77
Which is the best Android smartphone under 20k in India in today's date (2017)?	Which is the best android smartphone under 20k in India in today's date(2016)?	0.86	0.73
What is the best antivirus protected WhatsApp?	What is the best antivirus protection?	0.83	0.60
Where are the best honeymoon destinations in India in June?	Which is the best honeymoon destination for couples in India?	0.80	0.60

Tabla 3.7: Similitud y similitud ponderada usando *TF-IDF*

Aunque corrige un poco la alta similitud, el resultado de la similitud ponderada queda bastante alto en ciertos casos, de modo que no se puede siempre asegurar una diferencia en la clasificación.

3.4.2. Preprocesamiento de las preguntas

Para atacar el problema de los errores ortográficos, implementamos un corrector de ortógrafo usando el algoritmo siguiente:

1. En cada pregunta, ver si todas las palabras se encuentran en el diccionario. Si una no se encuentra:
    - si se encuentra también en la otra pregunta del par, puede ser un nombre propio: no lo cambiamos
    - si no, la consideramos un error
2. si tenemos un error, miramos a las correcciones propuestas, y tomamos la más cercana en términos de distancia de *Jaro-Winkler*.

El problema con ese algoritmo es que tiene que iterar sobre todas las palabras del conjunto, y tarda mucho. Se consideró que, en comparación con el número bastante débil de errores ortográficos, no se perdió mucho al no usarlo.

3.4.3. Resultados

Queremos evaluar hasta ahora nuestro modelo para ver cómo mejorarlo. Por eso, observamos los resultados obtenidos en una muestra de 50.000 pares (12.5 % del total) del conjunto de entrenamiento con *cross-validation* (Geisser, 1993). Además de la similitud y de la similitud ponderada por *TF-IDF*, añadimos el ratio de palabra en común.

Con eso, eligiendo como clasificador *Random Forest* con 300 árboles, y 10 *folds* en la *cross-validation*, obtenemos un *logarithmic loss* promedio de 0.59. Mostramos los resultados a continuación:

mean_test_score	**mean_train_score**	**param_class_weight**	**n_estimators**
-0.59	-0.18	balanced	300

Tabla 3.8: Resultados obtenidos con 300 árboles. Extraído de la salida del algoritmo *gridsearch* de *scikit-learn*

Mencionemos que esos parámetros fueron elegidos gracias al algoritmo *gridsearch* (Scikit-learn, 2017) que, dados un clasificador y una medida de calidad, entrega los parámetros que la maximicen. Aquí, el parámetro *"balanced"* se refiere al hecho de tomar en cuenta la repartición de los positivos y negativos en el conjunto total al abastecer cada árbol.

Esos resultados no son muy buenos, dado el hecho que la predicción ingenua que consiste a atribuirle 36.9 % a cada par resulta en un *log loss* de 0.66.

Para entender lo que hacemos mal, miremos ahora los falsos positivos y negativos responsables por los *log loss* más grandes:

question1	**question2**	**proba_pred**	**loss**
Can I restore my deleted emails from a Gmail account?	How can I retrieve and recover deleted items from my Gmail after mistakenly deleting and clearing trash?	0.0	34.54
What was there before the "beginning¿	What existed before universe?	0.0	34.54
How do I lose man boobs?	What is the best and quickest way to get rid of man boobs?	0.0	34.54
What is a consultant and what do they do?	What do consultants do?	0.0	34.54
Which is the best laptop under Rs 25000?	Which laptop is better for a enginnering student under Rs.25000?	0.0	34.54
How are fingerprints formed?	How do patterns on fingerprints form?	0.0	34.54

Tabla 3.9: Falsos negativos más penalizadores

Podemos ver que la similitud de las preguntas de la tabla 3.9 es muy clara, pero el hecho que estén formuladas con palabras distintas añade ruido a los vectores promedios.

Vemos como se comporta con los falsos positivos:

question1	**question2**	**proba_pred**	**loss**
What was your scariest travel experience?	What is the scariest experience you remember from your childhood?	0.95	3.00
Why does steak taste better the next day?	Which foods taste better the following day?	0.94	2.81
What are internal links?	What is ïnternal link stacking¿	0.94	2.81
What is the difference between '-' and '-=' operators in C++?	What is the difference between == and = operator in C++?	0.94	2.81
Does anyone understand the substation?	What is a substation?	0.94	2.81
Why didn't Amazon take out the money from my debit card after I ordered an item?	Can I transfer money from credit card to debit card?	0.93	2.66

Tabla 3.10: Falsos positivos más penalizadores

Al revés, las preguntas de la tabla 3.10 tienen muchas palabras en común salvo uno, cuyo aporte cambia totalmente el sentido de la frases.

Abordamos esos problemas en la siguiente sección.

3.5. Propuestas para mejorar el modelo

Un primer diagnóstico que podemos hacer es que hay preguntas que son predichas como idénticas, y que no son del mismo tipo, en el sentido de que buscan distintos tipos de información (ubicación, manera de hacer, fecha, etc). Podríamos añadir *features* que ataquen ese problema entonces.

Un segundo problema se presenta cuando dos preguntas se diferencian solamente por una palabra. Un primer caso es el par siguiente:

"What is impossible to do?" vs *"What is impossible to tell?"*

No podemos confiar en *word2vec* para la tarea de diferenciarlos, pues es probable que *"tell"* y *"do"* no estén tan lejos uno del otro, como verbos comunes de acción.

Un segundo ejemplo:

"What are the best activities in Wigan?" vs *"What are the best activities in Reading?"*

De la misma manera que en el caso anterior, es probable que *word2vec* no nos ayude mucho aquí. Si, por casualidad, ha escuchado de *Wigan* y *Reading*, sabrá que son dos ciudades inglesas, y por lo tanto van a tener una cierta similitud. En el otro caso, si esas ciudades no se encuentran en su diccionario, las va a ignorar y así considerar que está en presencia de dos preguntas idénticas.

Para concluir esta parte, notamos que hay varios tipos de palabras en una pregunta: en particular, los pronombres interrogativos, los verbos y los complementes circunstanciales de lugar tienen mucha importancia al diferir entre dos preguntas: nos pueden ayudar para la clasificación.

3.5.1. Varios tipos de palabras: ¿cómo tomarlos en cuenta?

Una técnica útil para el problema que tenemos es una que mencionamos en la introducción, y que se llama *part-of-speech tagging*. Sirve en particular para reconocer los grupos que mencionamos en la parte anterior: pronombres, verbos, nombres de lugar, etc. Con eso, podríamos comparar los distintos grupos entre sí. Específicamente, en el grupo:

- de los pronombres interrogativos: si son distintos, considerar que las preguntas son distintas
- de los verbos: ver si son sinónimos
- de los complementes de lugar: si son distintos, considerar que las preguntas son distintas

3.5.2. Análisis de bi-grams

Similarmente al análisis anterior, podemos notar que hay palabras, o grupos de palabras, que aparecen muchas veces en las preguntas. En particular, son grupos de palabras interrogativos como:

"what is", *"what are"*, *"how can"*, *"where can"*, *"how do"*, *"what should"*, *"can I"*

y *"best way"* también.

Podríamos analizar la presencia de combinaciones de esos grupos en un par, y ver si así logramos una mejor clasificación.

3.6. Implementación en *Python* del modelo

Para desarrollar nuestro modelo, creamos una clase en *Python*, y un documento de funciones de pre-procesamiento. Su contenido se puede ver en anexo.

Capítulo 4

Análisis literario

En ese problema, podemos notar que no necesitamos representar cada libro, sino que basta encontrar una manera de definir una distancia entre dos. Por eso proponemos varios métodos:

4.1. Elección de los libros

La lista de los libros elegidos se puede encontrar en anexo A. Hacen parte del siglo XIX principalmente, como compromiso entre una lengua moderna y obras de libre acceso. Los autores fueron seleccionados por su influencia en la literatura de ese siglo, y por la misma razón se tomaron los obras más emblemáticas disponibles.

4.2. *Clusters* de palabras

En ese método, vamos a usar la función `most_similar(`*palabra*`,` *n*`)` de *word2vec*, que toma un vector palabra como argumento, y que entrega los n vectores con los cuales el producto punto es el más alto. Después de haber elegido dos libros, usamos los pasos descritos a continuación. Desarrollamos luego un ejemplo.

Proceso propuesto

1. Limpiar los datos: Sacar las partes no importantes (licencia libre del libro, índice, etc.)
2. Prepararlos para *word2vec*: Sacar los *stopwords*, transformar el texto en una lista de frases, y las frases en una lista de palabras (*string*) en el formato *utf-8* (unicode)
3. Entrenar un modelo *word2vec* por libro
4. Tomar *n_comun* palabras en la intersección de los conjuntos de vocabulario de cada modelo

5. Considerar, por cada palabra w, el resultado de la función `most_similar(`w`, `n`)` con ambos modelos. Eso nos entrega la lista de los n palabras más parecidas a w en el modelo 1 ($lista_1$) y en el modelo 2 ($lista_2$).

6. Comparar esas dos listas, y construimos una medida de similitud como sigue: por cada palabra en $lista_1$ y $lista_2$, si la palabra se encuentra en ambas listas, consideramos la diferencia de similitud a la palabra original

$$D(mod_1, mod_2) = 1 - \sum_{\mathrm{i},j lista_1[\mathrm{i}]=lista_2[j]}^{n} |mod_1.sim(w, lista_1[\mathrm{i}]) - mod_2.sim(w, lista_2[j])| \tag{4.1}$$

7. Crear una distancia entre dos libros, promediando esa similitud sobre el número de palabras muestreadas,

4.2.1. Ejemplo ilustrativo

Tomamos dos libros:

- *Germinal* de Émile Zola, publicado en 1885
- *Le Rouge et le Noir* (*Rojo y Negro* en Español) de Stendhal, publicado en 1830.

Antes de construir el modelo, tenemos que transformar cada libro.

Veamos los características de los libros:

```
for text in [rouge_et_noir, germinal]:
    text_file = open(path+text).read()
    peso_str = str(round(os.path.getsize(path+text)/1000000,1))
    tam_str = str(len(corpus_to_sentences(text_file)))
    print 'El libro {} pesa {} megas y contiene {} frases'.format(text, peso_str , tam_str)


    El libro Stendhal-le_rouge_et_le_noir.txt pesa 2.8 megas y contiene 27453 frases
    El libro Zola-germinal.txt pesa 1.1 megas y contiene 10146 frases
```

Listing 1: Peso y número de frases en los libros elegidos

Ahora detallamos como se aplica el procedimiento descrito en la sección anterior a esos dos libros.

1. **Transformación, limpieza**
   En los pasos 1 y 2 se usan las funciones presentadas en el *listing* 2:

```
def corpus_to_sentences(corpus):
"""Transforma el corpus en una lista de frases"""

    tokenizer = nltk.data.load('tokenizers/punkt/french.pickle')

    #1. remover notas del autor
    res = re.sub(ur'\[[0-9]\]',u' ',corpus.decode('utf-8'),re.U)
    #2. Transforma en una lista de frases en minusculas
    res = tokenizer.tokenize(res.lower().strip())
    #3. remover non-letras y non-Unicode símbolos
    res = [re.sub(ur'[\W_]+', u' ', sents, \
                    flags=re.UNICODE) for sents in res]

    return res


def sentences_to_list(sents):
"""Transforma las frases en listas de palabras"""

    stopwords_fr = stopwords.words('french')

    return [sent.split() for sent in sents \
                    if sent not in stopwords_fr]
```

Listing 2: Funciones comentadas para transformar los libros

La limpieza profunda de los datos es laboriosa y podría constituir en si misma un tema de memoria. Todos los libros tienen una forma distinta, y enseñar a una máquina a reconocer el nombre y número de capitulo es difícil. Por eso, tomamos el partido de dejar algunas "suciedades" en los datos, considerando que van a ser estadísticamente despreciables.

Una vez hecha esa etapa, tenemos listas de frases, es decir listas de listas de palabras. Por ejemplo, mostramos aquí la tercera frases de *Germinal*:

```
print germinal_text[2]

   ['aucune', 'ombre', 'd', 'arbre', 'ne', 'tachait', 'le', 'ciel',
    'le', 'pavé', 'se', 'déroulait', 'avec', 'la', 'rectitude', 'd',
    'une', 'jetée', 'au', 'milieu', 'de', 'l', 'embrun',
    'aveuglant', 'des', 'ténèbres']
```

Listing 3: Tercera frase de *Germinal*

2. **Entrenamiento de los modelos**
   Como lo vimos más arriba, los libros tienen largos distintos. Sin embargo, se requiere encontrar parámetros que se adaptan al libro considerado. En particular, no vamos a usar el mismo número de dimensiones con libros de 30 o 800 páginas. Para entender eso, imaginamos dos casos extremos: no tendría sentido entrenar una única frase ("*the cat is sat on the mat*") en 300 dimensiones. Por otro lado, si tenemos como conjunto de entrenamiento todo *Wikipedia*, tampoco sería muy eficiente usar una sola dimen-

sión. Por eso, tenemos que determinar un criterio que nos permita elegir el número de dimensiones.

Una posibilidad sería considerar las palabras dadas por la función `most_similar` para palabras simples, como "hombre" o "mujer" en varias dimensiones, y evaluar los resultados a simple vista.

En nuestro caso, proponemos – de manera heurística – el criterio siguiente (por tema de velocidad, las dimensiones usadas tienen que ser múltiples de 4):

Tamaño del libro	**Dimensión usada**
Menos de 0.5 Mo	48
Entre 0.5 Mo y 2 Mo	72
Más de 2 Mo	100

Tabla 4.1: Criterio de dimensionalidad de los espacios

Entrenamos los libros con el código siguiente:

```
def text_to_model(text):
"""Transforma la lista de frases en un modelo w2v"""

    if round(os.path.getsize(path+text)/1000000,1) < 0.5 :
        dim = 48
    elif round(os.path.getsize(path+text)/1000000,1) < 2:
        dim = 72
    else:
        dim = 100

    text_file = open(path+text).read()
    text_file = sentences_to_list(corpus_to_sentences(text_file))

    return to_w2v(text_file,dim)
```

Listing 4: Función para entrenar los modelos

Obtenemos de *word2vec* las informaciones siguientes:

```
2017-06-07 21:28:40,174 : INFO : collecting all words and their counts
2017-06-07 21:28:40,176 : INFO : PROGRESS: at sentence #0, processed 0 words, keeping 0 word
 types
2017-06-07 21:28:40,274 : INFO : PROGRESS: at sentence #10000, processed 174545 words, keepin
g 13656 word types
2017-06-07 21:28:40,277 : INFO : collected 13743 word types from a corpus of 177190 raw words
and 10146 sentences
2017-06-07 21:28:40,278 : INFO : Loading a fresh vocabulary
2017-06-07 21:28:40,353 : INFO : min_count=1 retains 13743 unique words (100% of original 137
43, drops 0)
2017-06-07 21:28:40,355 : INFO : min_count=1 leaves 177190 word corpus (100% of original 1771
90, drops 0)

Training model...

2017-06-07 21:28:40,460 : INFO : deleting the raw counts dictionary of 13743 items
2017-06-07 21:28:40,462 : INFO : sample=0.0002 downsamples 211 most-common words
2017-06-07 21:28:40,464 : INFO : downsampling leaves estimated 99212 word corpus (56.0% of pr
ior 177190)
2017-06-07 21:28:40,466 : INFO : estimated required memory for 13743 words and 72 dimensions:
21494052 bytes
2017-06-07 21:28:40,508 : INFO : constructing a huffman tree from 13743 words
2017-06-07 21:28:41,415 : INFO : built huffman tree with maximum node depth 18
2017-06-07 21:28:41,502 : INFO : resetting layer weights
2017-06-07 21:28:41,833 : INFO : training model with 4 workers on 13743 vocabulary and 72 fea
tures, using sg=1 hs=1 sample=0.0002 negative=5 window=10
2017-06-07 21:28:41,835 : INFO : expecting 10146 sentences, matching count from corpus used f
or vocabulary survey
```

Fig. 4.1: Informaciones recuperadas al entrenar el algoritmo sobre *Germinal*

Notemos que el *down-sampling* de las 200 palabras más comunes divide por dos el número de palabra total en el conjunto de entrenamiento.

3. **Distancia entre libros**
   Ahora, para efectuar los pasos 4 a 7 descritos en la sección 4.2., usamos la función siguiente:

```
1  def dist_word(pal,mod):
2      """toma una palabra, y entrega la distancia entre dos modelos"""
3      topn_val = 30;
4      result = pd.merge(pd.DataFrame(mod[0].most_similar(pal,topn=topn_val),\
5                                     columns=['palabra','similaridad_mod_1']),\
6                        pd.DataFrame(mod[1].most_similar(pal,topn=topn_val),\
7                                     columns=['palabra','similaridad_mod_2']),\
8                        how='inner', on='palabra')
9
10     return abs(result['similaridad_mod_1']-result['similaridad_mod_2']).mean()
```

Listing 5: Función para establecer la distancia entre dos modelos sobre una palabra

De ahí, nos basta aplicar esta función para el numero de palabras elegido en la intersección de dos libros, n_sample:

```python
def dist_mod(mod):
    """toma dos modelos, y aplica la función dist_word sobre n_sample palabras"""
    n_sample = 250;
    set_intersect = intersect(mod[0].wv.vocab.keys(),mod[1].wv.vocab.keys())
    n_intersect = len(set_intersect)
    words = random.sample(set_intersect,min(n_sample,n_intersect))

    sim_words = pd.DataFrame(columns=['palabras','diff_sim'])
    sim_words['palabras'] = words
    sim_words['diff_sim'] = sim_words['palabras'].apply(dist_word, args=(mod,))

    return sim_words['diff_sim'].mean()
```

Listing 6: Función para establecer la distancia entre dos modelos con n_sample palabras

Para darse un poco de intuición, la aplicamos a algunos ejemplos, y explicamos luego los resultados. Empezamos con la palabra *"homme"* ("hombre"):

palabra	traducción	sim_mod_1	sim_mod_2
jeune	joven	0.80	0.61
boisait	enmaderaba	0.73	0.61
disparût	desapareció	0.69	0.68
mécompte	desengaño	0.68	0.62
huerait	abuchearía	0.66	0.61

Tabla 4.2: intersección de `mod[i].most_sim("homme")` para i = ["Germinal", "Le Rouge et le Noir"]

Resumimos lo que acabamos de hacer: tomamos las 30 palabras más parecidas a *"homme"* en *Germinal* y en *Le Rouge et le Noir*. Eso nos entregó dos listas de 30 palabras, cuya intersección aparece en la tabla 4.2. Ahí podemos definir la distancia entre los dos modelos en la palabra *"homme"* tomando el promedio de la diferencias de similitud.

Ahora con la palabra "*alcool*" ("alcohol"):

palabra	traducción	sim_mod_1	sim_mod_2
détraquée	descuadrada	0.94	0.89
ascendance	ascendencia	0.93	0.86
ivrognes	borrachos	0.87	0.78
poison	veneno	0.86	0.73
homicide	homocidio	0.83	0.77

Tabla 4.3: intersección de `mod[i].most_sim("alcool")` para i = ["Germinal", "Le Rouge et le Noir"]

La distancia promedia entre los dos modelos en la palabra *"homme"* aquí sería de 0.085, y de 0.077 en *"alcool"*.

Luego, nos basta promediar esas distancias sobre n_sample palabras en la intersección de este par de libros.

Para reducir el sesgo debido a la elección de palabras particulares, aplicamos esa función sobre $n\_sample = 250$ palabras comunes a ambos textos, considerando en cada libros las 30 palabras más similares. Eso nos hace la pregunta de la eficiencia del algoritmo, pues son 45 libros, o sea 990 pares de libros. En promedio, el algoritmo tarda 4 segundos por par, o sea 3960 segundos en total.

4.2.2. Comentarios

Con ese proceso surgen dos limites. El primero es que implica de abastecer solamente un libro por modelo, mientras que los resultados exitosos de *word2vec* fueron obtenidos con conjunto muchos más grandes (tipo 1 millón). El segundo problema es que puede ocurrir que no haya palabras en común en las dos listas, sino palabras de sentido cercano, como se presenta en la sección 4.3.2. Ahí intentamos resolver esos problemas, introduciendo unas propuestas de métodos mejorados.

4.2.3. Clustering

Una vez obtenida la distancia entre cada par de libros, podemos crear una matriz de distancia. De ahí, existen métodos para crear *clusters*, como el *hierarchical clustering* que presentamos en lo anterior.

El número de *clusters* buscado siempre es un parámetro del algoritmo, y se puede considerar que su elección se hace bajo criterios subjetivos, pues se trata de aprendizaje no-supervisado. Consideramos que un *cluster* es interesante cuando junta obras de un mismo estilo, o de un mismo autor.

En nuestro caso, los resultados parecieron interesantes para números de *clusters* entre 7 y 12, y en particular para 8: es la cantidad que fijamos.

4.2.4. Resultados

A continuación presentamos los resultados obtenidos:

Clusters	Libros					
1	Flaubert - Madame Bovary	Gautier - Mademoiselle de Maupin	Hugo - Quatre-vingt treize	Nerval - Voyage en Orient	Verne - Vingt-mille lieues sous les mers	Zola - L'Argent
	Zola - L'Assommoir	Zola - La Bête humaine	Zola - La Curée	Zola - Nana	Zola - Au Bonheur des dames	
2	Hugo - Les Contemplations	Verne - Autour de la Lune	Verne - Voyage au centre de la Terre	Verne - Le Tour du monde en 80 jours	Verne - De la Terre à la lune	Zola - Thérèse Raquin
3	Daudet - Lettres de mon moulin	Maupassant - Boule de Suif	Maupassant - Pierre et Jean	Maupassant - Contes du jour et de la nuit	Maupassant - Le Horla	Voltaire - Candide
4	Chateaubriand - Mémoires d'Outre-Tombe	Dumas - Le Comte de Monte Cristo	Hugo - Les Misérables	Lamartine - Cours familier de Littérature	Stendhal - Le Rouge et le Noir	
5	Dumas - Les Trois Mousquetaires	Hugo - Notre Dame de Paris	Hugo - L'Homme qui rit	Stendhal - La Chartreuse de Parme	Verne - L'ile Mysterieuse	
6	Hugo - La Esmeralda	Molière - L'avare	Voltaire - Jeannot et Colin	Voltaire - Micromegas		
7	Mallarmé - Pages	Rimbaud - Poesie complète	Stendhal - La Duchesse de Palliano			
8	Beaudelaire - Les Fleurs du Mal	Hugo - Le Dernier jour d'un condamné	Maupassant - Clair de Lune	Voltaire - L'ingénu		

Tabla 4.4: Los ocho *clusters* que obtuvimos con el método de *clusters de palabras*

Examinemos el contenido de los *clusters*:

1. El primero contiene novelas clásicas. Está toda la colección *"Les Rougon-Macquart"* de Zola, así que *Madame Bovary*, entre otros.

2. El segundo contiene casi todo Jules Verne

3. El tercero parece ser el de Maupassant

4. El *cluster* 4 parece contener grandes novelas emblemáticas: *"Les Misérables"*, *"Le Rouge et le Noir"*, *"Le Comte de Monte-Cristo"*

5. El 5 contiene novelas de aventura

6. El 6 contiene *"La Esmeralda"* y *"L'Avare"* que son las dos obras de teatro de este corpus. Las obras de Voltaire son dos cuentos filosóficos

7. Con Mallarmé y Rimbaud, el cluster 7 contiene dos de las tres obras de poesía de ese corpus

8. El 8 contiene el resto

Ese reagrupamiento, aunque no está perfecto, parece relevante en los sentidos siguientes:

- toma en cuenta una cierta paternidad. Por lo menos, parece reconocer las obras de Zola, Maupassant y Verne.
- parece reconocer el estilo: las obras de teatro y de poesía casi tuvieron su propio *cluster* respectivo
- parece reconocer el género, en particular en los grupos 1 y 4

Sin embargo, una duda subsiste con el *cluster* 4: además de ser las obras las más emblemáticas de este corpus, son también las mas largas. Ese puede haber influido el algoritmo

que propusimos.

4.3. Propuestas de mejoramiento de resultados

Proponemos algunos cambios que podrían mejorar nuestro modelo.

4.3.1. Un método estadístico

Una idea alternativa sería de construir un modelo probabilístico, no entero, sino solamente de las palabras de intersección entre dos libros. Dada una palabra en esa intersección, podríamos ver que tan probable es que está seguida por otra palabra, entre el libro 1 y en el libro 2, y así crear una medida de distancia. En más detalles:

4.3.2. Lematización y círculos de vecinos palabras

Para atacar el segundo problema del método de *clusters de palabras*, consideramos el hecho siguiente: puede ser que una palabra dé lugar a palabras distintas en dos listas, pero con sentidos cercanos.

Lematización

Por ejemplo, en la lista_1 de la palabra "peur" (*miedo*), puede aparecer "ai" (*tengo*) y en la segunda "avons" (*tenemos*), sin que el primero se encuentre el la segunda lista, y *vice versa*. En el caso particular de los verbos, un método para aplicar sería de reducir "avons" y "ai" a "avoir" (*tener*). Eso es la *lematización* que presentamos en introducción.

Círculos de vecinos de palabras

Para ilustrar el caso más general, consideremos el caso siguiente con la palabra "père" (padre). Supongamos que, en la lista_1, la única palabra de sentido próximo que nos da es "famille" (familia) y en la lista_2 "filiation" (filiación). Según nuestra medida, no habrá similitud entre los dos modelos, mientras que "famille" y "filiation" si tienen sentidos cercanos. Lo que proponemos entonces es considerar:

- `mod_1.most_similar(‘‘famille’’)`, y ver si "filiation" se encuentra.
  - Si tal es el caso, tomamos como distancia:

$$|\texttt{mod\_1.sim(‘‘père’’,‘‘famille’’)*mod\_1.sim(‘‘famille’’,‘‘filiation’’)} - \texttt{mod\_2.sim(‘‘père’’,‘‘filiation’’)}| \tag{4.2}$$

- `mod_2.most_similar(‘‘filiation’’)`, y ver si “famille” se encuentra.
  - Si tal es el caso, y que la condición anterior se cumple, tomamos como distancia el mínimo entre:
    * `|mod_1.sim(‘‘père’’,‘‘famille’’ )*mod_1.sim(‘‘famille’’,‘‘filiation’’ ) - mod_2.sim(‘‘père’’,‘‘filiation’’)|`
    * `|mod_1.sim(‘‘père’’,‘‘famille’’ ) - mod_2.sim(‘‘père’’,‘‘filiation’’)*mod_2.sim(‘‘filiation’’,‘‘famille’’ )|`

    (o sea el camino más corto entre los dos modelos)
  - Si tal es el caso, y que la condición anterior *no* se cumple, tomamos:

$$|\texttt{mod\_1.sim(‘‘père’’,‘‘famille’’)} - \texttt{mod\_2.sim(‘‘père’’,‘‘filiation’’)*mod\_2.sim(‘‘filiation’’,‘‘famille’’)}| \tag{4.3}$$

En pocas palabras, lo único que cambia con respecto al método de la sección anterior es que consideramos grados adicionales (o círculos más largos) de vecinos de una palabra. Sin embargo, ese método alargaría considerablemente el tiempo de computación de nuestro algoritmo.

4.3.3. Ampliar el tamaño de los modelos

En todo lo anterior, el problema que tenemos es la pertinencia de las palabras más cercanas de una. Una manera de arreglar ese problema es crear modelos a partir de conjuntos más grandes. Después, podríamos atribuirle un vector a cada libro y, con un método de representación (como *t-SNE*), podemos hacer una visualización gráfica de esos libros, y buscar *clusters*.

Por eso, proponemos dos métodos:

Un único modelo *word2vec*

En esa parte, se trata de concatenar todos los textos que tenemos, para construir un único modelo.

Una vez el modelo construido, el procedimiento propuesto es el siguiente:

1. Tomar n muestras de párrafos en dos libros.
2. Tomar un vector promedio por cada párrafo: obtenemos n vectores por libros.
3. De ahí, elegir un vector en el libro 1, y tomar la distancia con el vector más cercano en el libro 2.
4. Iterar hasta n.

Con eso, proponemos otra manera de definir una distancia entre dos libros.

Comparar no libros sino autores

En la literatura francesa, algunos autores fueron muy prolíficos: Zola, Balzac, Dumas, entre otros, escribieron más de 30 libros. La obra entera de uno de esos autores puede constituir un buen soporte para entrenar un modelo. Así, con lo anterior podríamos definir distancia ya no entre libros sino entre autores, pero eso necesitaría mucho más libros que el corpus propuesto.

Capítulo 5

Conclusión

5.1. Resumen del trabajo hecho

En este trabajo de memoria, quisimos hacer una introducción al procesamiento de lenguaje natural, mediante el algoritmo *word2vec*.

Después de haber introducido algunas técnicas de NLP, presentamos un ejemplo de utilización de *word2vec*, con su interés y sus limites. Para terminar, aplicamos ese algoritmo al análisis de obras de literatura, y así pusimos en evidencia la existencia de palabras comunes, así que de asociaciones de palabras comunes a dentro de las obras de un autor, de un estilo o de un género.

Empezamos por una introducción al procesamiento de lenguaje natural: presentamos el contexto histórico de su nacimiento, el estado del arte y algunas aplicaciones prometedoras.

Presentamos algunas técnicas elementales de NLP:

- al nivel de una palabra: *Regular expression*, tokenización, *stopwords*, lematización, distancia de edición mínima
- al nivel de una frase: segmentación de frases, estudio de *n-grams*
- al nivel de un texto: análisis de sentimiento (con *bag-of-words*)

Una vez introducidos con esos elementos, presentamos el algoritmo *word2vec*, que propone representaciones vectoriales de palabras en espacio de tamaño reducido. Ese algoritmo, cuyo modelo fue inventado por Bengio et al. (2003), fue hecho famoso por el equipo de *Google* de Mikolov et al. (2015) por su implementación muy rápida. Explicamos como funciona esa implementación, y cuales son sus principales parámetros. Luego, presentamos dos métodos más avanzados de clasificación de documentos: *Latent Semantic Analysis* y *Latent Dirichlet Allocation*.

En una segunda parte, intentamos resolver el problema de detección de preguntas duplicadas en *Quora* con *word2vec*. Vemos como ese último logra resultados decentes, aún aplicado

de manera ingenua, es decir tomando vectores promedios de preguntas. Analizamos cuales son los puntos fuertes y débiles de nuestro modelo, y luego proponemos algunas opciones para mejorar nuestros resultados.

Por fin, proponemos un análisis de 45 obras de literatura francesa del siglo XIX mediante *word2vec*. Se trata de comparar los libros por pares, usando una distancia derivada de métodos de *word2vec*, que toma en cuenta las palabras más parecidas a una palabra que se encuentra en ambos libros del par.

Primero, se empieza por explicar como podemos limpiar los datos con el fin de aplicar el algoritmo. Damos algunos detalles sobre la manera de configurar los parámetros, en particular en una situación de aprendizaje no-supervisado. Luego, se estudia la presencia de *clusters*. Se hace la pregunta de la repartición de los libros en ellos según el número de grupos que buscamos, y en particular, se investiga el sentido que les podemos atribuir a cada uno. Específicamente, se nota que con 8 *clusters*, obtenemos resultados interesantes: en casi todos se revela un atributo común, que sea la presencia de un mismo un autor, la predominancia de un genero literario (novelas, teatro, o poesía) o de un mismo estilo (novela de aventura o "clásica"). Eso atesta de la relevancia de nuestro procedimiento. Sin embargo, le observamos algunas debilidades y por eso terminamos por hacer propuestas para mejorarlo, y confirmar nuestros resultados.

5.2. Trabajo futuro

A lo largo de nuestro estudio, propusimos cada vez mejoras de nuestras modelizaciones. Sería interesante implementar esos métodos para compararlos con nuestros resultados: además, podría ser una manera de supervisar el aprendizaje de nuestro modelo.

Por otra parte, mencionamos que existía la posibilidad de que algunos grupos fueran formados por atributos técnicos (como el tamaño del archivo por ejemplo): para descartar esa posibilidad, necesitamos probar nuestro algoritmo en conjuntos de libros más largos, con libros más diversos: de distintas épocas, de estilos distintos, etc.

Por fin, sería interesante investigar por el lado de la paternidad de los libros: en particular, un proyecto relevante consistiría en elegir libros de toda América del Sur, y ver si se forman *clusters* según el origen geográfica. Lo único que nos detiene hacer eso es el carácter reciente de esa literatura: la mayoría de los libros fueron publicados en el último siglo, así que pocos son de libre acceso.

Bibliografía

Bengio, Y., Ducharme, R., y Vincent, P. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, *3*, 1137–1155.

Beth, J. (1968). *Development of a stemming algorithm* (Vol. Mechanical Translation and Computational Linguistics).

Blei, D. M., Ng, A. Y., y Jordan, M. I. (2002). Latent dirichlet allocation. *Journal of Machine Learning Research*, *3*, 2003.

Breiman, L. (2001). Random forests. *Machine learning*, *45*(1), 5–32.

Broder, Glassman, Manasse, y Zweig. (1997). *Syntactic clustering of the web* (Vol. 29).

Chomsky, N. (1957). Syntactic structures. *Mouton & Co*, *abs/1511.08130*.

Citron. (2017). *Citron, le chatbot pour trouver des lieux où sortir à paris.* Descargado de `http://www.tom.travel/2017/02/27/citron-le-chatbot-pour-trouver-des-lieux-ou-sortir-a-paris/`

Cohen, W. W., Ravikumar, P., y Fienberg, S. E. (2003). *A comparison of string distance metrics for name-matching tasks.* Descargado de `https://www.cs.cmu.edu/afs/cs/Web/People/wcohen/postscript/kdd-2003-match-ws.pdf`

Davis, H. (2014). *Transprose : translating literature into music.* Descargado de `http://www.musicfromtext.com/`

Geisser. (1993). *Predictive inference.* Chapman and Hall.

Harris, D., y Harris, S. (2007). *Digital design and computer architecture.*

Hinton, G. (2011). *Distributed representations.*

Huang, Alleva, Hon, Hwang, y Rosenfeld. (1992). *The sphinx-ii speech recognition system: An overview* (Vol. 7).

IBM. (2005). *Alchemy api.* Descargado de `https://www.alchemyapi.com/`

Ko, Y. (2012). A study of term weighting schemes using class information for text classification. *SIGIR'12 ACM*.

Kullback, y Leibler. (1951). On information and sufficiency. *Annals of Mathematical Statistics*, *22*(1), 79–86.

Le, Q. V., y Mikolov, T. (2014). Distributed representations of sentences and documents. *CoRR*, *abs/1405.4053*. Descargado de http://arxiv.org/abs/1405.4053

Messenger. (2011). Descargado de www.messenger.com/

Mikolov, T. (2013). *Learning representations of text using neural networks.*

Mikolov, T., Chen, K., Corrado, G., y Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, *abs/1301.3781*. Descargado de http://arxiv.org/abs/1301.3781

Mikolov, T., Joulin, A., y Baroni, M. (2015). A roadmap towards machine intelligence. *CoRR*, *abs/1511.08130*. Descargado de http://arxiv.org/abs/1511.08130

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., y Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *CoRR*, *abs/1310.4546*. Descargado de http://arxiv.org/abs/1310.4546

Mikolov, T., Yih, W., y Zweig, G. (2013). Linguistic regularities in continuous space word representations. *HLT-NAACL*, 746–751.

Mitkov, R. (2003). *The oxford handbook of computational linguistics.* Descargado de https://books.google.cl/books?id=yl6AnaKtVAkC&pg=PA754&redir_esc=y#v=onepage&q&f=false

Muller, T., Cotterell, R., Fraser, A., y Schutze, H. (2014). *Joint lemmatization and morphological tagging with lemming.* Descargado de http://www.cis.lmu.de/~muellets/pdf/emnlp_2015.pdf

Navarro, G. (2001). *A guided tour to approximate string matching* (Vol. 33). Descargado de http://repositorio.uchile.cl/bitstream/handle/2250/126168/Navarro_Gonzalo_Guided_tour.pdf

NLTK. (2014). *Natural language processing with python – analyzing text with the natural language toolkit.* Descargado de http://www.nltk.org/book/

pandas. (2017). *10 minutes to pandas.* Descargado de http://pandas.pydata.org/pandas-docs/stable/10min.html

Rajaraman, A., y Ullman, J. D. (2011). *Mining of massive datasets.* Descargado de http://i.stanford.edu/~ullman/mmds/ch1.pdf

Rokach, y Maimon. (2008). *Data mining with decision trees: theory and applications.* World Scientific Pub Co Inc.

Rokach, y Maimon, O. (2005). *Clustering methods.* Springer.

Scikit-learn. (2017). *uning the hyper-parameters of an estimator.* Descargado de `http://scikit-learn.org/stable/modules/grid_search.html#grid-search`

Shrivastava, G. (2016). *Weather reports with poncho, the cat chatbot.* Descargado de `https://whatarechatbots.com/weather-reports-with-poncho-the-cat-ea6d0de9cdbb`

Socher, R. (2013). *Deep learning for nlp (without magic).*

Sudderth, E., Torralba, A., Freeman, W. T., y Willsky, A. S. (2013). *Describing objects in visual scenes: Is visual.*

Textio. (s.f.). *Textio, a startup that analyzes text performance, raises $8m.* Descargado de `https://techcrunch.com/2015/12/16/textio-a-startup-that-analyzes-text-performance-raises-8m/`

Turing, A. (1950). Computing machinery and intelligence. *Mind*, *LIX*, 433-460. Descargado de `http://loebner.net/Prizef/TuringArticle.html`

van der Maaten, L., y Hinton, G. (2008). Visualizing data using t-sne. En *Cost-sensitive machine learning for information retrieval 33.*

Whatsapp. (2009). Descargado de `https://www.whatsapp.com/`

Anexos

Anexo A

Modelo *Python* para el problema de *Quora*

A continuación mostramos algunas partes importantes de la clase que definimos en *Python*

```
# -*- coding: utf-8 -*-
import pandas as pd
import numpy as np
import logging as log
import functions_preproc as fn_pp
import re

from scipy import spatial
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression as LogReg
from sklearn.metrics import log_loss as loss
from sklearn.metrics import confusion_matrix as cm
from sklearn.metrics import f1_score, make_scorer, confusion_matrix, \
        accuracy_score, precision_score, recall_score
from sklearn.model_selection import cross_val_score as CV
from sklearn.model_selection import cross_val_predict
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import GridSearchCV as GS_CV
from sklearn.feature_extraction.text import TfidfVectorizer
from functools import reduce
from nltk.corpus import stopwords
log.basicConfig(level=log.DEBUG, format='%(asctime)s %(message)s')
```

Listing 7: Lista de paquetes importado

```
class Model:
    def __init__(self, trainDataset,  testDataset=None, N_train=False,model = LogReg):
        self.Classif = model() #RF, SVM, Reglog
        self.FeatureFunctions = [self.feat1,self.feat2,self.feat4,self.feat6,self.feat7]
        self.FeatureName = ["sim", "sim_pond","common_words_ratio","normed_diff_pond","normed_diff"]
        if N_train:
            self.trainData = trainDataset.sample(N_train, random_state=23)
        else:
            self.trainData = trainDataset
        self.testData = testDataset if testDataset is not None else pd.DataFrame()
        self.loadFeatures()
```

Listing 8: Constructor de la clase definida

```
    def ComputeFeatures(self, test = False, features= None):

        if features is None:
            features = range(len(self.FeatureFunctions))
        if not test:
            for f in features:
                log.debug('Calculando features %s ...'%f)
                applyTemp = self.trainData.apply(self.FeatureFunctions[f], axis=1)
                log.debug('Hecho !')
                if isinstance(applyTemp, pd.Series):
                    log.debug(self.FeatureName[f])
                    self.Features[self.FeatureName[f]] = applyTemp
                else:
                    for col in applyTemp.columns:
                        self.Features[self.FeatureName[f] + col] = applyTemp[col]
                        log.debug(self.FeatureName[f]+col)
        else:
            for f in features:
                log.debug('Calculando features %s ...' % f)
                applyTemp = self.trainData.apply(self.FeatureFunctions[f], axis=1)
                log.debug('Hecho !')
                if isinstance(applyTemp, pd.Series):
                    self.testFeatures[self.FeatureName[f]] = applyTemp
                else:
                    for col in applyTemp.columns:
                        self.testFeatures[self.FeatureName[f] + col] = applyTemp[col]
```

Listing 9: Función que calcule los *features*

```
    def Preprocessing(self):
        #Funcion de preprocesamiento de string
        log.debug("Botando los NaNs...")
        self.trainData = self.trainData.dropna(how="any")
        log.debug("Hecho !")

        log.debug("Limpiando las preguntas...")
        STOP_WORDS = set(stopwords.words())
        regex = re.compile('([^\s\w]|_)+')
        for col,col_mod in zip(['question1', 'question2'],['question1_mod', 'question2_mod']):
            self.trainData[col_mod] = self.trainData[col].\
            apply(fn_pp.clean_sentence,args=[STOP_WORDS,regex])
        log.debug("Hecho !")

        log.debug("Creando el corpus...")
        corpus = self.trainData['question1'].tolist() + self.trainData['question2'].tolist()
        log.debug("Hecho ! Corpus : %s"%corpus[:10])

        log.debug("Entrenando TFIDF ...")
        tfidf_list_q1 = {}
        tfidf_list_q2 = {}
        tf = TfidfVectorizer(input='content', analyzer='word', ngram_range=(0,1),
                     min_df = 0, stop_words = 'english', sublinear_tf=True)
        tfidf_matrix = tf.fit_transform(corpus)
        feature_names = tf.get_feature_names()
        for i,index in enumerate(self.trainData.index):
            tfidf_list_q1[index] = fn_pp.tfidf_weights(i, tfidf_matrix, feature_names)
            tfidf_list_q2[index] = fn_pp.tfidf_weights(i+len(self.trainData), \
                tfidf_matrix, feature_names)
        log.debug("Hecho ! Lista : %s"%tfidf_list_q1.items()[:10])
        self.trainData['dict_tfidf_q1'] = pd.Series(tfidf_list_q1)
        self.trainData['dict_tfidf_q2'] = pd.Series(tfidf_list_q2)
```

Listing 10: Función que calcule los *features*

Anexo B

Lista de libros considerados

Título original	Título castellano	Autor	Año	Peso	Comentario
Les Fleurs du Mal	Las Flores del mal	C. Beaudelaire	1857	153 Ko	Poesía
Mémoires d'Outre-Tombe	Memorias de Ultratumba	Chateaubriand	1850	3.2 Mo	Autobiografía
Lettres de mon Moulin	Cartas desde mi molino	A. Daudet	1879	287 Ko	Novela
Les Trois Mousquetaires	Los Tres Mosqueteros	A. Dumas	1844	1.4 Mo	Novela (Aventura)
Le Comte de Monte-Cristo	El Conde de Montecristo	A. Dumas	1844	2.9 Mo	Novela (Aventura)
Madame Bovary	Madame Bovary	G. Flaubert	1857	715 Ko	Novela
Madame de Maupin	Madame de Maupin	T. Gautier		705 Ko	Novela
Le Dernier Jour d'un condamné	El último día de un condenado	V. Hugo	1829	158 Ko	Novela
La Esmeralda	La Esmeralda	V. Hugo	1829	158 Ko	Teatro
L'Homme qui rit	El Hombre que ríe	V. Hugo	1869	1.3 Mo	Novela
Les Contemplations	Las Contemplaciones	V. Hugo	1856	294 Ko	Poesía
Les Misérables	Los Miserables	V. Hugo	1862	3.2 Mo	Novela
Notre Dame de Paris	Nuestra Señora de París	V. Hugo	1831	1.1 Mo	Novela
Quatre-vingt treize	Noventa y tres	V. Hugo	1874	721 Ko	Novela
Cours Familier de littérature	Curso familiar de literatura	A. Lamartine		3.2 Mo	Novela
Pages	Páginas	S. Mallarmé	1891	142 Ko	Poesía
Boule de Suif	Bola de Sebo	G. Maupassant	1870	230 Ko	Novela
Clair de Lune	Bajo la Luna	G. Maupassant	1883	157 Ko	Novela
Contes du jour et de la nuit	Cuentos del día y de la noche	G. Maupassant	1885	258 Ko	Novela
Le Horla	El Horla	G. Maupassant	1886	257 Ko	Novela
Pierre et Jean	Pierre y Jean	G. Maupassant	1887	263 Ko	Novela
L'avare	El avaro	Molière	1668	142 Ko	Teatro
Voyage en Orient	Viaje al Oriente	G. Nerval	1851	874 Ko	Novela
Poésie complète	Poesía completa	A. Rimbaud	1875	98 Ko	Poesía

Tabla 5.1: Lista de libros (1/2)

Título original	Título castellano	Autor	Año	Peso	Comentario
La Chartreuse de Parme	La Cartuja de Parma	Stendhal	1841	1.1 Mo	Novela
La Duchesse de Palliano	La Duqueza de Palliano	Stendhal	1837	52 Ko	Novela
Le Rouge et le Noir	Rojo y Negro	Stendhal	1830	2.8 Mo	Novela
Autour de la Lune	Alrededor de la Luna	J. Verne	1870	372 Ko	Novela (aventura)
De la terre à la Lune	De la Tierra a la Luna	J. Verne	1865	357 Ko	Novela (aventura)
L'île mystérieuse	La isla misteriosa	J. Verne	1875	1.3 Mo	Novela (aventura)
Le Tour du monde en 80 jours	La vuelta al mundo en ochenta días	J. Verne	1873	439 Ko	Novela (aventura)
Vingt-mille lieues sous les mers	Veinte mil leguas de viaje submarino	J. Verne	1870	921 Ko	Novela (aventura)
Voyage au Centre de la Terre	Viaje al centro de la Tierra	J. Verne	1864	440 Ko	Novela (aventura)
Candide	Cándido	Voltaire	1759	205 Ko	Cuento filosófico
Jeannot et Colin	Jeannot y Colin	Voltaire	1764	19 Ko	Cuento filosófico
L'Ingénu	L'ingénu	Voltaire	1767	144 Ko	Cuento filosófico
Micromégas	Micromegas	Voltaire	1752	45 Ko	Cuento filosófico
Au Bonheur des dames	El Paraíso de las Dames	E. Zola	1883	954 Ko	Novela (colección *"Les Rougon-Macquart"*)
Germinal	Germinal	E. Zola	1885	1.1 Mo	Novela (colección *"Les Rougon-Macquart"*)
L'Argent	El Dinero	E. Zola	1891	918 Ko	Novela (colección *"Les Rougon-Macquart"*)
L'Assommoir	L'Assommoir	E. Zola	1877	995 Ko	Novela (colección *"Les Rougon-Macquart"*)
La Bête Humaine	La Bestia humana	E. Zola	1890	803 Ko	Novela (colección *"Les Rougon-Macquart"*)
La Curée	La Jauría	E. Zola	1872	667 Ko	Novela (colección *"Les Rougon-Macquart"*)
Nana	Nana	E. Zola	1880	912 Ko	Novela (colección *"Les Rougon-Macquart"*)
Thérèse Raquin	Thérèse Raquin	E. Zola	1867	421 Ko	Novela

Tabla 5.2: Lista de libros (2/2)