Universidad de Murcia

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Aplicación de Word Embeddings para el desarrollo de un sistema de preguntas/respuestas basado en analogías

---

Autor:

José Reverte Cazorla - jose.revertec@um.es

Tutores:

Jose Antonio Miñarro Giménez - jose.minyarro@um.es

Catalina Martínez Costa - cmartinezcosta@um.es

05 de septiembre de 2021

Agradecimientos

Primero, gracias a mi hermana Isabel Reverte Cazorla, graduada en Derecho y estudiante del máster de abogacía en la Universidad de Murcia por ayudarme en la evaluación de los modelos, de la interfaz y en la creación del dataset de analogías.

Gracias también a buen amigo Félix Conesa Pérez estudiante de Ingeniería Industrial en la Universidad Politécnica de Madrid por ayudarme con el modelo de evaluación y la implementación del mismo.

Resumen

En el mundo de la Inteligencia artificial, el Procesamiento del Lenguaje Natural (PLN) actualmente se encuentra en auge gracias a estudios de las últimas dos décadas dirigidos a sus aplicaciones como, por ejemplo, el análisis y comprensión del lenguaje. Es tal el interés en estos campos que nacen los Word Embeddings como herramientas para procesar el lenguaje de tal manera que la máquina pueda analizarlo y comprenderlo entrenando estos a partir de un gran corpus de texto libre, trayendo así el lenguaje humano a los computadores[1][2].

Dentro del ámbito de los Word Embeddings, existen varias aplicaciones o 'tareas' para las que estos podrían ser entrenados[3]. Este es el caso de las analogías[4], pues en la llegada de las primeras herramientas de generación de embeddings se observó como las distancias entre las palabras de cierto vocabulario, al representarlas en el espacio vectorial del embedding, eran directamente dependientes de la relación que existía entre dichas palabras, siendo muy cortas entre palabras muy similares y viceversa[5]. Este hecho hace posible la existencia de analogías en el embedding, formando paralelogramos en el espacio vectorial siendo los vértices de este los vectores las cuatro palabras de la analogía[6].

Este trabajo no busca realizar un estudio de este hecho, si no aprovecharse de él para implementar una sistema de preguntas/respuestas basado en analogías del ámbito jurídico español. Para ello (i) se ha obtenido un gran corpus representativo de este dominio, (ii) se ha realizado un estudio de la precisión de los modelos embedding generados dependiendo de la herramienta de generación y sus parámetros de entrenamiento para hacernos con el mejor embedding posible y (iii) se ha creado una interfaz web para el uso de dicho modelo.

Para la obtención del corpus se ha desarrollado un extractor web, para el estudio de la precisión, un script que entrena, guarda y evalúa los modelos generados con las herramientas y parámetros de entrenamiento deseados y, para la interfaz, una sencilla web de uso.

Apoyándonos en nuestro estudio y en los resultados obtenidos se concluye que para nuestra tarea concreta, con nuestro corpus y dataset de evaluación, la herramienta de generación Word2Vec[5] supera a FastText[7] y, en cuanto a los parámetros de entrenamiento, son muy favorables ventanas de contexto local pequeñas (2 palabras como ventana óptima) y gran cantidad de dimensiones en el espacio vectorial del embedding final (de 400 a 500). Una vez más, estos resultados son para nuestro estudio concreto y no se pretende demostrar nada de forma genérica con ellos.

Extended Abstract

In the domain of computer science, artificial intelligence has played an important role since its rise in the 1960s to the present day, introducing several fields to the domain, such as the study of natural language, bringing human language to machines. Such is the case of Natural Language Processing as a way of studying efficient computational mechanisms for the analysis, understanding and use of natural language by computers using neural networks. Such is the interest in these fields that Word Embeddings are born as tools to process language in such a way that the machine can analyse and understand it by training them from a large corpus of free text [1][2].

Within this field, there are several ways of generating and creating models that achieve this effect, as Word Embedding has been developed from the 1990s to the present day. Specifically, there are two main approaches: models based on global matrix factorisation[4] and models based on local context window[5]. Global matrix factorisation assumes that words that are close in meaning will appear in similar text fragments (the distribution hypothesis). These are algorithms such as LSA (Latent Semantic Analysis), pLSA (probabilistic Latent Semantic Analysis), LDA (Latent Dirichlet Allocation) and pLDA (probabilistic Latent Dirichlet Allocation). On the other hand, models based on local context window are based on the fact that a word is defined by its closest words[8]. These are the CBOW and Skip-Gram algorithms. In either case, all words in a vocabulary contribute to the definition of a given word.

The generation of an embedding can be divided into two subproblems: first, getting your model to represent the words in vectors, and second, training the model to adjust this representation to the problem at hand. The first problem is already solved thanks to pre-trained models in this task, thus making use of the concept of learning transfer in machine learning. These pre-trained models are what we know today as embedding generation tools, such as Word2Vec, GLOVE, FastText and others. We must then tackle the second subproblem, training the model.

Once the domain and the task to be solved by our embedding is clear, a large amount of text representative of the domain that our model will study must be collected in order to be trained. This large amount of text is known as corpus and it will be used to train the neural network. Once a large corpus representative of a domain has been obtained, it must be preprocessed. The idea of preprocessing is to prepare our text to enter the algorithm directly. The preprocessing of a corpus consists of two steps, first, to leave in the text only the tokens that we want to be analysed by our model and second, to prepare the corpus as input for our pretrained model. Once the corpus has been preprocessed, it can be trained. When training a model there are many parameters that can affect the training time and the quality of the final model. Parameters such as minimum word occurrence, size of model vectors, window size or ignored words.

Once our model is trained, we can evaluate it. The way we evaluate our model depends directly on the task we are trying to solve.

There are various applications or 'tasks' for which these models could be trained[3]. Examples of these are taste analysis, text generation, language translation or analogies. The latter is the one we will study in this work, because in the advent of the first embedding generation tools it was observed how the distances between words of certain vocabulary, when represented in the embedding vector space, were directly dependent on the relationship between those words, being very short between very similar words and vice versa. For example, we expect the vector associated with the term 'cat' to be closer to the vector 'dog' than to the vector 'potato'. This fact makes possible the existence of analogies in embedding, forming parallelograms in the vector space, the vertices of which are the vectors of the four words of the analogy. For example, we expect to find between the vectors 'king' and 'queen' a distance very similar to that which could be seen between the vectors 'man' and 'woman'.

This work does not seek to study this fact, but to take advantage of it to implement a question/answer system based on analogies from the Spanish legal domain, as it would be very interesting to check whether embedding would be able to capture the relationships between more complex legal terms. To this end, (i) a large representative corpus of this domain has been obtained, (ii) a study of the accuracy of the embedding models generated depending on the generation tool and its training parameters has been carried out in order to obtain the best possible embedding and (iii) a web interface has been created for the use and enjoyment of this model. All the code implemented in this work is available under the GitHub repository *tfg-repo* of my user *WhiteSockLoafer*[1] under an MIT license that grants the use, modification and distribution of the code to anyone in any form.

To extract the corpus, a Python script has been implemented based on the library *Scrapy*[9] with which we can search the website of the State Agency of the Official State Gazette following links to the results and downloading the body of the HTML response of the results. The final corpus is approximately 480 MB of plain text files which makes approximately 18000 BOE articles with a final vocabulary of current terms and popular ones (from 2021 to 2015 except for some exceptions such as the civil code).

Then, going into the study of the accuracy of the models, the corpus was preprocessed to avoid inappropriate characters or terms that would cause noise in our training. Specifically, our preprocessing consists of (i) eliminating all characters that are not letters with or without accents or spaces, (ii) putting all the text in lowercase so as not to differentiate words that are really the same, (iii) eliminating words smaller than two characters as they produce some unnecessary noise and (iv) leaving only sentences with more than two words. Moreover, as having the whole corpus (480 MB in size, remember)

[1] `https://github.com/WhiteSockLoafer/tfg-repo`

in a variable would be spending unnecessary amounts of memory at runtime, a class has been designed where iteration over the corpus is 'memory-friendly', not having to bring the whole corpus into memory and preprocessing small parts of it at the same time as they are used.

The next step was to create a Python script with the help of the Gensim[10] library that trains, saves and evaluates the generated models with the desired training tools and parameters, in particular, we evaluated the models created with Word2Vec (since it is well demonstrated that this pre-trained model generates more accurate embeddings for generic natural language processing tasks such as ours) and FastText (since the tool makes use of n-grams to train the model and it would be interesting to see if this method could help us in our study with analogies), context windows of 2, 5 and 10 words and, finally, with final dimensions of 200, 350 and 500. Thus, exactly 18 models were trained and evaluated, which took 5 hours, 38 minutes and 25 seconds to finish.

When evaluating the models, a different evaluation model than the normal one was proposed, which is much more restrictive. This is because when evaluating a certain analogy, the first three words of the analogy are introduced in the model and we expect the fourth word as an answer and, as the models are not exact, they answer with a list of words closer to the result: while the usual models only take into account the appearance or not of the expected word in the list, our evaluation model also takes into account the position that this word has in the list, not always being the first one. In addition, a dataset of 40 analogies from the legal field was created to evaluate the models.

The output of the created script is a folder with the results of the evaluation in a JSON file, a small comparative graph of the accuracies and a subfolder with the saved embedding models.

Finally, a web interface was designed to complete the question/answer system. At the beginning and being very optimistic, this interface could be addressed to people in the legal field in Spain, from a student to any person already experienced in the domain, such as a teacher. This is why it was decided to make a web interface, as it is the easiest way for a person to make use of our trained model. The interface has been developed in *DJANGO*, because (i) the whole project is developed in Python, as is django, (ii) web interfaces usually involve a frontend and a backend container, although our frontend will be very simple and it is not worth separating the interface into two containers, django can be used to unify these two concepts for cases like this and (iii) we already had experience with this technology, which makes the task much easier considering that the main objective of the project was not this interface.

The training and evaluation of the models was then carried out with:

1. Hardware of 16 GB RAM, 8 cores at 2 GHz on average
2. Corpus of 17,978 files, exactly 482.7 MB of plaintext
3. 18 trained models
    3.1. FastText and Word2Vec tools
    3.2. Window sizes 2, 5 and 10
    3.3. Model dimensions 200, 350 and 500
4. Evaluation dataset of 40 analogies
5. Number of words closest to the solution vector topn = 10

Analysing the results: First of all, the training times. It is clearly visible that FastText is always in the worst position. It takes in the worst case up to 10 times longer than Word2Vec to complete its training. This is probably due to the n-grams, as the combinations of n-grams in a vocabulary are larger than the number of words in the vocabulary, requiring significantly more training time as the number of operations on the model and its vocabulary will always be larger. On the other hand, the n-grams offer great advantages such as, for example, a better relation between compound words or the approximation of the vectors of tokens that do not exist in the model and are consulted because, although the vector of the word does not exist, the vectors of its n-grams probably do.

The results also show how Word2Vec is always more accurate than FastText. This is true only for the dataset we have created, because in this dataset there are no tokens that do not belong to the corpus or to the vocabulary of the model itself. It is in these cases that Word2Vec could not be applied at all and FastText could obtain an approximation to the result word thanks to the n-grams, which would then play an important role in the accuracy of the model. It could also be said that these technologies are very different and that the results are not very representative due to the size of the corpus which could be somewhat small.

It can also be seen that small windows are much more favourable than large windows, this is directly related to the corpus. As it appears in works such as Maryam Fanaeepour et al. and Pierre Lison and Andrei Kutuzov's[11] and Andrei Kutuzov's[12], there is no definitive window size for all models as the idea is to obtain the smallest window that perfectly captures the context and this depends directly on the constructions of each of the sentences in the corpus. The best value of this parameter depends on each corpus. By obtaining the best results in small windows it could be concluded that in the corpus the semantics of a word is not related to distant words. The context is very small. This is possible if the text is very dense in different concepts. Short, concise sentences rather

than a long narrative about a single concept. In the BOE, sections are short and deal with many topics, laws, directives. Usually one section is not related to the next.

As for the number of dimensions of the model, this parameter is also closely related to the corpus. As we see that more dimensions make our model more accurate in most cases, we understand that the corpus is representative of the domain. The fact that between 350 and 500 dimensions the accuracy does not change much could be due to the fact that above these numbers the models obtain an accuracy of the corpus content according to semantic relations between words and it could be that by increasing the number of dimensions, the models would not be able to represent better these relations, although this is not proven in this study.

Commenting on future avenues, this work was really focused on obtaining the best possible embedding to create a system of questions/answers for use, although the results obtained when evaluating the models could be quite interesting for a study of embeddings on obtaining the optimal training parameters in the task of analogies, this study being a possible avenue. For this, practically all the training and evaluation code could be reused, as the whole project was designed to be reused in any field, not only in the legal field or with a corpus obtained with our extractor. For this study I would try to obtain more graphs of the models to, for example, see how the window size affects the accuracy with some fixed embedding dimension or, also, this same but with the technologies used in the study.

It should be noted that, while the dataset is very representative, it is somewhat short. It would be advisable to increase its size to ensure that the results are really those obtained.

Another future way could be to improve the web interface and add some improvements such as a help button or an interactive tutorial because searching for a certain term by analogy is not something we are used to as we usually search for terms based on their similarity with others, using simple relations.

Índice

1. Introducción

En el campo de las ciencias de computación, la inteligencia artificial ha jugado un importante rol desde su auge en la época de los 60 hasta el día de hoy, introduciendo esta varios campos al ámbito como es el caso del estudio del lenguaje natural, trayendo a las máquinas el lenguaje humano. Tal es el caso que que surge el Procesamiento del Lenguaje Natural como vía de estudio de mecanismos computacionales eficaces para el análisis, comprensión y uso del lenguaje natural por parte de las computadoras. Uno de los mecanismos investigados y actualmente líderes en el desarrollo de estos sistemas son los Word Embeddings, basados en el estudio de grandes corpus de texto para obtener una representación de todas las palabras de cierto vocabulario en un espacio vectorial[1][2]. Existen, dentro de este ámbito, varias formas de generación y creación de modelos que consigan este efecto pues los Word Embedding se han ido desarrollando desde la época de los 90 hasta el día de hoy.

Al representar las palabras de un dominio en un espacio vectorial, es obvio pensar que estas no son colocadas arbitrariamente, sino que su posición en el espacio debe ser representativa del término. En los Word Embeddings, las distancias que hay entre las palabras del vocabulario que representan son completamente dependientes de la relación que guardan entre ellas, teniendo así los vectores de palabras muy similares muy cercanos entre sí y viceversa[8]. Llevado esto a un extremo, los modelos embedding son capaces de representar también relaciones de analogías entre cuatro palabras formando con sus vectores paralelogramos casi perfectos en el espacio[4][6], añadiendo este hecho al estudio del Procesamiento del Lenguaje Natural y sirviéndonos esto como una manera de evaluar los modelos generados para esta tarea[3][11].

En este trabajo buscamos utilizar las analogías en los embedding para crear un sistema basado en preguntas/respuestas que, con la ayuda de un embedding, sea capaz de predecir una analogía que le propongamos. Para ello la idea es centrarse en un dominio específico pues hacer un embedding del lenguaje general en español, además de costoso, está más que estudiado. Tras elegir un dominio, tendremos que obtener un gran corpus de texto representativo con el que poder entrenar un modelo final y preprocesarlo para evitar ruido a la hora de entrenar nuestro embedding. Tras esto, la idea es realizar un estudio sobre las técnicas y métodos que hay de generar embeddings para conseguir así el mejor modelo que se adapte a nuestro objetivo. Por último, se debe pensar en cómo realizar este sistema de preguntas/respuestas para hacer uso del modelo de tal manera que no sea costosa de implementar y que sea cómoda de usar.

2. Estado del arte

2.1. Historia de Word Embeddings

El procesamiento del lenguaje natural (PLN) es un campo de conocimiento de la Inteligencia Artificial que se ocupa de investigar la manera de traer a las máquinas el lenguaje natural humano (i.e. idiomas). Ciertas ramas del PLN (incluido los embedding) hacen uso de las redes neuronales para llevar a cabo su tarea. Las redes neuronales están originalmente diseñadas para trabajar con datos numéricos y, por ello, las palabras del lenguaje natural deben representarse como tal.

Word Embeddings es el nombre que recibe el conjunto de modelos y técnicas de procesamiento del lenguaje natural donde las palabras o frases se representan como vectores de números reales. Esta representación permiten identificar similitudes entre palabras en función de la co-ocurrencia entre ellas, del lugar que ocupan en cierto texto o basándonos en el resto de palabras que la acompañan[13]; Todas las palabras de un vocabulario aportan en la definición de una palabra determinada, consiguiendo así relacionar las palabras de un vocabulario usando las distancias que hay entre sus posiciones en el espacio vectorial que los embedding representan. Es por esto que palabras sintáctica y semánticamente similares tendrían una distancia corta entre sus vectores y viceversa. Por ejemplo, esperamos que el vector asociado al término 'gato' esté más cerca del vector 'perro' que del vector 'patata'. Yendo más allá, los embedding consiguen representar relaciones ciertamente complejas entre las palabras, i.e. las **analogías**, relaciones establecidas entre cuatro términos dónde, por ejemplo, esperamos encontrar entre los vectores 'rey' y 'reina' una distancia muy parecida a la que pudiera verse entre los vectores 'hombre' y 'mujer' (**Figura 1**).

En la época de los 90 el modelo de representación del lenguaje natural predominante eran vectores en un espacio discreto que daban lugar a grandes matrices dispersas. El término Word Embeddings no fue concebido hasta 2003 por Y. Bengio et al. en '*A neural probabilistic language model*'[1], donde crearon un modelo probabilístico neuronal con esta nueva representación de las palabras. Uno de los primeros modelos más populares nació a manos de R. Collobert y J. Weston [2] en 2008, en cuya presentación muestran su utilidad al uso en numerosas tareas presentes en el campo como reconocimiento de entidades nombradas, etiquetado gramatical o etiquetado semántico. Es en 2013 cuando Word Embeddings empezó a popularizase globalmente con la llegada de Word2Vec [5], una herramienta creada por Tomas Mikolov et al. bajo el dominio de Google que ofrecía la posibilidad de entrenar y usar modelos propios. Un año después, Pennington et al. introdujeron GLOVE [4], una nueva herramienta cuya implementación difería con respecto a la de Word2Vec en la forma en la se definían la semántica de cada palabra. Estas dos implementaciones dieron lugar a dos grandes vertientes diferenciadas por la

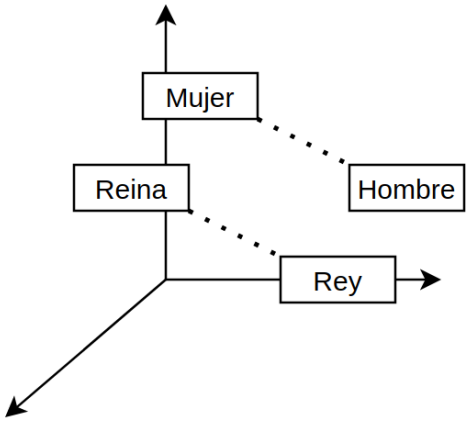

Figura 1: Ejemplo que representa distancias similares entre palabras semánticamente análogas

forma de entrenar a sus modelos: Aprendizaje por métodos de factorización global de matrices como LSA, LDA, pLSA y sLDA, y métodos de ventana de contexto local como skip-gram o CBOW. A partir de estas dos herramientas fue cuando Word Embeddings se convirtió en una de las corrientes principales dentro del Procesamiento del Lenguaje Natural, llevando así su uso a aplicaciones como:

- Análisis de sentimientos y gustos: Un modelo entrenado con opiniones sobre cierto producto podría predecir si cierta opinión nueva tiene una connotación positiva o negativa. De la misma manera podríamos clasificar textos en base a los sentimientos que contienen y entrenar un modelo con ellos para que sea capaz de diferenciarlos.
- Generación de texto: Mediante el uso de redes recurrentes es posible generar texto de forma automática basado en otros textos con los que se entrenó el modelo. Pueden generarse letras de canciones, guiones de películas, libros...
- Traducción de idiomas: El traductor de Google hace uso del modelo seq2seq [14] que hace uso de corpus paralelos para su entreno, i.e. textos idénticos en distintos idiomas.
- Recomendaciones basadas en tus gustos: Estos sistemas hacen uso de la representación vectorial para crear un vector basado en el contenido que el usuario haya consumido para recomendar posteriormente el contenido cuyo vector se asemeje considerablemente al del usuario.

2.2. Qué es un Embedding

En la sección anterior se ha comentado acerca de los modelos, algoritmos de aprendizaje, redes neuronales y demás términos relacionados con los Word Embeddings en el ámbito del procesamiento del lenguaje natural desde un punto de vista genérico e introductorio. En esta sección se muestra explicación extendida sobre este campo y sus términos, explicando así cómo funcionan y los problemas que resuelven. El vídeo sobre el procesamiento del lenguaje natural del canal de YouTube de ciencia y tecnología de *Dot CSV* [15] ha sido de gran ayuda para redactar este aparatado.

Intentar traducir a reglas formales todas nuestras formas de comunicación es una tarea no solo difícil sino tediosa, pues nuestro lenguaje se encuentra en constante evolución. Es precisamente por este tipo de tareas y problemas que surgió el *machine learning* como algoritmos que tratan de automatizar estos procesos de aprendizaje. Los Word Embeddings se apoyan en el machine learning para analizar grandes corpus de texto haciendo uso de redes neuronales, tomando como entrada las palabras de estos corpus y obteniendo un modelo bien formado capaz de relacionar y definir estas palabras o términos. Las redes neuronales trabajan con datos numéricos y es por esto que se deben representar las palabras como tal. En los embedding actuales las palabras pueden ser representadas en un modelo de varias maneras, sea carácter a carácter, palabra a palabra o en subpalabras (**Figura 2**), referenciando así cada término del corpus como **token** y al proceso de división, **tokenización**. Puesto que en este proyecto se trabaja con palabras y relaciones entre ellas, nos centraremos en la división por palabra y en representarlas numéricamente mediante vectores.

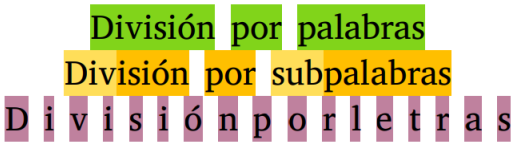

Figura 2: Tokenización de palabras

La razón por la que las palabras se representan mediante vectores y no simples números es porque todos los números vienen estrechamente relacionados, relación que las redes neuronales utilizan para trabajar y que en un principio las palabras no poseen. Cuando decimos que los números poseen cierta relación que las palabras no es porque, por ejemplo, 4 es el doble de 2 o 150 es 10 veces 15, mientras que las palabras de un lenguaje natural no se acogen a estas relaciones pues en un principio no tendría mucho sentido dividir 'árbol' entre 'mesa'. La idea de asignarle a cada palabra un vector es que, aunque las palabras no posean esa relación tan estrecha y directa como los números, sí que guardan cierta relación semántica que podría representarse en un espacio vectorial,

apareciendo los vectores de dos palabras con cierta similaridad muy cerca el uno del otro (**Figura 3**).

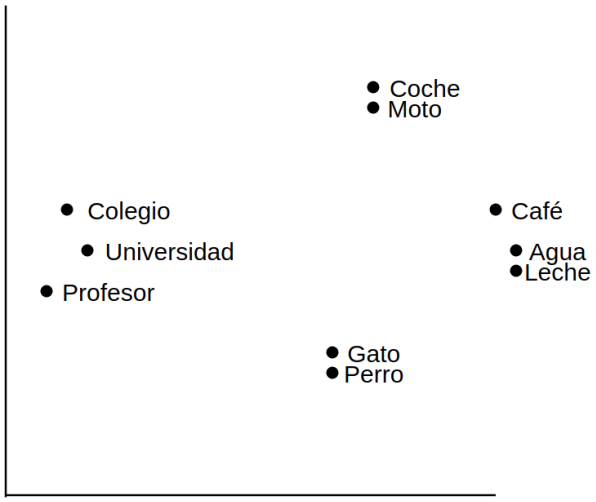

Figura 3: Representación de la relación entre palabras

En la sencilla representación One-Hot Encoding los vectores asignados a cada palabra tienen tantas dimensiones como palabras distintas en nuestro vocabulario, es decir, si tuviéramos el corpus 'The cat sat', nuestro vocabulario sería de 3 palabras y podríamos vectorizarlo en un espacio tridimensional como el de la **Figura 4**, teniendo todos los vectores tamaño 1 y equidistantes entre sí. Esto en un vocabulario de por ejemplo 20 mil palabras daría lugar a vectores de 20 mil dimensiones todas a cero menos en la dimensión que la palabra representa, cuyo valor estaría a 1. Obviamente esta representación daría lugar a enormes matrices de numerosos elementos la mayoría a 0, como si estableciéramos un diccionario entero por cada palabra, un diccionario que solo definiera la palabra que representa.

De forma natural y habitual, las redes neuronales comprimen y ordenan los datos de manera inteligente para poder resolver de la mejor manera la tarea que se le haya encomendado, haciendo uso del concepto de reducción de la dimensionalidad del problema. En la **Figura 5** podemos ver una red neuronal de tres capas: La primera capa por dónde entran los datos tiene una dimensión n menor que m , la segunda dimensión m mayor que 1 y la de salida dimensión 1. Esta red neuronal conseguiría reducir la dimensionalidad del problema de n dimensiones que tienen los datos de entrada a 1 dimensión, independientemente de cual sea el problema que esté intentando resolver.

Esta misma reducción de la dimensionalidad podríamos aplicarla a nuestra representación de palabras a vectores en One-Hot Encoding. Volviendo al ejemplo del corpus con un vocabulario de 20 mil palabras, si introdujéramos nuestro vocabulario en una

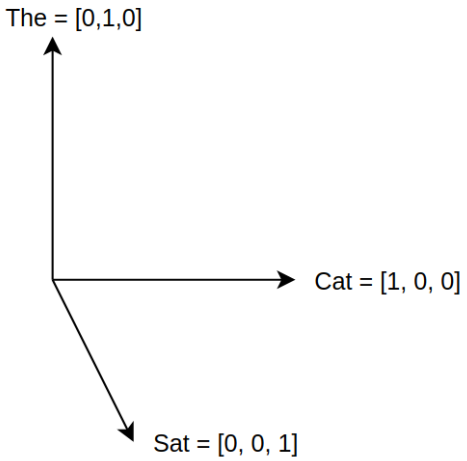

Figura 4: Espacio vectorial de un vocabulario de 3 palabras

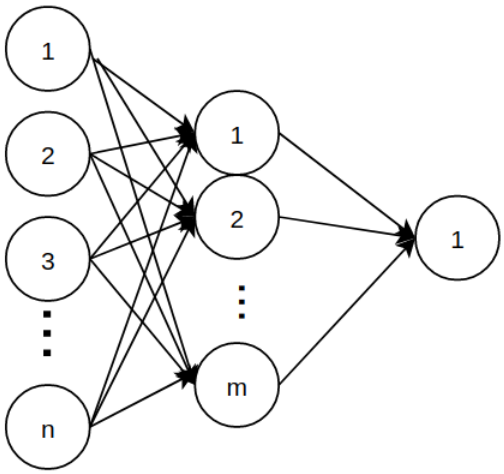

Figura 5: Ejemplo de red neuronal

red neuronal cuya capa de entrada fuera de 20 mil neuronas y la de salida de 100 neuronas, nuestra representación del vocabulario de 20 mil dimensiones se vería truncada a 100 dimensiones. Si bien el concepto es sencillo, la tarea no tanto, pues estaríamos pidiendo a nuestra red neuronal que encontrase la representación de 100 dimensiones que represente lo mejor posible nuestro espacio de 20 mil dimensiones. Es por esto que en un principio la red no conseguiría los resultados esperados y las palabras no estarían bien representadas en el nuevo espacio vectorial, necesitando así la red apoyarse en el concepto de entrenamiento.

Al empezar el entrenamiento, paso a paso la red conseguiría entender a comprimir y ordenar las palabras de la manera más óptima para resolver el problema que le hayamos

planteado. Poniendo un ejemplo los textos 'Odio los girasoles' y 'Me gustan mucho las orquídeas': si estuviéramos estudiando si cierto texto tuviera un sentimiento positivo o negativo la red debería ajustar sus parámetros poniendo más peso a la hora de elegir un resultado a palabras como 'gustan' y 'Odio'. Si marcamos la primera oración como positiva y la segunda como negativa y el corpus fuera lo suficientemente grande, la red encontraría que cada vez que la palabra 'Odio' aparece, el texto viene clasificado como negativo, teniendo esta palabra un gran peso en el resultado. Otras palabras como 'girasoles' u 'orquídeas' quedarían completamente en un segundo plano en este problema en concreto. La nueva representación que saldrá de la red viene estrechamente relacionada con el problema que se quiere resolver, encontrando la codificación de los datos de entrada que mejor le permita acercarse a la solución óptima.

Este proceso que ocurre en la primera capa de la red neuronal en el que se debe aprender a comprimir los vectores One-Hot Encoding a una representación compacta para resolver su tarea es lo que conocemos como **embedding**. Como bien se puede deducir, la creación de un embedding se divide en dos subproblemas: primero, hacer que tu modelo consiga representar las palabras en vectores y segundo, el entrenamiento del modelo para ajustar esta representación al problema que se nos presente. De forma análoga podríamos decir que si la red neuronal fuera una persona y quisiéramos que entendiera un texto científico, tendríamos que enseñar a esa persona primero a leer para poder después entender el texto.

En el campo del *machine learning* existe el concepto de *transfer learning*, concepto basado en que el conocimiento ganado en una red neuronal puede transferirse a otra y es por esto que podríamos apoyarnos en conocimientos previamente adoptados por otra red para agilizar nuestra tarea. En el caso de la generación de embeddings, puesto que el primer subproblema de generar una representación vectorial de un vocabulario es en su mayor parte común a todos los problemas que podamos encontrar en el mundo de los word embeddings, podría crearse un primer **modelo embedding preentrenado** que ya supiera realizar esta tarea común de la forma más genérica y universal posible para poder usarse en todas las generaciones de embedding y solo dejar el segundo subproblema (el entrenamiento del modelo) específico de cada tarea por resolver. Efectivamente estos modelos preentrenados ya existen y son precisamente las herramientas de generación de embeddings que ya se nombraron en el apartado anterior, Word2Vec y GLOVE entre las más populares. Estas herramientas no son más que una interfaz de funciones que rodean al modelo de generación preentrenado con el que podríamos crear nuestro propio embedding centrándonos solo en su entrenamiento. Volviendo a la analogía anterior, nuestra persona ya sabría leer, solo necesitaríamos que entendiera el corpus.

Para poder entender mejor los embedding, podríamos visualizar el espacio vectorial de cierto vocabulario ya entrenado para encontrar así como palabras similares se encontrarían cercanas unas de otras. Las herramientas actuales de generación de embedding

admiten como parámetro el número de dimensiones que nuestro embedding final tendrá, normalmente de 100 a 500 dimensiones pues bajar de 100 supondría perder demasiada precisión y subir de 500, en general, no aumenta demasiado la precisión de los modelos. El cerebro humano es incapaz de visualizar más de 3 dimensiones de espacio vectorial, si bien como ya se comentó, es posible reducir la dimensionalidad del espacio mediante las redes neuronales, con lo que podríamos reducir la dimensionalidad nuestro modelo de embedding de cientos de dimensiones a un espacio tridimensional para poder visualizarlo. Es el caso de la herramienta *Embedding Projector*[2] de TensorFlow [16] que nos permite subir y visualizar nuestro propio embedding además de cargar ciertos modelos populares ya entrenados, como el 10K de Word2Vec, un embedding de 10 mil palabras con vectores de 200 dimensiones (**Figura 6**).

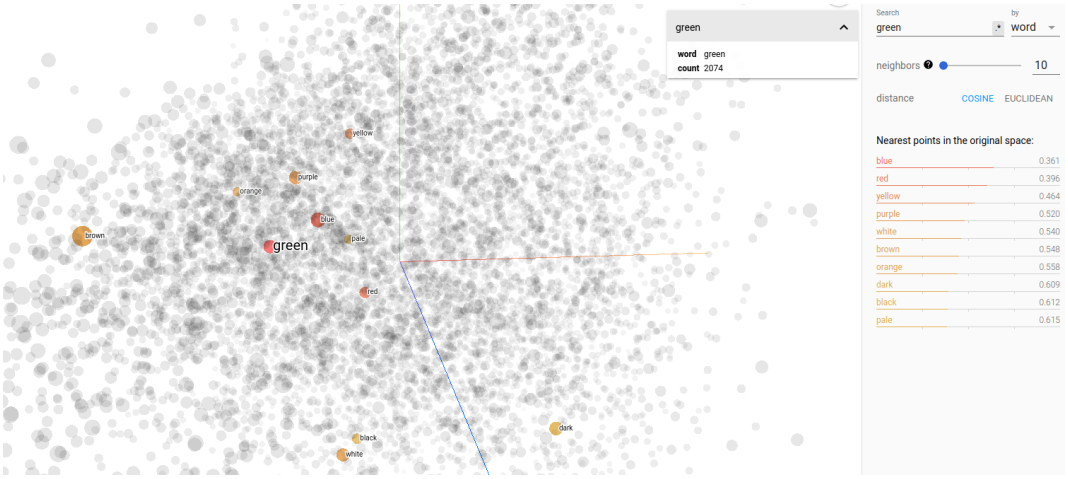

Figura 6: Visualizador de embeddings de TensorFlow

2.3. Algoritmos de aprendizaje

Puesto que la forma en la que los modelos se entrenan difiere entre las herramientas de generación de embeddings que existen actualmente, se procede a explicarlas de cara a escoger el modelo preentrenado que más se ajuste a nuestro estudio.

Como ya se explicó en la historia de los embedding, existen dos grandes vertientes diferenciadas en la forma en la que se define cierta palabra o termino: modelos basados en factorización global de matrices y basados en ventana de contexto local. En la factorización global de matrices se analizan las relaciones entre un conjunto de documentos y los términos que contienen asumiendo que las palabras que tienen un significado cercano aparecerán en fragmentos de texto similares (la hipótesis de distribución). El

[2] `https://projector.tensorflow.org/`

primer paso es generar nuestra matriz documento-palabra. Dados m documentos y n palabras en nuestro vocabulario, podemos construir una matriz A de m × n en la que cada fila representa un documento y cada columna representa una palabra. En la versión más simple de este modelo, cada entrada puede ser simplemente un recuento del número de veces que apareció la j-ésima palabra en el i-ésimo documento. Una vez que tengamos nuestra matriz de documentos y términos, podemos realizar una reducciones de dimensionalidad en ella utilizando técnicas de algebra lineal como SVD (Single Value Decomposition) o métodos probabilísticos. Dependiendo de como hagamos esta reducción diferenciamos entre los modelos LSA (Latent Semantic Analysis), pLSA (probabilistic LSA), LDA (Latent Dirichlet Allocation) y pLDA (probabilistic LDA).

Por otro lado, los modelos basados en ventana de contexto local se basan en que una palabra viene definida por sus palabras más cercanas. La 'ventana' no es más que el número de palabras a la izquierda y derecha en las que fijarnos a la hora de definir nuestro término (**Figura 7**). Estos modelos mueven la ventana a través de datos de texto y entrena una red neuronal de 1 capa oculta basada en la tarea sintética de, dada una palabra, devolvernos un conjunto de probabilidades predichas de palabras cercanas a la entrada. La representación de la ya nombrada One-Hot Encoding pasa a través de la capa 'oculta'. Si la capa oculta tiene 300 neuronas, esta red nos dará incrustaciones de palabras de 300 dimensiones.

En el modelo CBOW, dado el contexto de una supuesta palabra (i.e. las palabras que rodean nuestra palabra objetivo), intenta predecirla. Contrariamente, en el modelo Skip-gram dada una palabra, predice su contexto, es decir, las palabras cercanas. También hacen uso de One-Hot Encoding, siendo una vez más la capa 'oculta' la encargada de pasar esta representación a vectores con pesos.

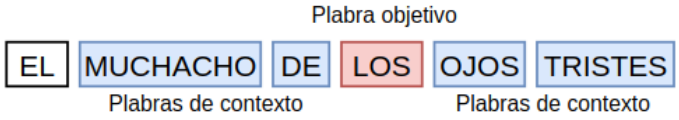

Figura 7: Ventana de contexto local

2.4. Generación de Embeddings

A día de hoy contamos con numerosas herramientas de generación de embeddings que ya cuentan con un modelo preentrenado que resuelva la primera parte del problema, representar las palabras de un vocabulario en un espacio vectorial genérico y desvirtuado. La idea es que, a partir de estos modelos preentrenados, dotemos a los vectores de cada palabra de un valor que caracterice su su significado respecto al del resto de

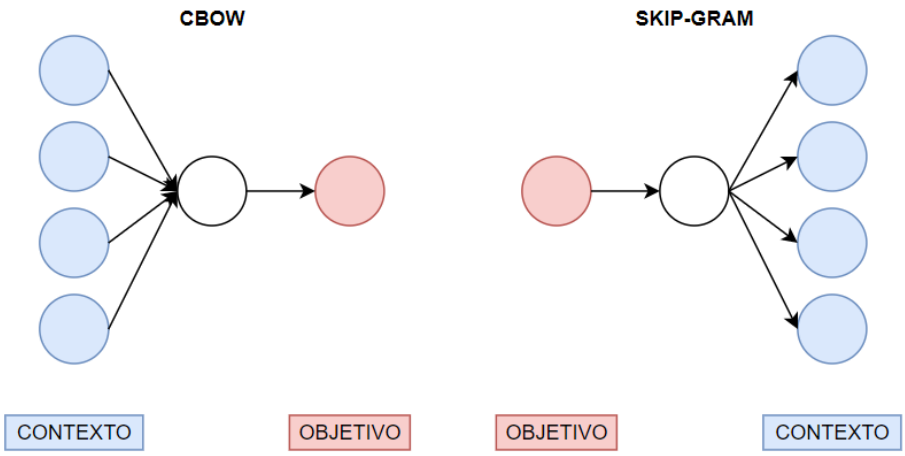

Figura 8: Modelos CBOW y Skip-gram

palabras en este espacio vectorial, entrenando así nuestro modelo para resolver nuestro problema específico. En este apartado se procede a explicar cómo generar nuestro propio embedding.

Como ya sabemos, la generación de un embedding puede dividirse en dos subproblemas: primero, hacer que tu modelo consiga representar las palabras en vectores y segundo, el entrenamiento del modelo para ajustar esta representación al problema que se nos presente. Sabemos que el primer problema viene ya resuelto gracias a modelos preentrenados en esta tarea, haciendo uso así del concepto de *learning transfer* en machine learning. Debemos entonces abordar el segundo subproblema: entrenar el modelo.

Para entrenar el modelo primero debemos tener claro qué tarea van a resolver. Como vimos en la historia de los Word Embedding, a día de hoy podemos hacer uso de estos para un gran número de tareas como analizar sentimientos, generar texto, traducir palabras y demás. Una vez se tenga claro el dominio y la tarea que resolverá nuestro embedding, se debe colectar una gran cantidad de texto representativo del dominio que nuestro modelo estudiará para así entrenarse. Esta gran cantidad de texto se le conoce como **corpus** y será con este con el que entrenaremos la red neuronal y, en algunos casos, la evaluaremos, separando entonces los casos de entrenamiento y los casos de evaluación. Dependiendo del problema, el corpus puede tomar distintas formas. Si, por ejemplo, nuestro problema fuera analizar si cierta review de una pelicula tuviera una connotación sentimental negativa o positiva, obtendríamos un gran número de reviews y deberíamos separar las reviews positivas de las negativas y, después, el conjunto de entrenamiento de el de evaluación, pues si el conjunto de evaluación apareciera también

en el de entrenamiento el modelo acertaría prácticamente siempre con ese conjunto. Otro ejemplo sería la generación de texto, donde realmente no interesa clasificar el corpus en subcorpus y este podría ser no más que un fichero de texto plano de gran tamaño.

El dominio debe ser relevante, pues no de todos los dominios se puede obtener un gran corpus con el que trabajar. Tampoco todos los dominios tienen un carácter evaluable con el que podamos más tarde determinar la precisión de nuestro modelo entrenado. A no ser que estuviéramos resolviendo una tarea de traducción, para facilitar el trabajo conviene recopilar el corpus en un idioma determinado, pues mezclar idiomas tendría dificultad añadida por el hecho de que un término o concepto tendría dos representaciones y no solo deberíamos asegurarnos de que las relaciones entre términos análogos fueran representativas sino también asegurarnos de que las palabras y sus traducciones tuviesen también distancias representativas. Ejemplos de corpus populares podrían ser el *Large Movie Review Dataset*[3] de la Universidad de Standford o Wikimedia Dumps[4] de Wikipedia (directorio con dumps y backups de varios años).

Una vez obtenido un gran corpus representativo de un dominio, se debe **preprocesar** este. La idea del preprocesamiento consiste en preparar nuestro texto para entrar en el algoritmo directamente. El preprocesamiento de un corpus consta de dos pasos, primero dejar en el texto solo los token que queramos que se analicen por nuestro modelo y segundo, preparar el corpus como entrada de nuestro modelo preentrenado. En la primera parte del preprocesamiento deben eliminarse del corpus toda clase de ruido que vaya a entorpecer nuestro entrenamiento, como podría ser:

1. Eliminación de símbolos que no aparecieran en el abecedario como puntos, comas, exclamaciones...
2. Eliminar palabras de una letra pues no son variables representativas y producen cierto ruido en los modelos.
3. Poner todas las letras en minúscula para que, por ejemplo, el algoritmo no categorice las palabras 'mano' y 'Mano' como dos palabras distintas...

La calidad del preprocesamiento del corpus está estrechamente ligada a la precisión que nuestro modelo final pueda tener. Una vez terminado este primer paso se debe preparar el corpus como entrada a nuestro modelo preentrenado. Como ya se comentó, los modelos preentrenados son las herramientas de generación de embeddings y cada una de estas proporcionan interfaces distintas al modelo. Debemos tener en cuenta como espera la herramienta recibir el corpus: Algunas herramientas piden una lista de

[3] `http://ai.stanford.edu/~amaas/data/sentiment/`
[4] `https://dumps.wikimedia.org/`

listas de tokens, siendo las listas de tokens las frases de nuestro corpus y la lista que envuelve nuestras frases el corpus en sí. Otras herramientas piden un directorio con un fichero por cada item con el que entrenar o evaluar el modelo. Si, por ejemplo, estuviéramos usando la herramienta de clasificación de texto con embeddings FastText y dentro del directorio dónde está nuestro corpus los ficheros vienen clasificados en directorios distintos estaríamos preclasificando el corpus para llevar a cabo un entrenamiento supervisado de nuestro modelo. En ciertos problemas que los embedding intentan resolver, la precisión puede medirse directamente con el corpus obtenido. Si es el caso, se debe separar el corpus en un gran conjunto de entrenamiento que nuestro modelo estudiará y otro pequeño subconjunto de evaluación con el que evaluar la precisión del modelo final. Interesa separar estos conjuntos pues si el conjunto de evaluación apareciera en el entrenamiento rara vez no acertaría el modelo sobre dicho conjunto al momento de su evaluación.

Una vez preprocesado el corpus, este puede ser objeto de un **entrenamiento**. A la hora de entrenar un modelo existen muchos parámetros que pueden afectar al tiempo de entrenamiento y la calidad del modelo final. Dependiendo de la herramienta que usemos y su modelo preentrenado, podremos ajustar los distintos parámetros a nuestro gusto. Posibles parámetros más importantes que vayamos a encontrar en un modelo entrenado con contexto de ventana local serían:

- **Aparición mínima de palabras**, teniendo un contador del mínimo número de apariciones que una palabra puede tener para que cuente como token de nuestro vocabulario, lo que nos ayuda a quitarnos cierta basura y ruido provocado por un preprocesamiento no tan preciso.
- **Dimensiones de los vectores**, siendo este número la cantidad de neuronas que encontraríamos en la capa 'oculta' de nuestra red neuronal y las dimensiones de nuestra representación. Gran tamaño requiere más tiempo y datos de entrenamiento. No siempre más dimensiones hacen un modelo más preciso, se corre el riesgo de que el modelo no sea significativo y los resultados empeoren. Más dimensiones hacen que el modelo represente con más precisión el contenido del corpus pero no las relaciones semánticas entre las palabras. Valores razonables podrían ser de los 100 a los 600 como mucho.
- **Tamaño de la ventana**, en el caso de usar un modelo basado en ventana de contexto local, pudiendo elegir el número de palabras cercanas a nuestra palabra objetivo (a la izquierda y derecha) que nuestro algoritmo debe de tener en cuenta a la hora de entrenar el embedding. Mayor ventana equivale a mas tiempo de entrenamiento, siendo según Levy y Goldberg[8] ventanas grandes ayudan a la hora de capturar tópicos e ideas de los documentos, mientras ventanas pequeñas se concentran en la palabra en sí y ayudan a encontrar mas similitudes entre palabras y sinónimos.

- **Palabras ignoradas**, siendo este argumento un conjunto de palabras que sabemos no nos interesa tener en cuenta pues serían ruido en nuestro modelo para la tarea que queremos llevar a cabo.

Una vez entrenado nuestro modelo, podemos **evaluarlo**. La forma de evaluar nuestro modelo depende directamente de la tarea que estemos intentando resolver. A día de hoy, herramientas como word2vec cuentan con datasets de evaluación ya preparados para modelos que intentan resolver un análisis del lenguaje de manera genérica, intentando capturar el mayor número de palabras y sus relaciones posibles. En tareas de, por ejemplo, clasificación del texto, la manera ideal de evaluar un modelo final es separando cierta cantidad de corpus ya clasificado en un pequeño conjunto de evaluación, dejando al algoritmo clasificar el texto y comprobando si la clasificación fue o no correcta. Otro caso es el de traducción, dónde la forma de evaluación es bien sencilla pues es un ámbito mayormente preciso. Puede evaluarse este con datasets de texto donde las palabras y sus tradcciones vayan una al lado de la otra.

2.5. Analogías en Vectores

Como bien se comentó en apartados anteriores, los word embedding son capaces de captar similitudes semánticas y sintácticas entre palabras. La similitud de dos palabras del vocabulario se miden en su representación usando comúnmente la similitud coseno de ellos[17]:

$$\cos(x, y) = \cos\theta = \frac{\langle x, y\rangle}{\|x\|\,\|y\|}$$

dónde θ es el ángulo formado entre x e y. El rango de similitud coseno es desde -1 (vectores opuestos) a 1 (vectores paralelos). La mayoría de veces se normalizan los vectores antes de usarse, por lo que tenemos que $\cos(x, y) = \langle x, y\rangle$. Palabras semánticamente similares deberían aparecer con un valor muy cercano a 1 en su similitud coseno.

En nuestro lenguaje, una similitud entre dos palabras no es única de estas, sino que otras dos palabras pueden compartir esa misma relación entre sí. Por ejemplo, 'sucio' y 'limpio' son contrarios, 'nuevo' y 'antiguo' también lo son. Los modelos de word embedding estudiados consiguen captar estas similitudes en modelos entrenados sin supervisión, poniendo aproximadamente las palabras 'sucio' y 'limpio' a la misma distancia que 'nuevo' y 'antiguo' (**Figura 9**).

Yendo más allá, Pennington et al. [4] remarcaron como los Word Embeddings conseguían captar muy bien **analogías** de nuestro vocabulario haciendo uso de estas similitudes. Las analogías no son más que relaciones horizontales y verticales entre cuatro palabras, repitiéndose estas relaciones entre ellas pero no de forma cruzada. Por ejemplo, en la analogía clásica '*Hombre* es a *Rey* como *Mujer* es a *Reina*', tenemos una

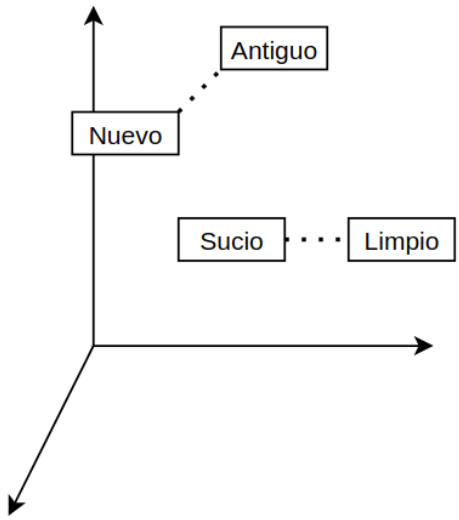

Figura 9: Mismas distancias pues es la misma relación

relación de 'género distinto' entre hombre y mujer y también entre rey y reina, además de una relación de 'clase social superior' entre hombre y rey y también entre mujer y reina. Puede verse que no existen esas mismas relaciones directas entre hombre y reina ni entre mujer y rey (no hay relaciones cruzadas) (**Figura 1**).

La forma intrigante que los Word Embeddings creados a partir de redes neuronales tienen de representar estas analogías a menudo se pueden resolver simplemente sumando y restando incrustaciones de palabras.Volviendo a la analogía anterior, '*Hombre* es a *Rey* como *Mujer* es a *Reina*' pude resolverse calculando el vector mas cercano a *Rey* - *Hombre* + *Mujer*, resultando ser *Reina*, sugiriendo:

$$W_{REY} - W_{HOMBRE} + W_{MUJER} \approx W_{REINA}$$

Siendo W_X el vector asociado a la palabra x. En términos geométricos, los las analogías de palabras en word embeddings forman aproximadamente paralelogramos **Figura 10**.

Si bien esto se ajusta a nuestra intuición, el fenómeno no deja de ser intrigante pues los word embeddings no se empezaron entrenan para dar lugar a estas analogías. Este hecho es la causa directa de querer representar el vocabulario de cierto corpus en un espacio vectorial. De hecho, en la práctica no se forman estos paralelogramos perfectamente. En la **Figura 11** puede verse como la representación de *reina* no termina de ajustarse al vértice del supuesto paralelogramo ideal, si bien es la palabra más cercana a este. Por ello, a la hora de mostrar el resultado de la analogía en la práctica, se muestra una lista las palabras más cercanas a dicho vértice.

Se expone una relación de 'realeza' entre las palabras 'rey' y 'hombre', siendo 'rey' la versión 'real' (de la corona) de 'hombre' y 'reina' la de 'mujer'. Parecen los embedding

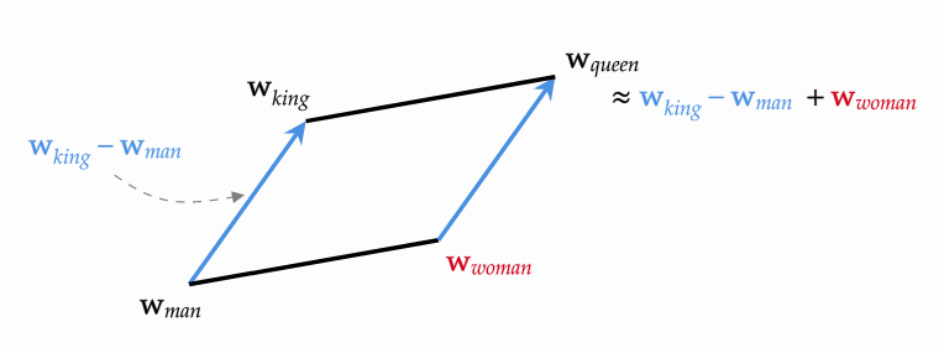

Figura 10: Carl Allen[18]: Analogías en el espacio vectorial

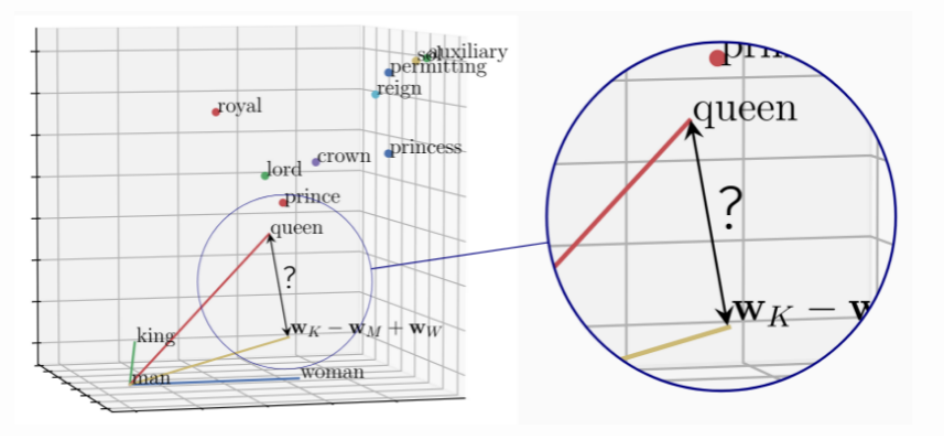

Figura 11: Carl Allen[18]: Analogías en la práctica

saber que, efectivamente entre las palabras existen estas relaciones y que ciertas relaciones pueden producirse entre términos completamente distintos, si bien no entienden ni tienen forma de entender estas relaciones ni las palabras. Esto significa que, siempre el corpus con el que se haya entrenado el embedding sea lo suficientemente representativo, el modelo podría captar varios tipos de relaciones independientemente de su naturaleza, desde la contrariedad, pluralidad u otros tipos de relaciones sencillas hasta pertenencia (uno es la unidad y el otro es el conjunto de unidades, como *'piara' es a 'cerdo' lo que 'banco' es a 'pez'*) o grado semántico (uno es la exageración o la disminución del otro, como feo y horroroso).

2.6. Herramientas actuales

A día de hoy existen un gran número de herramientas de trabajo para la generación y uso de vectores de palabras. En esta sección analizaremos las aproximaciones más populares de generación de word embeddings y sus métodos para posteriormente elegir

una o más con la que poder realizar nuestro estudio de analogías.

- Word2Vec [5]: Esta herramienta hace uso de un modelo predictivo basado en métodos de ventana de contexto local. En concreto, implementa los modelos CBOW y Skip-gram. Las capas internas de la red neuronal codifican la representación de la palabra objetivo, los word embeddings. Estos modelos entrenan sus vectores para mejorar su capacidad predictiva.
- GLOVE [14]: Mientras Word2Vec es un modelo predictivo, GLOVE se basa en el conteo de las palabras y su co-ocurrencia, haciendo uso de los modelos basados en factorización global de matrices. Primero construyen una matriz grande de información de co-ocurrencia, es decir, para cada 'palabra' (las filas), cuenta la frecuencia con la que vemos esta palabra en algún 'contexto' (las columnas) en un gran corpus. El número de 'contextos' es, por supuesto, grande, ya que es esencialmente combinatorio en tamaño. Entonces, factorizan esta matriz para producir una matriz de menor dimensión (palabra x características), donde cada fila ahora produce una representación vectorial para cada palabra. En general, esto se hace minimizando una 'pérdida de reconstrucción' que intenta encontrar las representaciones de menor dimensión que sirva para representar la mayor parte de la varianza en los datos de alta dimensión.
- FastText [7] es una extensión del modelo Word2Vec hecha por el equipo de Facebook. Cada palabra es tratada como la suma de sus composiciones de caracteres llamados *n-grams*. El vector para una palabra está compuesto por la suma de sus *n-grams*. Por ejemplo el vector para la palabra 'gato' está compuesto por la suma los vectores para los *n-grams* '<ga, gat, gato >, at, ato >, to >'. De esta forma se espera obtener mejores representaciones para palabras menos frecuentes, con menos apariciones en corpus de textos, y así poder generar vectores para palabras que no se encuentran en el vocabulario de los word embeddings.

Si bien existen muchas más herramientas, una pequeña introducción a estas tres es suficiente para nuestro estudio.

3. Objetivos y metodología

La idea en este trabajo es desarrollar un sistema de preguntas y respuestas basado en analogías haciendo uso de los Word Embeddings sobre un corpus de texto propio. Para ello, debemos crear y entrenar un modelo al que poder hacer consultas basándonos en analogías. Un ejemplo de consulta sería 'hijo' es a 'hijos' lo que 'hija' es a 'x', donde esperaríamos que nuestro modelo predijera el vector 'hijas'. Así pues, se proponen los siguientes objetivos:

1. Estudio de las tecnologías de Word Embeddings.
2. Obtención de un modelo para la extracción semántica.
3. Análisis de la parametrización de modelos para la optimización de los resultados semánticos.
4. Desarrollo de una interfaz de consultas para la explotación de modelos Word Embeddings.

Para dichos objetivos se propone una metodología compuesta de las siguientes tareas:

1. Elegir un dominio sobre el que trabajar y recopilar un gran corpus de este con el que entrenar nuestro modelo.
2. Entrenar varios modelos con distintos parámetros y herramientas en un script automatizado.
   2.1. Preprocesar el corpus.
   2.2. Elegir las herramientas y los parámetros de entrenamiento variables.
   2.3. Guardar los modelos.
3. Evaluar nuestros modelos mediante un dataset de evaluación propio de analogías del dominio.
   3.1. Crear el dataset.
   3.2. Diseñar el modelo de evaluación.
   3.3. Automatizar el proceso de evaluación.
   3.4. Obtener gráficas de precisión final.
4. Crear una interfaz web donde poder consultar palabras en el mejor modelo haciendo uso de las analogías.

4. Diseño e implementación del trabajo realizado

4.1. Código

Todo el código implementado en este trabajo está disponible bajo el repositorio de GitHub *tfg-repo* de mi usuario *WhiteSockLoafer*[5] bajo una licencia MIT que garantiza el uso, modificación y distribución del código a quien sea en cualquiera de sus formas.

Los scripts más importantes los podemos encontrar bajo el **Anexo I**.

4.2. Dominio y Corpus

Como ya se comentó en la **sección 2.4**, el corpus del dominio elegido debe ser representativo, fácil de extraer, específico y ad hoc con la tarea objetivo del embedding. la idea es no usar uno ya preparado puesto que parte de la línea de trabajo es la obtención de un corpus propio. Al inicio se intentó encaminar el proyecto por el mundo de la música, concretamente al análisis de letras de canciones. En la web *GENIUS Lyrics*[6] se encuentran las letras de las canciones de la mayoría de artistas populares y, además, la web cuenta con un servicio API público con el que resulta relativamente fácil extraer las letras de montones de canciones haciendo uso de algún script. Después de automatizar el proceso de obtención del corpus e indagar en dominio, se llegó a la conclusión de que este no era un buen dominio para la tarea encomendada. Sí, el corpus era representativo, fácil de obtener y de mucho volumen, pero no se ajustaba al problema que nuestro modelo quería resolver, pues las letras de canciones no representan un campo exacto con el que poder evaluar la precisión de un embedding en lo que a las palabras y sus relaciones se refiere. Para otros problemas como la generación de palabras automatizadas o recomendadores de contenido sí podría ser interesante este campo, no para el nuestro.

Al replantear el problema se propuso un analizador de textos jurídicos en español. La web de la *Agencia Estatal del Boletín Oficial del Estado*[7] cuenta con una biblioteca jurídica muy extensa y en un formato web de fácil extracción. Esta web pone a disposición pública documentos españoles dotados desde 1661 hasta la actualidad de todas las comunidades autónomas en todos los idiomas del país. El corpus que puede obtenerse es representativo del dominio y, además, es interesante comprobar si un embedding podría captar las relaciones entre términos jurídicos algo más complejos.

[5] `https://github.com/WhiteSockLoafer/tfg-repo`
[6] `https://genius.com/`
[7] https://www.boe.es/

Para la extracción del corpus se ha implementado el script en Python basado en la librería *Scrapy*[9] con la que podemos surcar una búsqueda de la web de la *Agencia Estatal del Boletín Oficial del Estado* siguiendo links de los resultados y descargando el cuerpo de la respuesta HTML de los mismos. El script lo podemos encontrar en el **Anexo I 'Extractor del BOE'** y una llamada al mismo sería:

```
$ python3 boe_extractor.py [PÁGINAS] [CORPUS_DIR] [URL_BÚSQUEDA]
```

Argumentos del script:

1. **Páginas**: Al realizar una búsqueda en el BOE podemos elegir el número de resultados por página, desde 50 hasta 2000. Como 2000 no son suficientes para obtener un corpus suficientemente extenso, con este elemento podemos elegir el número de páginas de resultados que nuestro script debería intentar extraer. Si por ejemplo las páginas fueran de 50 elementos y este argumento fuera 5, intentaría extraer $5 * 50 = 250$ elementos.
2. **Directorio del corpus**: Al correr el script, el resultado final esperado sería un directorio lleno de tantos ficheros como artículos del BOE haya descargado. Este argumento es la ruta relativa o absoluta al directorio, teniendo que estar creada la carpeta contenedor de antemano.
3. **URL de búsqueda**: Url de una búsqueda en la web de la *Agencia Estatal del Boletín Oficial del Estado* (*link de ejemplo*).

Desglosando el script: En la línea 15 se declara una 'araña' de la librería *Scrapy*, la cual es la encargada de navegar y extraer el contenido de una web según el comportamiento que le implementemos. En la línea 19 vemos como se inicializa la araña y sus variables especiales pasadas como argumentos del script en la línea 61. Definimos cómo empiezan las peticiones de la araña en la línea 26, donde primero ajunta las petición a la url de la búsqueda donde extraerá los artículos y, además, si existen ciertos documentos deseados que no sabemos si pudieran aparecer en la búsqueda, podemos añadir su url en la línea 7 a *ADDITIONAL_DOCS* para que esta función adjunte sus peticiones también, documentos interesantes como el código civil[8], la constitución[9] o el código penal[10]. La página de respuesta a la búsqueda es procesada por *parseSearch* en la línea 33, donde todos los resultados de las búsquedas vienen marcados en el cuerpo HTML dentro de una etiqueta 'a' de clase 'resultado-busqueda-link-defecto' (**Figura 12**). Si el link pertenece a un artículo del BOE (Boletín oficial del estado) intentaremos

[8] `https://www.boe.es/buscar/act.php?id=BOE-A-1889-4763`
[9] `https://www.boe.es/buscar/act.php?id=BOE-A-1978-31229`
[10] `https://www.boe.es/buscar/act.php?id=BOE-A-1995-25444`

extraer su contenido con la función *parseArticle* de la línea 45, donde se crea un fichero con el nombre del artículo y se escribe cada párrafo de este con cada frase en una línea distinta. El contenido se extrae de las etiquetas HTML 'p' cuya clase sea 'parrafo' o 'parrafo_2' (**Figura 13**), que son las etiquetas donde aparece texto bien formado y sin demasiados símbolos ruidosos como números o caracteres especiales. La idea es que bajo el directorio que hayamos pasado como segundo argumento del script queden todos los artículos del boe descargados.

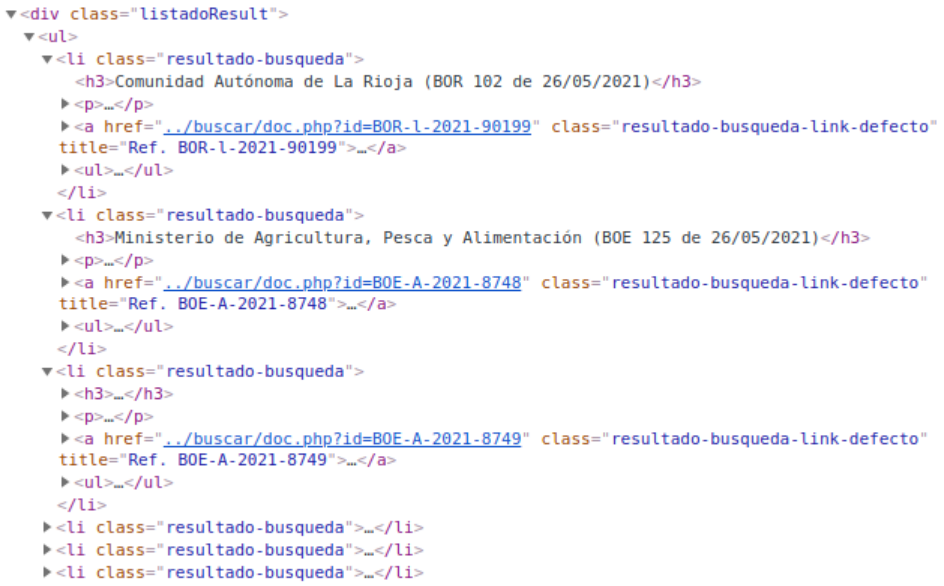

Figura 12: Cuerpo HTML de una búsqueda de artículos en la web del BOE

Figura 13: Contenido del artículo BOE-A-2021-8748

Para nuestro corpus, se inició el script con una url de una búsqueda del boe ordenada por fecha de publicación, 2000 resultados por página y 15 páginas de profundidad, lo

que (teniendo en cuenta que no todos los artículos pertenecen al boletín oficial del estado) hace aproximadamente 18000 artículos del BOE (480 MB de texto plano) con un vocabulario final de términos y populares a día de hoy (desde 2021 hasta 2015 salvo alguna excepción como el código civil).

4.3. Entrenamiento de los Embedding

En referencia a la sección 2.4, tras obtener nuestro corpus del dominio, se debe preprocesar este para posteriormente entrenar los posibles modelos finales con él y, por último, evaluarlos. Para nuestro estudio se ha decidido hacer uso de la librería *Gensim*[10] de *Python* para englobar todo el proceso de entrenamiento, pues cuenta con una API fácil de usar y parametrizar sobre los distintos tipos de modelos y tecnologías que podamos encontrar en la generación de embeddings, pudiendo instanciar modelos independientemente de su naturaleza o tecnología (LDA, LSA, Word2Vec, FastText...).

Para empezar, debemos preparar el preprocesamiento del corpus. La API de *Gensim* pide el corpus a la hora de llamar a un entrenamiento de una manera determinada, siendo esta una lista de oraciones donde las oraciones son listas de palabras, i.e. una lista de listas de palabras (**Figura 14**).

```
'''El gato atacó al ratón. El ratón se escapó.'''
[['el', 'gato', 'atacó', 'al', 'ratón'], ['el', 'ratón', 'se', 'escapó']]
```

Figura 14: Representación de un corpus para la API de *Gensim*

Como tener todo el corpus (480 MB de tamaño, recordemos) en una variable sería gastar cantidades innecesarias de memoria en tiempo de ejecución, se ha diseñado la clase *DirectoryCorpusReader* disponible en la librería de nuestro proyecto en el **Anexo I 'Librería de entreno y evaluación'** línea 10, donde la iteración sobre el corpus es 'memory-friendly', no teniendo que traer este al completo a memoria y preprocesando pequeñas partes de este al mismo tiempo que se usan. Un objeto de esta clase se inicializa pasando como variable el directorio donde se encuentra nuestro corpus. La idea es que podamos iterar sobre el objeto y este nos devuelva una lista de palabras que represente una frase preprocesada de cada una de las frases que hay en cada uno de los artículos existentes en el directorio del corpus. Haciendo uso de la función *yield* de *Python* conseguimos que no esté todo el corpus cargado en memoria.

Como se comentó en la **sección 4.2** el texto en los ficheros del corpus ya tienen las frases separadas por un salto de línea. Teniendo esto en cuenta, nuestro preprocesamiento lo podemos encontrar bajo el **Anexo I 'Librería de entreno y evaluación'** y consta de:

1. Eliminar todos los caracteres que no sean letras con o sin tildes o espacios (línea 24).
2. Poner todo el texto en minúscula para no diferenciar palabras que realmente son la misma (línea 25).
3. Eliminar palabras menores de dos caracteres pues producen cierto ruido innecesario (línea 25).
4. Dejar solo las frases de más de dos palabras (línea 19).

Como resultado, en la línea 13 del script en el **Anexo I 'Evaluación de los modelos'** tenemos un ejemplo de inicialización y en la línea 39 de uso.

Una vez diseñado el preprocesamiento, procedemos al entrenamiento. La idea de nuestro estudio es crear varios modelos con distintos parámetros de entrada (**sección 2.4**) y herramientas (**sección 2.6**). En cuanto a las herramientas se ha decidido incluir:

1. **Word2Vec** puesto que es bien demostrado que este modelo preentrenado genera embeddings muy precisos en lo que a tareas del procesamiento natural del lenguaje se refiere como la nuestra [19] [3] y tenemos la librería disponible en *Gensim*[10] para Python.
2. **FastText** pues como bien se comentó en la **sección 2.6** la herramienta hace uso de los n-gramas para entrenar el modelo y sería interesante comprobar si este método nos podría ayudar en nuestro estudio con analogías del ámbito jurídico. Existe también la posibilidad de usar esta herramienta con *Gensim*.

En cuanto a los parámetros de entrenamiento, introducimos de forma variable **las dimensiones del embedding final** y **el tamaño de la ventana**, pues tanto Word2Vec como FastText son modelos CBOW/Skip-Gram (**sección 2.3**) y gracias a estudios como el de Maryam Fanaeepour et al.[11] puede apreciarse que estos parámetros son los que realmente tienen influencia directa en la precisión final del modelo. Los valores que vamos a usar en estos parámetros están declarados en las líneas 20 y 21 del código en **Anexo I 'Creador y evaluador de modelos'**. Una vez claras las tecnologías y los parámetros, se procede a desglosar el script mencionado. Una llamada al script sería:

```
$ python3 train.py [CORPUS_DIR] [DATASER_PATH]
```

Argumentos del script:

1. **Directorio del corpus**: Ruta relativa o absoluta al directorio donde se encuentra nuestro corpus en ficheros separados. No hace falta usar un corpus extraído con nuestro extractor de la **sección 4.2**.

2. **Dataset**: Ruta relativa o absoluta al dataset de analogías con las que evaluaremos nuestros modelos. Se desarrolla en profundidad en la **sección 4.4**.

El resultado de iniciar el script es una carpeta */out* en la que encontraremos por un lado los modelos finales entrenados en el subdirectorio */models* y los resultados de la evaluación. En esta sección nos concentraremos en lo primero. La idea del script son dos *foreach*, uno recorre el array de los tamaños de vector y otro el de ventanas, teniendo de resultado dos modelos por iteración, uno de Word2Vec y otro FastText. Se han escogido los valores 200, 350 y 500 para las dimensiones de los embedding finales y 2, 5 y 10 para las ventanas basándonos en el estudio de Maryam Fanaeepour et al.[11] y lo comentado en la **sección 2.4**. En cuanto a los parámetros no variables, usamos 'workers' a 8 pues este parámetro representa el número de núcleos con el que cuenta el hardware con el que se entrenará el modelo para poder paralelizar el trabajo lo máximo posible y 'min_count' a 10 pues es el número mínimo de veces que debe aparecer una palabra para tenerla en cuenta y evitar así ruido provocado por alguna consideración que no hayamos tenido en cuenta en el preprocesamiento. Todos los modelos entrenados se almacenan para poder ser evaluados más tarde.

Ya que solo con los parámetros y tecnologías elegidas se deben entrenar 18 modelos distintos, el script tomará bastante tiempo en terminar, y, para tener algo de control sobre el estado del script, contamos con variables de control del tiempo y varios 'print' de monitorización.

Una vez terminadas las iteraciones de los *foreach* de las líneas 27 y 28, todos los modelos han sido creados, entrenados y guardados bajo el directorio */out/models*.

4.4. Evaluación de los modelos

Siguiendo con el script de la sección anterior (**Anexo I 'Creador y evaluador de modelos'**), se procede a explicar la segunda parte, la parte de evaluación.

Primero, se debe crear el dataset de evaluación. Este dataset debe contar con muchas analogías, cuantas más mejor para poder evaluar correctamente el modelo. Como se comentó en la **sección 2.5**, una analogía son relaciones horizontales y verticales entre cuatro palabras, repitiéndose estas relaciones entre ellas, pero no de forma cruzada. Si tuviéramos, por ejemplo, dentro del ámbito jurídico 'cobro es a acreedor lo que pago es a deudor', (i) acreedor realiza un cobro y deudor un pago y (ii) si hay un acreedor, hay un deudor y si hay un cobro, también hay un pago. En el dataset deben figurar las

analogías de forma que haya una en cada línea, con cada palabra separada por espacios, por lo que la analogía anterior se encontraría en nuestro dataset tal que 'cobro acreedor pago deudor'. La idea es introducirle al modelo que estemos evaluando las tres primeras palabras (cobro, acreedor, y pago) y comprobar que el resultado coincide con la última (deudor) de la siguiente manera (línea 6):

```
1 from gensim.models import Word2Vec
2
3
4 model = Word2Vec.load('trainer/model_trainer/out/models/word2vec_8.model')
5
6 print(model.wv.most_similar(negative=['cobro'], positive=['acreedor', 'pago'], topn=8))
```

Listing 1: Ejemplo de cómo introducir las tres primeras palabras de una analogía a un modelo

Recordemos la **figura 10**, cambiando las variables:

$$W_{ACREEDOR} - W_{COBRO} + W_{PAGO} \approx W_{DEUDOR}$$

El resultado de ejecutar las anteriores líneas de código es el siguiente:

```
[('deudor', 0.5627394914627075), ('ejecutante', 0.546737551689148), ('prestamista', 0.48770323395729065), ('arrendador', 0.4731365144252777),
('adjudicatario', 0.4473281502723694), ('contribuyente', 0.4395899176597595), ('avalista', 0.43529975414276123), ('prestatario', 0.43253549933
43353)]
```

Figura 15: Top 8 palabras más cercanas al vector resultado de la operación

Como se comentó en la **sección 2.5** el vector de la palabra esperada no tiene por qué coincidir exactamente con el vector resultado de la operación, y, es por esto mismo que normalmente se pide un número de palabras que mostrar, siendo 8 nuestro caso (línea 6, parámetro *topn* a 8) de cara a dar cierto margen de error. En nuestro caso, el vector correspondiente a la palabra 'deudor' aparece en primera posición, lo que es un acierto rotundo. La idea es recorrer el dataset haciendo esta misma operación por cada analogía que haya en él y apuntando los aciertos que haya, comprobando así la precisión de nuestro modelo.

El dataset final de nuestro estudio puede encontrarse en el anexo II con un total de 40 analogías representativas del ámbito jurídico español.

A la hora de evaluar un modelo con cierta analogía, normalmente solo el hecho de aparecer en la lista resultado la palabra esperada ya se considera un acierto, independientemente de si esta sale en primera o en última posición de la lista. A esto se le llama *precisión a n*, siendo n el número de palabras que tenemos en cuenta como margen de

error (parámetro *topn* en la línea 6 del código anterior). En nuestro estudio se decide ir un poco más allá, siendo más restrictivos a la hora de evaluar los modelos teniendo en cuenta también la posición en la que se encuentra nuestra palabra resultado, pues si hay una o más palabras más cercana al vector resultado que la que esperamos, creemos debería contar menos que si ninguna otra palabra se acercase más. En la **figura 15** el vector 'deudor' aparece en la primera posición y es por ello que obtendría la máxima puntuación en esa analogía. Veamos el caso de 'magistrado tribunal juez juzgado':

```
[('frob', 0.4513923227787018), ('órgano', 0.4426656663417816), ('legislador', 0.40924444794654846), ('juzgado', 0.37728235125541687),
 ('csn', 0.3758147656917572), ('notario', 0.3745468854904175), ('consejo', 0.3705185651779175), ('jurado', 0.3633576035499573)]
```

Figura 16: Top 8 palabras más cercanas al vector resultado de la analogía 'magistrado tribunal juez juzgado'

El vector 'juzgado' aparece en cuarta posición. Si bien es un acierto, no es tan rotundo como en el caso de 'deudor' y por ello debería penalizarse. Se propone entonces un modelo de evaluación en el que, además de tener en cuenta la aparición de la palabra, también su posición:

1. Si un modelo acertase todas las analogías en primera posición, este debería obtener un 100 % de precisión.
2. Si un modelo no consigue acertar ninguna analogía, no apareciendo el resultado en ninguna prueba, este debería obtener un 0 % de precisión.

Descomponiendo el problema, supongamos primero que el modelo de evaluación no tiene en cuenta las posiciones, solo la aparición la palabra esperada. Digamos que hay 4 analogías y evaluamos tres modelos: modelo 1, 100 % preciso, modelo 2 intermedio y un último 3 fatal con 0 % de precisión. Un 0 es un acierto, un 1 es un fallo. Podríamos tener entonces la **Tabla 1**.

Modelo	Array	Aciertos	Precisión
1	[0, 0, 0, 0]	4	100 %
2	[0, 1, 0, 0]	3	?? %
3	[1, 1, 1, 1]	0	0 %

Tabla 1: Ejemplo de evaluaciones de tres modelos sin tener en cuenta las posiciones de la palabra objetivo

Debemos calcular la precisión de los modelos intermedios, como el número 2. Al haber una máxima puntuación de 100, mínima de 0 y 4 analogías, el modelo pierde 25 puntos por cada fallo, la ecuación resultado parece sencilla. Siendo n el número de fallos:

$$100 - 25 \cdot n$$

Como el número de analogías es también un número variable, no siempre será 25, realmente sería la puntuación máxima entre el número de analogías a, quedando así:

$$100 - \frac{100}{a} \cdot n$$

Nuestro tercer modelo tendría una precisión del 75 %.

Añadamos ahora el problema de las posiciones, deshaciéndonos del array de aciertos e introduciendo un array con las posiciones de las palabras esperadas. Digamos que estamos usando topn 10. Como la posición 0 es la primera y 9 la última, consideraremos 10 como que la palabra no aparece en los resultados. Como todas las analogías tienen el mismo peso, podemos usar el sumatorio del array de posiciones. Podríamos obtener con los mismos modelos de la **Tabla 1** la **Tabla 2**.

Modelo	Posiciones	Sumatorio	Precisión
1	[0, 0, 0, 0]	0	100 %
2	[2, 10, 4, 7]	23	?? %
3	[10, 10, 10, 10]	40	0 %

Tabla 2: Ejemplo de evaluaciones de tres modelos teniendo en cuenta las posiciones de la palabra objetivo

Está claro que cuanto mayor sea el sumatorio, peor será la nota. Cuando no teníamos en cuenta las posiciones, un fallo costaba 25 puntos menos en la nota final, pero ahora un fallo no es que no aparezca la palabra, un modelo falla cada posición que la palabra esperada se aleja de la primera, son fallos más pequeños, lo que hace la nota mucho más granular. Debemos penalizar el modelo de forma proporcional a la gravedad del error.

Antes restábamos $\frac{100}{a}$ por fallo y ahora, dividimos el fallo en fallos más pequeños, representados por las posiciones en el top, es decir:

$$\frac{\frac{100}{a}}{10} = \frac{100}{a \cdot 10}$$

Y, como el número de palabras en el top t es variable:

$$\frac{100}{a \cdot t}$$

Quedando entonces nuestra ecuación para la nota final de un modelo:

$$100 - \frac{100}{a \cdot t} \cdot n$$

Recordemos, a es el número de analogías, t el número de palabras en el top y n el número de fallos, no siendo el mismo n que antes pues como ya se ha comentado, si tenemos en cuenta las posiciones, un fallo pesa t veces menos que si no las tuviéramos en cuenta. Nuestro modelo número 2 tiene un 42,5 % de precisión. Sin duda un modelo de evaluación mucho más estricto que el habitual.

Volviendo al código del **Anexo I 'Creador y evaluador de modelos'**, una vez terminado el proceso de entrenamiento empieza la evaluación de los modelos en la línea 64. Nuestro script hace uso de la función *evaluate* y *makeplots* de la librería en el **Anexo I 'Librería de entreno y evaluación'**. *Evaluate* toma dos argumentos, el directorio con los modelos guardados (*/out/models*) y el número de palabras a tener en cuenta en el top. La idea es cargar cada uno de los modelos en el directorio con el *for* de la línea 46 y, recorrer el dataset de analogías cargando estas en el modelo, guardando así el resultado de cada una de ellas según el modelo planteado anteriormente, es decir, un array con tantas posiciones como analogías en el que se guarden las posiciones de las palabras esperadas. Si alguna palabra no existe en el modelo o no aparece en los resultados, se cuenta como error (línea 60 y 67). Si aparece la palabra, se guarda su posición en el array. De cada modelo, guardamos en un diccionario de python su id, nombre de modelo, ventana, tamaño de vector, array de posiciones y precisión con la ecuación antes descrita (líneas 73-82). Al terminar la función, volviendo al script principal, se guardará este diccionario en en un archivo JSON disponible bajo el directorio */out*. Este archivo JSON nos servirá para poder hacer gráficas estadísticas con la precisión de los modelos reflejada en ella.

Además de todo esto, se introduce la variable *bads*. Esta variable sirve para comprobar qué enologías del dataset no han sido acertadas por ningún modelo, siendo las más difíciles del conjunto y que quizá podrían replantearse.

Es en la línea 67 del script de entrenamiento y evaluación donde llamamos a *makeplots* con los resultados del script para devolvernos así una pequeña gráfica hecha con la librería *Matplotlib*[20] que guardaremos también en la carpeta */out* como resultado final.

4.5. Interfaz web

Como bien indica el título del trabajo, debemos crear un sistema de preguntas respuestas basado en analogías. Es interesante completar el proyecto con una pequeña interfaz gráfica lo más cómoda posible de cara a que cualquier persona pueda sacar provecho del estudio. En un principio y siendo muy optimistas, esta interfaz podría ir dirigida hacia personas del ámbito jurídico en España, desde un estudiante hasta cualquier persona ya curtida en el dominio, como podría ser un profesor. Es por esto

que se ha decidido hacer una interfaz web pues es la forma más sencilla que una persona tendría de hacer uso de nuestro modelo entrenado, pues raro sería no contar con un navegador web instalado en un ordenador cualquiera que este sea.

La interfaz se ha desarrollado en *DJANGO*, pues (i) el proyecto entero está desarrollado en python, al igual que django, (ii) las interfaces web suelen conllevar un contenedor frontend y otro backend, si bien nuestro frontend tendrá un carácter muy sencillo y no merece la pena separar la interfaz en dos contenedores, django puede usarse para unificar estos dos conceptos para casos como este y (iii) ya se contaba previamente con experiencia en esta tecnología, lo que facilita bastante la tarea teniendo en cuenta que el el objetivo principal del proyecto no era esta interfaz.

Todo el código de la interfaz se encuentra en el repositorio del trabajo[11]. Aquí solo comentaremos la estructura general y los puntos más importantes del proyecto.

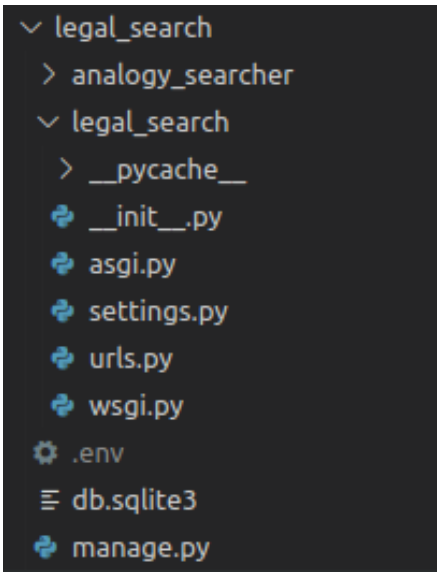

Figura 17: Estructura del proyecto I

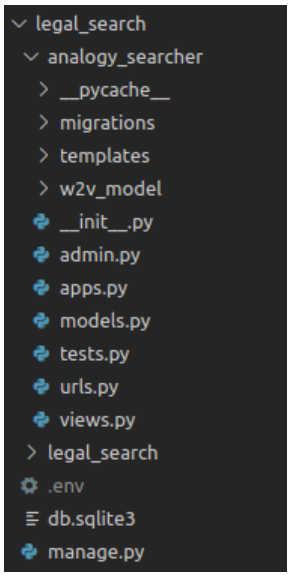

Figura 18: Estructura del proyecto II

En la **figura 17** se muestra la carpeta padre */legal_search* como directorio del proyecto. El proyecto principal y contenedor se encuentra bajo la subcarpeta */legal_search/legal_search* mientras que la subcarpeta */legal_search/analogy_searcher* es una app (la única) añadida al proyecto. Primero de todo, cabe destacar que en nuestro caso no nos hace falta hacer uso de una base de datos ni de un portal de administración que por defecto vienen en django. Dentro de la carpeta *analogy_searcher/w2v_model* encontraríamos el mejor modelo según nuestro estudio, el cual cargaremos en el proyecto más adelante.

[11] `https://github.com/WhiteSockLoafer/tfg-repo/tree/main/legal_search`

En */legal_search/legal_search/urls.py* y */legal_search/analogy_searcher/urls.py* establecemos las url visitables del proyecto.

```
1 #from django.contrib import admin
2 from django.urls import include, path
3
4 urlpatterns = [
5     # path('admin/', admin.site.urls),
6     path('', include('analogy_searcher.urls'))
7 ]
```

Listing 2: Archivo */legal_search/legal_search/urls.py*

```
1 from django.urls import path
2
3 from . import views
4
5 urlpatterns = [
6     path('', views.predict, name='predict')
7 ]
```

Listing 3: Archivo */legal_search/analogy_searcher/urls.py*

Como puede verse, en *urls.py* del proyecto principal eliminamos la entrada al portal de administración de django e incluimos todas las url de *analogy_searcher* en la raíz del sitio web, es decir, la app *analogy_searcher* es la encargada de las rutas. En este último archivo *urls.py* solo existe una ruta en la raíz en la que se carga la view *predict* (**figura 19**).

Comentamos ahora el código del **Anexo I 'Vista de la interfaz web'**. Esta última **figura 19** es el resultado de hacer una petición GET a la raíz de la web (línea 26), un portal donde poder realizar las consultas que se quieran basándonos en analogías del ámbito jurídico. El portal HTML está basado en la tecnología *Bootstrap*[21], basada en *JavaScript* con lo que el renderizado HTML toma un aspecto mucho más claro, cómodo y bonito.

Por defecto, el portal viene con un ejemplo sencillo para facilitar el entendimiento de la página 'hombre es a hombres lo que mujer es a ???'. Una vez pulsemos el botón 'Predict', mandaremos una petición POST a la raíz de la web (línea 12 del código). En el formulario del POST, encontraríamos los campos *n1*, *p1*, y *p2* correspondientes a las palabras de los vectores que hay que usar en la operación para obtener el cuarto vector

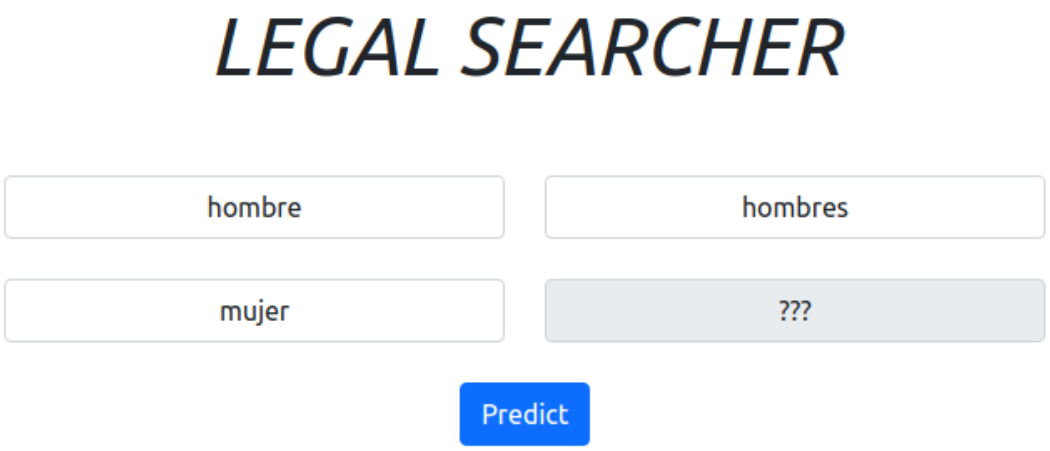

Figura 19: urls.py en *analogy_searcher*

de la analogía (**sección 2.5**). El modelo viene precargado antes de llamar a la view para no tener que cargarlo en cada petición (línea 6). A la hora de renderizar la view desde una petición POST, se pasa al HTML las variables del formulario para que las ponga de nuevo en los campos de la view y, además, la lista con los resultados de calcular el cuarto vector para que la los muestre con un pequeño porcentaje que representa la similitud coseno (**figura 20**).

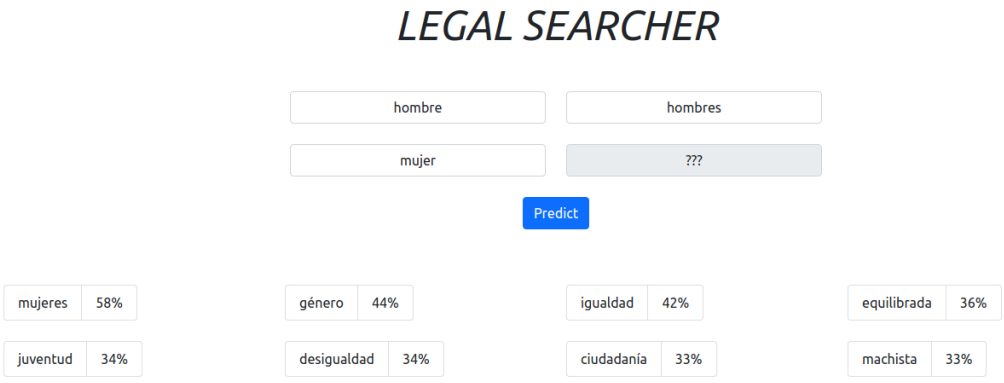

Figura 20: Predict view tras una petición POST bien formada

Cabe destacar que antes de enviar la petición POST, el cuerpo HTML realiza una validación del formulario, comprobando que todos los campos estén completos.

5. Análisis y resultados

Se procede a mostrar los resultados obtenidos de iniciar el script de entrenamiento y evaluación mostrado en el código del **Anexo I 'Creador y evaluador de modelos'**, el cual tardó **5 horas, 38 minutos y 25 segundos** en finalizar. Se recuerdan las variables del entrenamiento a tener en cuenta:

1. Hardware de 16 GB de RAM, 8 núcleos a 2 GHz de media
2. Corpus de 17.978 ficheros, exactamente 482,7 MB de texto plano
3. 18 modelos entrenados
    3.1. Herramientas FastText y Word2Vec
    3.2. Tamaños de ventana 2, 5 y 10
    3.3. Dimensiones del modelo 200, 350 y 500
4. Dataset de evaluación de 40 analogías
5. Número de palabras más cercanas al vector solución topn = 10

En las **Figuras 21, 22, 23 y 24** encontramos la salida por pantalla de nuestro script.

```
(venv) ➜  model_trainer git:(main) ✗ python3 train.py ../corpus ../analogies/dataset
Preparing models with:
        VECTOR SIZE -> 200
        WINDOW -> 2

Training model 1/18...
Time taken : 11.61 mins
Model fasttext 1 saved

Training model 2/18...
Time taken : 5.15 mins
Model word2vec 2 saved

Preparing models with:
        VECTOR SIZE -> 200
        WINDOW -> 5

Training model 3/18...
```

Figura 21: train.py I

```
Training model 3/18...
Time taken : 18.36 mins
Model fasttext 3 saved

Training model 4/18...
Time taken : 5.27 mins
Model word2vec 4 saved

Preparing models with:
        VECTOR SIZE -> 200
        WINDOW -> 10

Training model 5/18...
Time taken : 29.00 mins
Model fasttext 5 saved

Training model 6/18...
Time taken : 5.46 mins
Model word2vec 6 saved

Preparing models with:
        VECTOR SIZE -> 350
        WINDOW -> 2

Training model 7/18...
Time taken : 18.45 mins
Model fasttext 7 saved

Training model 8/18...
Time taken : 5.77 mins
Model word2vec 8 saved

Preparing models with:
        VECTOR SIZE -> 350
        WINDOW -> 5

Training model 9/18...
Time taken : 29.94 mins
Model fasttext 9 saved
```

Figura 22: train.py II

```
Training model 9/18...
Time taken : 29.94 mins
Model fasttext 9 saved

Training model 10/18...
Time taken : 6.01 mins
Model word2vec 10 saved

Preparing models with:
        VECTOR SIZE -> 350
        WINDOW -> 10

Training model 11/18...
Time taken : 47.38 mins
Model fasttext 11 saved

Training model 12/18...
Time taken : 6.33 mins
Model word2vec 12 saved

Preparing models with:
        VECTOR SIZE -> 500
        WINDOW -> 2

Training model 13/18...
Time taken : 22.87 mins
Model fasttext 13 saved

Training model 14/18...
Time taken : 6.02 mins
Model word2vec 14 saved
```

Figura 23: train.py III

```
Preparing models with:
        VECTOR SIZE -> 500
        WINDOW -> 5

Training model 15/18...
Time taken : 36.92 mins
Model fasttext 15 saved

Training model 16/18...
Time taken : 6.31 mins
Model word2vec 16 saved

Preparing models with:
        VECTOR SIZE -> 500
        WINDOW -> 10

Training model 17/18...
Time taken : 60.82 mins
Model fasttext 17 saved

Training model 18/18...
Time taken : 6.64 mins
Model word2vec 18 saved

Evaluating the models...
Time taken : 1.33 mins
Results report saved in ./out

None of the models passed the analogies:
[22, 25, 26, 29, 32, 33, 34, 35]
Total time taken : 330.41 mins

(venv) → model_trainer git:(main) ✗
```

Figura 24: Resultados IV

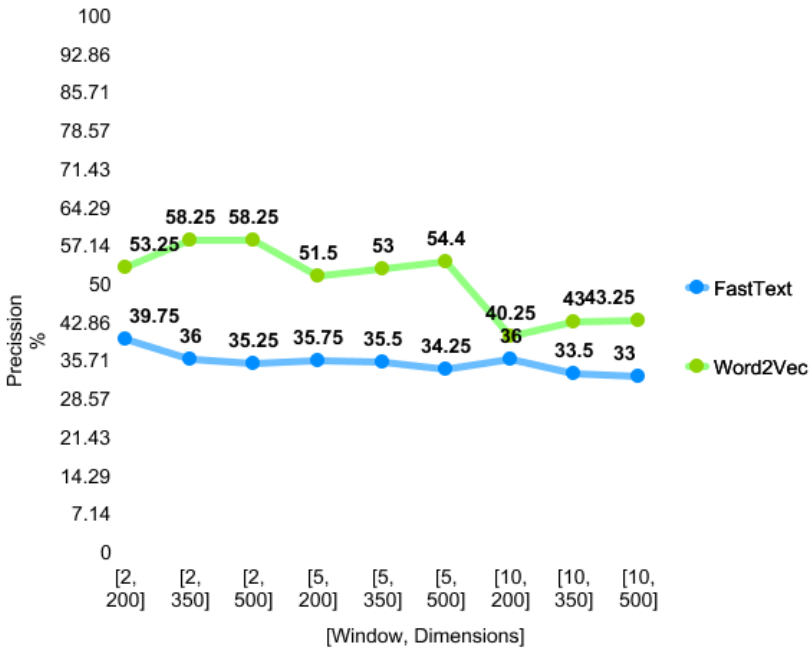

Figura 25: Gráfica de precisiones de los modelos

Además de la salida por terminal del script, se obtiene el archivo JSON comentado en la **sección 4.4** que, debido a su longitud, lo podemos encontrar en el **Anexo III**. Para ubicarnos mejor en el gráfico y las variables, se muestra una tabla (**Tabla 3**) con los datos más importantes de este fichero.

Modelo	Herramienta	Ventana	Dimensiones	Entrenamiento (min)	Precisión
1	Word2Vec	2	200	5,15	53,25 %
2	FastText	2	350	18,45	36 %
3	Word2Vec	5	500	6,31	54,4 %
4	FastText	10	500	60,82	33 %
5	FastText	5	350	29,94	35,5 %
6	FastText	2	500	22,87	35,25 %
7	FastText	2	200	11,61	39,75 %
8	Word2Vec	5	200	5,27	51,5 %
9	FastText	10	350	47,38	33,5 %
10	Word2Vec	10	200	5,46	40,25 %
11	Word2Vec	5	350	6,01	53 %
12	Word2Vec	10	350	6,33	43 %
13	Word2Vec	2	350	5,77	**58,25** %
14	Word2Vec	10	500	6,64	43,25 %
15	FastText	5	500	36,92	34,25 %
16	FastText	5	200	18.36	35,75 %
17	FastText	10	200	29	36 %
18	Word2Vec	2	500	6,02	**58,25** %

Tabla 3: Resultados de la evaluación de los modelos finales

Las analogías que aparecen al final de la salida del script en la **figura 24** son las que no han sido acertadas por ningún modelo. Podríamos pensar que son analogías mal formadas o poco representativas del dominio y por ende no fueran válidas, sin embargo son analogías bastante interesantes que hubiera estado genial que algún modelo captase, por ello no las quitamos del dataset y las tenemos en cuenta como difíciles.

Con los datos obtenidos de la **Tabla 3** se obtiene la gráfica **Figura 25** con las que podemos estudiar la evolución de la precisión sobre los parámetros de cada una de las tecnologías estudiadas.

Se procede a remarcar hechos interesantes de los resultados que posteriormente se analizarán: En primer lugar, en cuanto a los tiempos de entrenamiento, es claramente visible que FastText queda siempre en peor posición. Necesita en el peor de los casos hasta 10 veces más tiempo que Word2Vec en completar su entrenamiento. También cabe destacar que en los resultados parece que (i) Word2Vec es siempre más preciso que FastText pues FastText no supera nunca el 40 % de precisión y Word2Vec tampoco

baja de 40 y (ii) ventanas pequeñas son muy favorables pues puede verse claramente como las ventanas pequeñas suelen ser mucho más certeras independientemente de las dimensiones del modelo sobretodo en Word2Vec. Ventanas más grandes y mayores dimensiones ralentizan el proceso de entrenamiento como se podría esperar, si bien este caso no parece afectar a Word2Vec. Puede verse también que los modelos más precisos de todos tienen bastantes dimensiones. También vemos que, fijándonos solo en los modelos Word2Vec, entre 350 y 500 dimensiones del modelo final la precisión no cambia demasiado independientemente del tamaño de la ventana, si bien 500 suele ser ligeramente mejor.

Las respuestas del mejor modelo (ID 14 en la **Tabla 3**) a cada una de las analogías de nuestro dataset las podemos encontrar en el anexo V.

Analizando los resultados: En cuanto a que Word2Vec es ciertamente más rápido de entrenar que FastText, esto seguro se debe a los n-grams, pues las combinaciones de n-grams de un vocabulario son mayores que el número de palabras de este, necesitando bastante más tiempo de entrenamiento pues el número de operaciones sobre el modelo y su vocabulario siempre será mayor. A cambio, los n-grams ofrecen grandes ventajas como, por ejemplo, relacionar mejor las palabras compuestas o la aproximación de los vectores de tokens que no existan en el modelo y consultemos por ellos pues aunque el vector de la palabra no exista, los vectores de sus n-grams seguramente sí.

Pasando a otra cuestión, los resultados muestran como Word2Vec es siempre más preciso que FastText. Esto es cierto solo para el dataset de evaluación que hemos creado, pues si bien este tiene varios casos de palabras que comparten n-grams que podrían resultar beneficiosos para FastText, realmente no aparece ningún token que no pertenezca al corpus ni al vocabulario del modelo en sí. Es en estos casos que Word2Vec no podría aplicarse de ningún modo y FastText, como se comenta en el párrafo anterior, podría obtener una aproximación a la palabra resultado gracias a los n-grams, sería entonces que estos jugarían un papel importante en la precisión del modelo. Podría decirse también que estas tecnologías son muy diferentes y que los resultados no son muy representativos debidos al tamaño del corpus pues este podría ser algo pequeño.

Pasando a que las ventanas pequeñas son mucho más favorables que las grandes, esto está directamente relacionado con el corpus. Como bien aparece en trabajos como el de Maryam Fanaeepour et al.[11] y el de Pierre Lison y Andrei Kutuzov[12] no existe un tamaño de ventana definitivo para todos los modelos pues la idea es obtener la ventana más pequeña que recoja perfectamente el contexto y eso depende directamente de las construcciones de cada una de las frases del corpus. El mejor valor de este parámetro es dependiente de cada corpus. Al obtener los mejores resultados en ventanas pequeñas se podría concluir que en el corpus la semántica de una palabra no está relacionada con palabras distantes. El contexto es muy reducido. Esto es posible si el texto es muy denso en conceptos diferentes. Frases cortas y concisas en vez de una narrativa larga sobre un

único concepto. En el BOE, las secciones son cortas y hablan de muchos temas, leyes, directivas. Normalmente una sección no está relacionada con la siguiente, por lo que ventanas grandes no tendrían sentido.

En cuanto al número de dimensiones del modelo, este parámetro también está estrechamente relacionado con el corpus. Al ver que más dimensiones hacen nuestro modelo más preciso en la mayoría de casos, entendemos que el corpus es representativo del dominio. El hecho de que entre 350 y 500 dimensiones la precisión no cambie demasiado podría deberse a que sobre esas cifras los modelos obtienen una precisión del contenido del corpus acorde con relaciones semánticas entre las palabras y podría ser que al subir el número de dimensiones, los modelos no conseguirían representar mejor estas relaciones, si bien esto no está comprobado en este estudio. El hecho de que no cambie demasiado la precisión en este caso puede deberse también a que el dataset no es lo suficientemente grande, pues con menos pruebas los resultados no son tan representativos.

6. Conclusiones y vías futuras

Para abordar esta sección se procede a ir paso por paso de la metodología del estudio (disponible en la **sección 3**) intentando plasmar primero todas las conclusiones del trabajo y después las posibles mejoras y vías futuras.

En las conclusiones podemos centrarnos en la parte de entrenamiento y evaluación. Hablando de las precisiones de los modelos y siempre en el ámbito/tarea de las analogías con nuestro dataset de evaluación y corpus propio, dado que FastText no supera el 40 por ciento de precisión, concluimos que no la vamos a usar para nuestro modelo final. Cabe destacar que no rechazamos rotundamente FastText, si bien concluimos que no es una herramienta muy acorde para nuestro estudio en concreto.

También podemos concluir que, aunque en los modelos Word2Vec no cambie demasiado la precisión entre las dimensiones 350 y 500, 500 suele ser ligeramente mejor. Por esto y la razón dada en el apartado anterior, decidimos quedarnos con el modelo Word2Vec de ventana 2 y 500 dimensiones como modelo más preciso para nuestra interfaz web en django.

En cuanto a mejoras, primero se comentará sobre el corpus. Este tiene un tamaño suficiente para un proyecto tal como un trabajo fin de grado, sin embargo, en proyectos futuros sería interesante aumentar su tamaño hasta ser como otros corpus disponibles en internet de 2-10 GB. El tamaño de este podría aumentarse sin mucho esfuerzo añadiendo como primer parámetro del script extractor del BOE el número de páginas que nos aparezca en el resultado de la búsqueda de cara a extraer absolutamente todos los artículos existentes del BOE. También sería posible modificar el extractor, pues si bien las etiquetas del HTML que extraemos son las que principalmente tienen el texto interesante, podrían buscarse otras etiquetas interesantes (que las hay) y aumentar así el corpus. Se podrían añadir también los boletines de cada comunidad autónoma cuya lengua principal sea el castellano también o, yendo más allá, encontrar otra fuente de corpus que no sean los boletines del estado, como libros jurídicos y demás. Aumentar el tamaño del corpus ayudaría a afianzar los resultados obtenidos.

En cuanto al entrenamiento, sería interesante probar una herramienta que no esté basada en ventanas de contexto locales, sino en factorización global de matrices como *GLOVE*, pues a la hora de utilizarlos frente al problema de las analogías no suelen dar malos resultados y compararíamos así los dos métodos de generación de embeddings. También hubiera sido interesante poner todavía más parámetros, pero se contaba con ciertas limitaciones de hardware que harían el entrenamiento muy largo. También sería interesante hacer más modelos con mayores dimensiones para comprobar si la precisión de estos aumentara, se estancara o se viera reducida ya que los modelos más precisos tienen el máximo número de dimensiones probado.

Pasando a la evaluación de los modelos, si bien el dataset es muy representativo, es algo corto. Convendría aumentar su tamaño para asegurar que los resultados realmente son los obtenidos. Además, sería muy interesante introducir palabras que no formaran parte de los modelos para comprobar mejor las capacidades de FastText en el vocabulario jurídico, lo que haría que los modelos Word2Vec perdieran bastante precisión pues no podría calcular un resultado aproximado como lo haría FastText.

En la interfaz web podría añadirse cierta tolerancia a fallos cargando también el mejor modelo FastText de cara a que, cuando un usuario consulte un token no presente en el corpus, la consulta se realizara en este otro modelo.

Comentando las vías futuras, realmente este trabajo se enfocaba en obtener el mejor embedding posible para crear un sistema de preguntas/respuestas al uso, si bien los resultados obtenidos a la hora de evaluar los modelos pueden ser bastante interesantes para un estudio de embeddings sobre la obtención de los parámetros de entrenamiento óptimos en la tarea de las analogías, siendo este estudio una posible vía. Para ello, podría reutilizarse prácticamente todo el código de entrenamiento y evaluación pues el proyecto entero se diseñó para ser reutilizado en cualquier ámbito, no solo en el jurídico ni con un corpus obtenido con nuestro extractor. Para este estudio intentaría obtener más datos estadísticos de los modelos para, por ejemplo, ver cómo afecta el tamaño de ventana a la precisión con cierta dimensión fija del embedding o, también, esto mismo pero con las tecnologías usadas en el estudio.

Otra vía futura podría ser mejorar la interfaz web y añadir ciertas mejoras como por ejemplo un botón de ayuda o algún tutorial interactivo pues buscar cierto termino mediante una analogía no es algo a lo que se tenga costumbre pues normalmente buscamos términos en base a su similaridad con otros, utilizando relaciones simples. Sin duda hacer la interfaz más cómoda.

Referencias

[1] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003.

[2] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, page 160–167, New York, NY, USA, 2008. Association for Computing Machinery.

[3] Tobias Schnabel, Igor Labutov, David Mimno, and Thorsten Joachims. Evaluation methods for unsupervised word embeddings. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 298–307, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[4] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[5] Tomas Mikolov, Kai Chen, Greg S. Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[6] Carl Allen and Timothy Hospedales. Analogies explained: Towards understanding word embeddings, 2019.

[7] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*, 2016.

[8] Omer Levy and Yoav Goldberg. Dependency-based word embeddings. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 302–308, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[9] Dimitrios Kouzis-Loukas. *Learning Scrapy*. Packt Publishing Ltd, 2016.

[10] Radim Řehůřek and Petr Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. `http://is.muni.cz/publication/884893/en`.

[11] Maryam Fanaeepour, Adam Makarucha, and Jey Han Lau. Evaluating word embedding hyper-parameters for similarity and analogy tasks. *CoRR*, abs/1804.04211, 2018.

[12] Pierre Lison and Andrey Kutuzov. Redefining context windows for word embedding models: An experimental study. In *Proceedings of the 21st Nordic Conference on Computational Linguistics*, pages 284–288, Gothenburg, Sweden, May 2017. Association for Computational Linguistics.

[13] J. Firth. A synopsis of linguistic theory 1930-1955. In *Studies in Linguistic Analysis*. Philological Society, Oxford, 1957. reprinted in Palmer, F. (ed. 1968) Selected Papers of J. R. Firth, Longman, Harlow.

[14] Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1442–1451, Copenhagen, Denmark, September 2017. Association for Computational Linguistics.

[15] Dot CSV. Intro al natural language processing (nlp). `https://www.youtube.com/watch?v=Tg1MjMIVArc`. [Online; accessed August-2021].

[16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[17] P. D. Turney and P. Pantel. From frequency to meaning: Vector space models of semantics. *Journal of Artificial Intelligence Research*, 37:141–188, Feb 2010.

[18] Carl Allen. 'analogies explained'... explained. `https://carl-allen.github.io/nlp/2019/07/01/explaining-analogies-explained.html`. [Online; accessed August-2021].

[19] Marco Baroni, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 238–247, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

[20] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[21] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Number 57 in Monographs on Statistics and Applied Probability. Chapman & Hall/CRC, Boca Raton, Florida, USA, 1993.

Anexo I

Este anexo solo pretende mostrar el código para poder ser referenciado más tarde. La explicación y desglose de los distintos scripts que se muestran aparecerán en cada uno de los subapartados correspondientes dentro de la **sección 4**.

Extractor del BOE

```
from os import name
import sys
import scrapy
import re
from scrapy.crawler import CrawlerProcess

ADITTIONAL_DOCS = [
    'https://www.boe.es/buscar/act.php?id=BOE-A-1889-4763',     # CÓDIGO CIVIL
    'https://www.boe.es/buscar/act.php?id=BOE-A-1978-31229',    # CONSTITUCIÓN
    'https://www.boe.es/buscar/act.php?id=BOE-A-1995-25444',    # CODIGO PENAL
    'https://www.boe.es/buscar/act.php?id=BOE-A-2000-323',      # LEY ENJUICIAMIENTO CIVIL
    'https://www.boe.es/buscar/act.php?id=BOE-A-2015-11430'     # ESTATUTO DE LOS TRABAJADORES
]

class BoeSpider(scrapy.Spider):

    name = "articulos"

    def __init__(self, pages, corpus_dir, start_url):
        super().__init__()
        self.pages = pages
        self.corpus_dir = corpus_dir
        self.start_url = start_url


    def start_requests(self):
        yield scrapy.Request(url=self.start_url, callback=self.parseSearch)

        for url in ADITTIONAL_DOCS:
            yield scrapy.Request(url, callback=self.parseArticle)

```

```
32
33      def parseSearch(self, response):
34          self.pages -= 1
35
36          for result in response.css("a.resultado-busqueda-link-defecto::attr(href)").getall():
37              if '=BOE-' in result:
38                  yield response.follow(result, callback=self.parseArticle)
39
40          if self.pages > 0:
41              next = response.xpath('//a[span/@class="pagSig"]/@href').get()
42              yield response.follow(next, callback=self.parseSearch)
43
44
45      def parseArticle(self, response):
46          filename = re.search(r'.*id=(.*)', response.url).group(1)
47          with open(self.corpus_dir + '/' + filename, 'wb') as f:
48              for parrafo in response.xpath(
49                      '//p[@class="parrafo"]/text()|//p[@class="parrafo_2"]/text()'
50              ).getall():
51                  parrafo = re.sub(r'\.', '\n', parrafo)
52                  f.write(bytes(parrafo + '\n', 'utf-8'))
53          print(f'Saved file {filename}')
54
55
56  if __name__ == '__main__':
57      if len(sys.argv) == 4:
58          crawler = CrawlerProcess(settings={
59              'LOG_LEVEL': 'ERROR'
60          })
61          crawler.crawl(
62                  BoeSpider,
63                  pages=int(sys.argv[1]),
64                  corpus_dir=sys.argv[2],
65                  start_url=sys.argv[3]
66          )
67          crawler.start()
```

Librería de entreno y evaluación

```python
import os
import re
import statistics
import numpy as np
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from gensim.models import Word2Vec, FastText


class DirectoryCorpusReader(object):

    def __init__(self, dir_path):
        self.dir_path = dir_path

    def __iter__(self):
        for file_name in os.listdir(self.dir_path):
            for line in open(os.path.join(self.dir_path, file_name), encoding='utf-8'):
                line = self.preprocess(line)
                if len(line) > 2:  # Solo líneas mayores de 2 palabras
                    yield line

    def preprocess(self, line: str):
        # Eliminar caracteres que no sean del alfabeto castellano o espacios
        line = re.sub(r'[^a-zA-ZÀ-ÿ\u00f1\u00d1 ]', '', line)
        return [token for token in line.lower().split() if len(token) > 2]


class AnalogiesDatasetReader(object):

    def __init__(self, dataset_path):
        self.dataset_path = dataset_path

    def analogies(self):
        for line in open(self.dataset_path, encoding='utf-8'):
            yield line.split()

    def __len__(self) -> int:
        with open(self.dataset_path) as f:
            len = sum(1 for line in f)
```

```
        return len

    def evaluate(self, models_dir: str, topn: int) -> dict:
        results = []
        bad = [0 for i in range(len(self))]
        model_index = 0
        for file_name in os.listdir(models_dir):
            if file_name.endswith('.model'):
                model_index += 1
                if 'fasttext' in file_name:
                    model = FastText.load(os.path.join(models_dir, file_name))
                else:
                    model = Word2Vec.load(os.path.join(models_dir, file_name))

                marks = []

                for i, analogy in enumerate(self.analogies()):
                    try:
                        similars = model.wv.most_similar(
                            negative=[analogy[0]], positive=[analogy[1], analogy[2]], topn=topn)
                    except KeyError:
                        marks.append(topn)
                        bad[i] += 1
                    else:
                        similar = next((
                            (index, tuple) for (index, tuple) in enumerate(similars)
                            if tuple[0] == analogy[3]), None)
                        if similar is None:
                            marks.append(topn)
                            bad[i] += 1
                        else:
                            marks.append(similar[0])

                results.append(
                    {
                        "id": model_index,
                        "model_name": file_name,
                        "window": model.window,
                        "vector_size": model.vector_size,
                        "marks": marks,
                        "accuracy": 100 - (100/len(self) * sum(marks)/topn)
                    }
```

```
                )
        bads = [index + 1 for index,
                errors in enumerate(bad) if errors == model_index]
        return results, bads


def makeplots(report: dict):
    x = [model["id"] for model in report]
    y = [model["accuracy"] for model in report]

    plt.plot(x, y, marker=".")
    plt.xlabel("Model ID")
    plt.ylabel("Accuracy")
    return plt
```

Creador y evaluador de modelos

```
import os
import sys
import json
from lib import DirectoryCorpusReader, AnalogiesDatasetReader, makeplots
import time
from gensim.models import Word2Vec, FastText


if __name__ == '__main__':

    if len(sys.argv) == 3 and not os.path.exists('./out'):

        corpus = DirectoryCorpusReader(dir_path=sys.argv[1])
        dataset = AnalogiesDatasetReader(dataset_path=sys.argv[2])

        models_dir = './out/models'
        os.makedirs('./out')
        os.makedirs(models_dir)

        vector_sizes = [200, 350, 500]
        windows = [2, 5, 10]

```

```
23          init_time = time.time()
24
25          i = 0
26
27          for vector_size in vector_sizes:
28              for window in windows:
29
30                  print('Preparing models with:')
31                  print('\tVECTOR SIZE -> ' + str(vector_size))
32                  print('\tWINDOW -> ' + str(window) + '\n')
33
34                  i += 1
35                  print('Training model ' + str(i) + '/'
36                          + str(len(vector_sizes) * len(windows) * 2) + '...')
37                  partial_time = time.time()
38                  model = FastText(
39                          sentences=corpus,
40                          vector_size=vector_size,
41                          window=window,
42                          min_count=10,
43                          workers=8
44                  )
45                  print(f'Time taken : {(time.time() - partial_time) / 60:.2f} mins')
46                  model.save(models_dir + '/fasttext_' + str(i) + '.model')
47                  print('Model fasttext_' + str(i) + ' saved\n')
48
49                  i += 1
50                  print('Training model ' + str(i) + '/' +
51                          str(len(vector_sizes) * len(windows) * 2) + '...')
52                  partial_time = time.time()
53                  model = Word2Vec(
54                          sentences=corpus,
55                          vector_size=vector_size,
56                          window=window,
57                          min_count=10,
58                          workers=8
59                  )
60                  print(f'Time taken : {(time.time() - partial_time) / 60:.2f} mins')
61                  model.save(models_dir + '/word2vec_' + str(i) + '.model')
62                  print('Model word2vec_' + str(i) + ' saved\n')
63
64          print('Evaluating the models...')
```

```
65          partial_time = time.time()
66          results, bads = dataset.evaluate(models_dir=models_dir, topn=10)
67          plot = makeplots(results)
68          plot.savefig("out/results.png")
69          with open('out/results.json', 'w') as f:
70              json.dump(results, f)
71          print(f'Time taken : {(time.time() - partial_time) / 60:.2f} mins')
72          print('Results report saved in ./out\n')
73          if bads:
74              print('None of the models passed the analogies: ')
75              print(bads)
76
77          print(f'Total time taken : {(time.time() - init_time) / 60:.2f} mins\n')
```

Vista de la interfaz web

```
1   import os
2   from django.shortcuts import render
3   from django.conf import settings
4   from gensim.models import Word2Vec
5
6   model = Word2Vec.load(os.path.join(
7       settings.BASE_DIR, 'analogy_searcher/w2v_model/word2vec_4.model'
8   ))
9
10
11  def predict(request):
12      if request.method == 'POST':
13          tuples = model.wv.most_similar(negative=[request.POST['n1']], positive=[
14                                         request.POST['p1'], request.POST['p2']], topn=8)
15          final_list = []
16          for t in tuples:
17              final_list.append((t[0], int(t[1] * 100)))
18
19          return render(request, 'post_predict.html', context={
20              "result_list": final_list,
21              "n1": request.POST['n1'],
22              "p1": request.POST['p1'],
23              "p2": request.POST['p2']
```

```
24
25          })
26      else:
27          return render(request, 'get_predict.html')
```

Anexo II

En este anexo se muestra el dataset de analogías de evaluación creado para nuestro estudio.

```
hombre hombres mujer mujeres
padre madre hijo hija
mancomunado solidario dolo culpa
negligencia diligencia directos indirectos
agravar atenuar favorable desfavorable
favorable desfavorable atenuar agravar
agresión abuso robo hurto
congreso diputados senado senadores
acción omisión activo pasivo
procedente improcedente legítimo ilegítimo
grave prisión leve multa
constitucional inconstitucional legítimo ilegítimo
juicio judicial arbitraje extrajudicial
cobro acreedor pago deudor
completo parcial indefinido temporal
vendedor comprador arrendador arrendatario
cedente cesionario prestatario beneficiario
demanda civil denuncia penal
escrito expreso oral tácito
magistrado tribunal juez juzgado
prescripción extinción caducidad cese
trabajador sindicato empresario patronal
estado ministerio comunidad consejería
universidad rector gobierno presidente
gobierno decretoley ayuntamiento ordenanza
comunidad consejería municipio concejalía
cedente cesionario avalista avalado
extrajudicial laudo judicial sentencia
objetivo forma subjetivo fondo
legal lícito ilegal ilícito
libertad derecho obligación deber
tangible físico intangible moral
contractual obligación extracontractual indemnización
individual privado colectivo público
grave delito leve falta
cedente cesionario propietario usufructuario
```

```
37 tutor tutela curador curatela
38 legal legítimo ilegal ilegítimo
39 capacitado incapacitado comparecencia incomparecencia
40 avalado avalista hipotecado hipotecante
```

Anexo III

En este anexo se muestra el archivo JSON resultado de correr el script train.py (**Anexo I 'Creador y evaluador de modelos'**) con los parámetros y variables descritas en la **sección 5**.


```
[
    {
        "id": 1,
        "model_name": "word2vec_2.model",
        "window": 2,
        "vector_size": 200,
        "marks": [
            0,1,0,0,0,10,0,0,0,10,0,10,2,0,10,0,9,0,10,6,10,
            10,0,0,10,10,10,9,10,1,0,10,10,10,10,3,0,0,1,5
        ],
        "accuracy": 53.25
    },
    {
        "id": 2,
        "model_name": "fasttext_7.model",
        "window": 2,
        "vector_size": 350,
        "marks": [
            0,3,10,3,0,10,3,10,10,0,10,0,0,0,10,0,10,10,10,10,
            10,10,10,6,10,10,10,10,10,0,10,10,10,10,10,8,3,0,0,0
        ],
        "accuracy": 36.0
    },
    {
        "id": 3,
        "model_name": "word2vec_16.model",
        "window": 5,
        "vector_size": 500,
        "marks": [
            0,1,0,0,0,10,0,0,0,10,3,10,2,0,10,0,10,0,7,2,
            8,10,1,2,10,10,10,8,10,0,0,10,10,10,10,2,0,1,5,0
        ],
        "accuracy": 54.5
    },
    {
```

```
        "id": 4,
        "model_name": "fasttext_17.model",
        "window": 10,
        "vector_size": 500,
        "marks": [
            0,3,10,8,0,10,2,10,10,0,10,0,1,1,10,0,10,10,10,10,10,
            10,10,10,10,10,10,10,10,0,10,10,10,10,10,10,1,0,0,2
        ],
        "accuracy": 33.0
    },
    {
        "id": 5,
        "model_name": "fasttext_9.model",
        "window": 5,
        "vector_size": 350,
        "marks": [
            0,3,10,6,0,10,1,10,10,0,10,0,0,0,10,0,10,10,10,
            10,10,10,10,8,10,10,10,10,10,0,10,10,10,10,10,5,4,0,0,1
        ],
        "accuracy": 35.5
    },
    {
        "id": 6,
        "model_name": "fasttext_13.model",
        "window": 2,
        "vector_size": 500,
        "marks": [
            0,3,10,3,0,10,4,10,10,0,10,0,0,0,10,0,10,10,10,
            10,10,10,10,10,10,10,10,10,10,0,10,10,10,10,10,7,1,0,0,1
        ],
        "accuracy": 35.25
    },
    {
        "id": 7,
        "model_name": "fasttext_1.model",
        "window": 2,
        "vector_size": 200,
        "marks": [
            0,3,10,2,0,10,2,10,3,0,10,0,0,0,10,0,10,10,10,10,
            10,10,4,4,10,10,10,10,10,0,10,10,10,10,10,10,3,0,0,0
        ],
        "accuracy": 39.75
```

```
    },
    {
        "id": 8,
        "model_name": "word2vec_4.model",
        "window": 5,
        "vector_size": 200,
        "marks": [
            0,2,1,3,4,10,0,1,0,10,1,10,3,0,10,0,7,1,10,2,4,
            10,3,1,10,10,10,10,10,1,0,10,10,10,10,2,0,3,4,1
        ],
        "accuracy": 51.5
    },
    {
        "id": 9,
        "model_name": "fasttext_11.model",
        "window": 10,
        "vector_size": 350,
        "marks": [
            0,3,10,10,1,10,2,7,10,0,10,0,0,0,10,0,10,10,10,10,
            10,10,10,9,10,10,10,10,10,0,10,10,10,10,10,7,5,0,0,2
        ],
        "accuracy": 33.5
    },
    {
        "id": 10,
        "model_name": "word2vec_6.model",
        "window": 10,
        "vector_size": 200,
        "marks": [
            0,1,1,3,10,10,0,0,0,10,5,10,5,0,9,0,10,1,10,10,10,
            10,3,10,10,10,10,10,10,1,0,10,10,10,10,5,1,4,10,0
        ],
        "accuracy": 40.25
    },
    {
        "id": 11,
        "model_name": "word2vec_10.model",
        "window": 5,
        "vector_size": 350,
        "marks": [
            0,1,0,0,2,10,0,0,0,10,2,10,3,0,10,0,10,0,2,3,
            10,10,1,2,10,10,10,7,10,0,0,10,10,10,10,4,0,1,10,0
```

```
        ],
        "accuracy": 53.0
    },
    {
        "id": 12,
        "model_name": "word2vec_12.model",
        "window": 10,
        "vector_size": 350,
        "marks": [
            0,2,0,0,10,10,0,0,3,10,4,10,3,0,10,0,10,0,3,10,
            7,10,3,10,10,10,10,10,10,0,0,10,10,10,10,5,0,10,7,1
        ],
        "accuracy": 43.0
    },
    {
        "id": 13,
        "model_name": "word2vec_8.model",
        "window": 2,
        "vector_size": 350,
        "marks": [
            0,1,0,1,1,4,0,1,0,10,0,10,2,0,10,0,10,0,4,3,10,10,0,
            0,10,10,10,3,10,0,0,10,10,10,10,3,0,0,4,0
        ],
        "accuracy": 58.25
    },
    {
        "id": 14,
        "model_name": "word2vec_18.model",
        "window": 10,
        "vector_size": 500,
        "marks": [
            0,2,0,1,8,10,0,3,0,10,2,10,3,0,10,0,10,0,10,7,10,
            10,3,10,10,10,10,10,10,0,0,10,10,10,10,2,0,2,10,4
        ],
        "accuracy": 43.25
    },
    {
        "id": 15,
        "model_name": "fasttext_15.model",
        "window": 5,
        "vector_size": 500,
        "marks": [
```

```
            0,3,10,5,0,10,1,10,10,0,10,0,0,0,10,0,10,10,10,10,10,10,
            10,10,10,10,10,10,10,0,10,10,10,10,10,9,3,0,0,2
        ],
        "accuracy": 34.25
    },
    {
        "id": 16,
        "model_name": "fasttext_3.model",
        "window": 5,
        "vector_size": 200,
        "marks": [
            0,3,10,8,0,10,1,10,7,0,10,0,1,0,10,0,10,10,
            10,10,10,10,10,5,10,10,10,10,10,0,10,10,10,10,10,6,6,0,0,0
        ],
        "accuracy": 35.75
    },
    {
        "id": 17,
        "model_name": "fasttext_5.model",
        "window": 10,
        "vector_size": 200,
        "marks": [
            0,3,10,10,0,10,0,10,10,0,10,0,1,0,10,0,10,10,10,10,10,10,10,
            6,10,10,9,10,10,0,4,10,10,10,10,6,6,0,0,1
        ],
        "accuracy": 36.0
    },
    {
        "id": 18,
        "model_name": "word2vec_14.model",
        "window": 2,
        "vector_size": 500,
        "marks": [
            0,1,0,0,0,3,0,1,0,10,1,10,2,0,10,0,10,0,3,5,10,10,
            0,0,10,10,10,3,10,0,0,10,10,10,10,2,0,0,5,1
        ],
        "accuracy": 58.25
    }
]
```

Anexo IV

En este anexo se muestra una gran ayuda a la hora de crear modelos y es que gracias a *TensorFlow*[16] existe el visualizador de embeddings comentado en la **sección 2.2** pero para visualizar cierto embedding propio no basta con subir el archivo del modelo pues cada librería guarda estos a su manera. Este visualizador pide los modelos de una manera concreta: un fichero con los vectores y otro con los tokens, coincidiendo la línea de los vectores con la de su token en ambos ficheros. Para obtener estos dos ficheros de un modelo creado con *Gensim* se creo el siguiente script basado en el código del usuario de *GitHub BikerMan*[12].

```
import os


if __name__ == '__main__':

    import sys
    from gensim.models import Word2Vec, FastText


    if len(sys.argv) == 3:

        # Load the model
        if sys.argv[2] == 0:
            model = Word2Vec.load(sys.argv[1])
        else:
            model = FastText.load(sys.argv[1])

        keyedvectors = model.wv

        modelpath, modelname = os.path.split(sys.argv[1])


        # Vector file, `\t` seperated the vectors and `\n` seperate the words
        out_v = open(os.path.join(modelpath, modelname + '_vecs.tsv'), 'w', encoding='utf-8')


        # Meta data file, `\n` seperated word
        out_m = open(os.path.join(modelpath, modelname + '_meta.tsv'), 'w', encoding='utf-8')
```

[12] https://gist.github.com/BrikerMan/7bd4e4bd0a00ac9076986148afc06507

```
29
30
31          # Write meta file and vector file
32          for index in range(len(keyedvectors.index_to_key)):
33              word = keyedvectors.index_to_key[index]
34              vec = keyedvectors.vectors[index]
35              out_m.write(word + "\n")
36              out_v.write('\t'.join([str(x) for x in vec]) + "\n")
37
38          out_v.close()
39          out_m.close()
40
```

Pasándole al script como argumentos (i) un 0 si es un modelo Word2Vec o un 1 si es FastText y (ii) la ruta absoluta o relativa al modelo este nos extraerá los dos ficheros con los que podremos visualizar el embedding.

Anexo V

En este anexo se encuentran los resultados de evaluación del mejor modelo frente a las analogías de nuestro dataset.

```
"hombre" es a "hombres" lo que "mujer" es a ...
[('mujeres', 0.5230973362922668), ('juventud', 0.3250623643398285),
('género', 0.2931413948535919), ('jóvenes', 0.2872019112110138),
('trabajadoras', 0.26955747604370117), ('trabajadoresas', 0.26638877391815186),
('adolescentes', 0.26404350996017456), ('niños', 0.2590080201625824)]
Respuesta correcta "mujeres"

"padre" es a "madre" lo que "hijo" es a ...
[('hijoa', 0.584918200969696), ('hija', 0.4976992905139923),
('hijohija', 0.45770594477653503), ('descendiente', 0.4122593104839325),
('lactante', 0.3895905911922455), ('trabajador', 0.3561922311782837),
('ascendiente', 0.35543718934059143), ('progenitor', 0.33251357078552246)]
Respuesta correcta "hija"

"mancomunado" es a "solidario" lo que "dolo" es a ...
[('culpa', 0.44808125495910645), ('temeridad', 0.4184897840023041),
('intencionalidad', 0.4125036895275116), ('negligente', 0.41125988960266113),
('oneroso', 0.4101444482803345), ('deslealtad', 0.4025282859802246),
('impericia', 0.40251651406288147), ('injusto', 0.39898306131362915)]
Respuesta correcta "culpa"

"negligencia" es a "diligencia" lo que "directos" es a ...
[('indirectos', 0.3156263828277588), ('prontitud', 0.30328789353370667),
('transfronterizos', 0.2948523759841919), ('telemedidos', 0.28457018733024597),
('compradores', 0.2616986632347107), ('electrointensivos', 0.25324395298957825),
('expedidores', 0.2518971562385559), ('transportistas', 0.25188034772872925)]
Respuesta correcta "indirectos"

"agravar" es a "atenuar" lo que "favorable" es a ...
[('desfavorable', 0.34066829085350037), ('favorables', 0.32287055253982544),
('gravoso', 0.313117653131485), ('preceptivo', 0.3079657554626465),
('restrictivo', 0.2932899594306946), ('beneficioso', 0.2902422547340393),
('positivo', 0.27979621291160583), ('beneficiosos', 0.27812549471855164)]
Respuesta correcta "desfavorable"

"favorable" es a "desfavorable" lo que "atenuar" es a ...
```

```
[('mitigar', 0.569034993648529), ('minimizar', 0.5256971716880798),
('corregir', 0.49375835061073303), ('agravar', 0.471000999212265),
('detectar', 0.46562469005584717), ('remediar', 0.46530959010124207),
('contrarrestar', 0.4599568545818329), ('combatir', 0.4582829177379608)]
Respuesta correcta "agravar"

"agresión" es a "abuso" lo que "robo" es a ...
[('hurto', 0.552147626876831), ('estafa', 0.4190156161785126),
('hurtos', 0.4112131893634796), ('abusos', 0.3862901031970978),
('deslealtad', 0.3556312322616577), ('malversación', 0.3513648509979248),
('culpa', 0.3466698229312897), ('extravío', 0.34361696243286133)]
Respuesta correcta "hurto"

"congreso" es a "diputados" lo que "senado" es a ...
[('parlamentarios', 0.4340519905090332), ('senadores', 0.41236624121665955),
('atc', 0.3997620940208435), ('resoluciónxii', 0.392731636762619),
('vox', 0.3886149525642395), ('presidenta', 0.37979474663734436),
('concejales', 0.36949869990348816), ('lamaña', 0.3677447438240051)]
Respuesta correcta "senadores"

"acción" es a "omisión" lo que "activo" es a ...
[('pasivo', 0.37326425313949585), ('impago', 0.3016324043273926),
('prestamista', 0.288974791765213), ('inmovilizado', 0.2884366512298584),
('concursado', 0.28819364309310913), ('pensionista', 0.2793843448162079),
('acreedor', 0.27224430441856384), ('culpable', 0.2702978551387787)]
Respuesta correcta "pasivo"

"procedente" es a "improcedente" lo que "legítimo" es a ...
[('constitucionalmente', 0.48091763257980347), ('considerarla', 0.4369203448295593),
('irrazonable', 0.4303227365016937), ('casacional', 0.43026724457740784),
('obviamente', 0.4278033375740051), ('lícito', 0.4217012822628021),
('perseguido', 0.417487233877182), ('intolerable', 0.41724810004234314)]
Respuesta correcta "ilegítimo"

"grave" es a "prisión" lo que "leve" es a ...
[('pena', 0.3907957673072815), ('multa', 0.388974130153656),
('leves', 0.3385821282863617), ('pecuniaria', 0.33683839440345764),
('accesoria', 0.3264700174331665), ('cometida', 0.3261418044567108),
('coercitiva', 0.3168184459209442), ('condenados', 0.3109738230705261)]
Respuesta correcta "multa"

"constitucional" es a "inconstitucional" lo que "legítimo" es a ...
```

```
[('lícito', 0.45881813764572144), ('inadmisible', 0.45776501297950745),
('irrazonable', 0.4527793824672699), ('improcedente', 0.4318810701370239),
('absurdo', 0.43020233511924744), ('irrelevante', 0.4227430522441864),
('infundada', 0.4218432307243347), ('inaplicable', 0.4184796214103699)]
Respuesta correcta "ilegítimo"

"juicio" es a "judicial" lo que "arbitraje" es a ...
[('arbitral', 0.44380128383636475), ('mediación', 0.4070674777030945),
('extrajudicial', 0.3798501789569855), ('jurisdiccional', 0.3448133170604706),
('sima', 0.3226775527000427), ('exequátur', 0.32006868720054626),
('recaída', 0.31560537219047546), ('recurrible', 0.31349876523017883)]
Respuesta correcta "extrajudicial"

"cobro" es a "acreedor" lo que "pago" es a ...
[('deudor', 0.535301923751831), ('ejecutante', 0.50408935546875),
('prestamista', 0.46819809079170227), ('arrendador', 0.45285332202911377),
('adjudicatario', 0.44188162684440613), ('avalista', 0.42414742708206177),
('fiador', 0.419495552778244), ('prestatario', 0.41611433029174805)]
Respuesta correcta "deudor"

"completo" es a "parcial" lo que "indefinido" es a ...
[('indefinida', 0.3969869911670685), ('interinidad', 0.35417690873146057),
('discontinuo', 0.339643657207489), ('indefinidos', 0.3310133218765259),
('relevo', 0.322131484746933), ('fijosdiscontinuos', 0.31319794058799744),
('forzoso', 0.31147637963294983), ('rescisiones', 0.3093112111091614)]
Respuesta correcta "temporal"

"vendedor" es a "comprador" lo que "arrendador" es a ...
[('arrendatario', 0.5548598766326904), ('propietario', 0.5391174554824829),
('adquirente', 0.515740156173706), ('deudor', 0.5156306028366089),
('acreedor', 0.5108984708786011), ('transmitente', 0.5100216269493103),
('concesionario', 0.5082752704620361), ('ejecutante', 0.49037906527519226)]
Respuesta correcta "arrendatario"

"cedente" es a "cesionario" lo que "prestatario" es a ...
[('arrendatario', 0.4707559645175934), ('deudor', 0.4597863669395447),
('acreedor', 0.4553987681865692), ('concesionario', 0.4540356993675232),
('arrendador', 0.4403144121170044), ('ejecutante', 0.4316662847995758),
('comprador', 0.41550520062446594), ('fiador', 0.40833356976509094)]
Respuesta correcta "beneficiario"

"demanda" es a "civil" lo que "denuncia" es a ...
```

```
[('penal', 0.38497674465179443), ('criminal', 0.3242737650871277),
('denunciante', 0.3025871217250824), ('consular', 0.2744201123714447),
('disciplinaria', 0.25577783584594727), ('denunciado', 0.25565436482429504),
('incoación', 0.2373858392238617), ('parental', 0.23479317128658295)]
Respuesta correcta "penal"

"escrito" es a "expreso" lo que "oral" es a ...
[('léxico', 0.42595797777175903), ('explícito', 0.4110555946826935),
('inequívoco', 0.38726806640625), ('tácito', 0.36224666237831116),
('homogéneo', 0.33639398217201233), ('imparcial', 0.3311103880405426),
('hablada', 0.3263520300388336), ('protocolario', 0.3154848515987396)]
Respuesta correcta "tácito"

"magistrado" es a "tribunal" lo que "juez" es a ...
[('órgano', 0.4287617802619934), ('frob', 0.3982861340045929),
('notario', 0.37160080671310425), ('legislador', 0.36893340945243835),
('consejo', 0.35847795009613037), ('juzgado', 0.35308533906936646),
('csn', 0.34114477038383484), ('depositario', 0.3335928022861 4807)]
Respuesta correcta "juzgado"

"prescripción" es a "extinción" lo que "caducidad" es a ...
[('rescisión', 0.45713651180267334), ('disolución', 0.4481040835380554),
('revocación', 0.4097289443016052), ('terminación', 0.4073202610015869),
('reanudación', 0.40616685152053833), ('finalización', 0.37844881415367126),
('anulación', 0.3759590685367584), ('reversión', 0.3686228096485138)]
Respuesta correcta "cese"

"trabajador" es a "sindicato" lo que "empresario" es a ...
[('ugt', 0.35639089345932007), ('sindical', 0.32297539710998535),
('confederación', 0.32085689902305603), ('cgt', 0.30916428565979004),
('coalición', 0.2988000214099884), ('ficaugt', 0.2981036603450775),
('fesmcugt', 0.2911997139453888), ('afiliados', 0.2907421886920929)]
Respuesta correcta "patronal"

"estado" es a "ministerio" lo que "comunidad" es a ...
[('consejería', 0.47427669167518616), ('conselleria', 0.40867167711257935),
('secretaría', 0.3963572680950165), ('lacomunidad', 0.3837534785270691),
('ministra', 0.38211190700531006), ('administración', 0.36662617325782776),
('ministro', 0.3594343066215515), ('subsecretaría', 0.3469300866127014)]
Respuesta correcta "consejería"

"universidad" es a "rector" lo que "gobierno" es a ...
```

```
[('presidente', 0.46570533514022827), ('presidenta', 0.4447888433933258),
('protectorado', 0.4278838336467743), ('consell', 0.4141805171966553),
('pleno', 0.40946048498153687), ('ministros', 0.3910943865776062),
('parlamento', 0.38689562678337097), ('ejecutivo', 0.3722210228443146)]
Respuesta correcta "presidente"

"gobierno" es a "decretoley" lo que "ayuntamiento" es a ...
[('decreto', 0.4881055951118469), ('decretolegislativo', 0.3841046988964081),
('decretode', 0.36176493763923645), ('terremoto', 0.34174254536628723),
('precepto', 0.3339965343475342), ('decretoleyse', 0.3237600028514862),
('folio', 0.32205235958099365), ('acequia', 0.32121214270591736)]
Respuesta correcta "ordenanza"

"comunidad" es a "consejería" lo que "municipio" es a ...
[('conselleria', 0.4667457342147827), ('ayuntamiento', 0.45035773515701294),
('concejo', 0.39081764221191406), ('departamento', 0.38163432478904724),
('consejero', 0.3460429310798645), ('cabildo', 0.3391216993331909),
('conseller', 0.33389410376548767), ('ministerio', 0.3206239938735962)]
Respuesta correcta "concejalía"

"cedente" es a "cesionario" lo que "avalista" es a ...
[('fiador', 0.5685616135597229), ('ejecutante', 0.4707629978656769),
('concesionario', 0.4688176214694977), ('arrendatario', 0.46800196170806885),(
'acreedor', 0.45470643043518066), ('deudor', 0.44900989532470703),
('hipotecante', 0.4453747272491455), ('fiadora', 0.4438911974430084)]
Respuesta correcta "avalado"

"extrajudicial" es a "laudo" lo que "judicial" es a ...
[('juez', 0.39130061864852905), ('pronunciamiento', 0.36976099014282227),
('acto', 0.3665791153907776), ('sentencia', 0.3524792492389679),
('auto', 0.34255850315093994), ('tribunal', 0.34094539284706116),
('fallo', 0.3297366499900818), ('providencia', 0.32961323857307434)]
Respuesta correcta "sentencia"

"objetivo" es a "forma" lo que "subjetivo" es a ...
[('manera', 0.44612976908683777), ('acrecer', 0.2892124652862549),
('reabrir', 0.26913875341415405), ('sumisión', 0.2609488368034363),
('canónico', 0.2547096908092499), ('perjudicará', 0.24692127108573914),
('ponderal', 0.24540235102176666), ('negación', 0.2411067634820938)]
Respuesta correcta "fondo"

"legal" es a "lícito" lo que "ilegal" es a ...
```

```
[('ilícito', 0.4007524251937866), ('fraudulento', 0.3971666991710663),
('lícita', 0.38963812589645386), ('ilegítimo', 0.3846958875656128),
('inaceptable', 0.38340887427330017), ('problemático', 0.3747386932373047),
('legitimen', 0.3736691474914551), ('inadecuado', 0.37173786759376526)]
Respuesta correcta "ilícito"

"libertad" es a "derecho" lo que "obligación" es a ...
[('deber', 0.41712111234664917), ('obligatoriedad', 0.39380887150764465),
('posibilidad', 0.32596641778945923), ('obligado', 0.31119921803474426),
('bno', 0.29844531416893005), ('fno', 0.285728394985199),
('obligada', 0.2812459170818329), ('necesidad', 0.2719182074069977)]
Respuesta correcta "deber"

"tangible" es a "físico" lo que "intangible" es a ...
[('biológico', 0.311628133058548), ('fisiológico', 0.29987460374832153),
('termorregulación', 0.29587292671203613), ('inmovilizado', 0.29267120361328125),
('psicológico', 0.28878340125083923), ('crediticio', 0.2845750153064728),
('reproductivo', 0.2821581959724426), ('táctico', 0.27829959988594055)]
Respuesta correcta "moral"

"contractual" es a "obligación" lo que "extracontractual" es a ...
[('eximiendo', 0.5157281160354614), ('ccrr', 0.47269806265830994),
('incurrirán', 0.42064937949180603), ('exonerará', 0.4187719523906708),
('extintivas', 0.3673422336578369), ('conllevando', 0.3608334958553314),
('incurrirá', 0.35960569977760315), ('parental', 0.35379189252853394)]
Respuesta correcta "indemnización"

"individual" es a "privado" lo que "colectivo" es a ...
[('marítimopesquero', 0.3279331922531128), ('ferroviario', 0.32561194896698),
('publico', 0.31012970209121704), ('agroalimentario', 0.308331698179245),
('portuak', 0.304889053106308), ('cementero', 0.2829363942146301),
('carbunión', 0.2820993959903717), ('hostelería', 0.2779422104358673)]
Respuesta correcta "público"

"grave" es a "delito" lo que "leve" es a ...
[('dolosos', 0.39230936765670776), ('infracción', 0.3740946352481842),
('doloso', 0.37166792154312134), ('cometida', 0.3672946095466614),
('prevaricación', 0.3650059401988983), ('contrabando', 0.35667747259140015),
('delitos', 0.3537571132183075), ('cómplice', 0.34319940209388733)]
Respuesta correcta "falta"

"cedente" es a "cesionario" lo que "propietario" es a ...
```

```
[('arrendatario', 0.5379943251609802), ('arrendador', 0.4949844181537628),
('usufructuario', 0.48047998547554016), ('armador', 0.47706010937690735),
('concesionario', 0.46235939860343933), ('prestatario', 0.457051157951355),
('comprador', 0.44698992371559143), ('tenedor', 0.4462100863456726)]
Respuesta correcta "usufructuario"

"tutor" es a "tutela" lo que "curador" es a ...
[('curatela', 0.4544343054294586), ('patria', 0.36954689025878906),
('tanteo', 0.3306014835834503), ('soberanía', 0.3242020010948181),
('potestad', 0.3234032392501831), ('curatelas', 0.3180482089519501),
('dominicales', 0.31309354305267334), ('extrajudicialmente', 0.3074568510055542)]
Respuesta correcta "curatela"

"legal" es a "legítimo" lo que "ilegal" es a ...
[('ilegítimo', 0.445285439491272), ('ilícito', 0.41701266169548035),
('inadecuado', 0.39449343085289), ('fraudulento', 0.3742329776287079),
('ilegalmente', 0.3721776008605957), ('prostitución', 0.3653620481491089),
('intolerable', 0.36428114771842957), ('abusivo', 0.3615281283855438)]
Respuesta correcta "ilegítimo"

"capacitado" es a "incapacitado" lo que "comparecencia" es a ...
[('filiación', 0.388742595911026), ('citación', 0.38311848044395447),
('dimisión', 0.37111708521842957), ('defunción', 0.3675076365470886),
('ruina', 0.35819554328918457), ('incomparecencia', 0.35223835706710815),
('fallecimiento', 0.33473464846611023), ('matrimonio', 0.3342515230178833)]
Respuesta correcta "incomparecencia"

"avalado" es a "avalista" lo que "hipotecado" es a ...
[('fiadora', 0.48817262053489685), ('hipotecante', 0.4870907664299011),
('originadora', 0.4778938591480255), ('fiador', 0.45296749472618103),
('transmitente', 0.45180603861808777), ('enajene', 0.43767598271369934),
('pignoraticio', 0.4341638684272766), ('propietaria', 0.43317335844039917)]
Respuesta correcta "hipotecante"
```