Universitat Politècnica de Catalunya

Ciència i Enginyeria de Dades

Processament Oral i Escrit
Word Vectors

Gerard Comas Quiles & Miquel Lopez Escoriza

February 2024

1 Introducción

Esta práctica se centra en comprender el funcionamiento del modelo Continuous Bag of Words (CBOW) y en implementar mejoras para aumentar su precisión. Aunque el objetivo principal sigue siendo explorar el modelo y comprender sus capacidades en su totalidad.

El CBOW es un modelo de aprendizaje automático utilizado en el procesamiento del lenguaje natural (PLN) para generar representaciones vectoriales densas de palabras en un corpus de texto. Su principal tarea es predecir una palabra objetivo basándose en el contexto de las palabras que la rodean, utilizando las tres palabras anteriores y las tres posteriores como contexto.

En esta práctica, utilizaremos gran parte de la Wikipedia en catalán como corpus de texto y realizaremos un preprocesamiento del mismo para tokenizarlo y eliminar signos de puntuación. A partir de estas palabras, construiremos nuestro propio vocabulario.

La estructura inicial del modelo CBOW consistirá en una matriz de parámetros aprendibles (**E**) que representan los encodings de las palabras en vectores únicos. Utilizaremos las seis palabras del contexto como entradas y sumaremos sus encodings para generar un nuevo vector (**u**). Luego, este vector pasará por una capa lineal (**w**) y devolverá la predicción de la palabra (**z**).

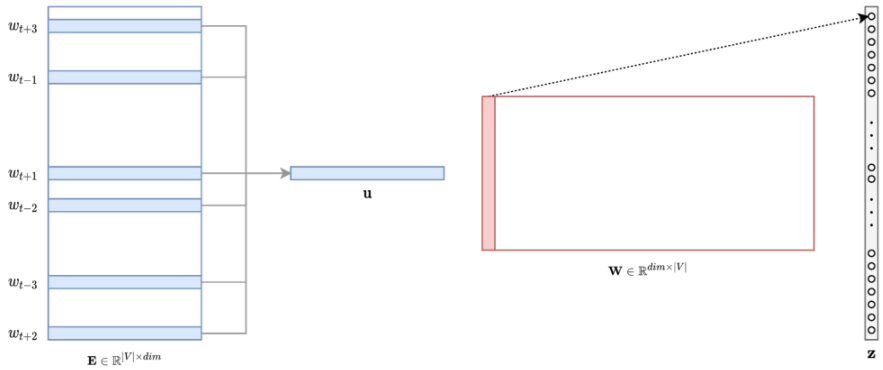

Figure 1: Arquitectura CBOW básica.

2 Improve CBOW model

El modelo básico de CBOW está bastante límitado, obteniendo accuracies bajas tanto en training, validación y test. Para el test, hemos cambiado el corpus de texto e intenta predecir palabras en contextos completamente distintos, en nuestro caso artículos del diario *El Perı́odico*.

En las siguientes subsecciones mostraremos posibles mejoras al módelo y una comparación quantitativa de los resultados.

2.1 CBOW Fixed Scalar Weight

El modelo CBOW se basa en sumar los encodings de las palabras de contexto para generar un nuevo vector, que luego corresponderá a la palabra predecida. Por como funciona nuestro lenguaje, nos puede parecer intuitivo que no todas las palabras del contexto sean igual de importantes y que las que más cerca esten tengan un mayor peso. Es por ello, que proponemos el Fixed Scalar Weight (FSW) como una posible mejora.

Ahora, el vector **u** resultante será la combinación lineal de los encodigns pero multiplicados por los pesos $[1, 2, 3, 3, 2, 1]$ según la posición de la palabra.

Esto equivale a introducir la siguiente linea de código en el método _init_() de la clase **CBOW**:

```
self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],
↪  dtype=torch.float32))
```

Esto permite al modelo centrarse mejor en las palabras más proximas a la predicción y mejora considerablemente la precisión. En la *Tabla 1* se puede evaluar con más detalle.

Modelo	**Training %**	**Validation %**	**Test %**
Base	26.2	23.9	15.4
FSW	33.6	31.4	21.4

Table 1: Comparación de acuracy del modelo FSW con el modelo base

2.2 CBOW Trained Scalar Weight

Los pesos del CBOW FSW se han basado en valores predefinidos por nosotros según nuestro criterio y nuestros conocimientos sobre el lenguaje. Aun así, sería interesante que fuera el propio modelo el que aprendiera cual han de ser los pesos idoneos para obtener mejores resultados. A este módelo le llamaremos CBOW Trained Scalar Weight (TSW). Como pesos iniciales usaremos los mismos que

en el FSW, aunque después compararemos resultados con otras inicializaciones.

Esto equivale a introducir la siguiente linea de código en el método _init_() de la clase **CBOW**:

```
self.position_weight = nn.Parameter(torch.tensor([1,2,3,3,2,1],
↪  dtype=torch.float32))
```

Tal y como esperabamos, dejar al modelo aprender los pesos de las palabras del contexto se ha reflejado en una mejora de la acuracy. En la *Tabla 2* se puede ver el cambio respecto a los modelos anteriores.

Modelo	**Training %**	**Validation %**	**Test %**
Base	26.2	23.9	15.4
FSW	33.6	31.4	21.4
TSW	35.4	33.3	22.8

Table 2: Comparación de acuracy del modelo TSW con los modelos anteriores

En cuanto los pesos del contexto nos parece lógico el resultado, pues las palabras centrales siguen teniendo la mayor fuerza dentro del contexto. Aunque las palabras posteriores a la palabra parecen tener un poco más de peso en el contexto.

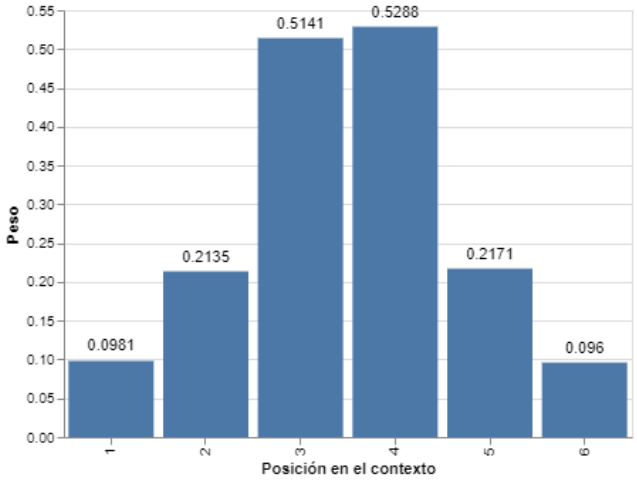

Figure 2: Pesos de las palabras de contexto

2.3 CBOW Trained Vector Weight

Vemos como dejar que el modelo aprenda sus parametros suele ser mejor que predefinirlos. A nosotros nos parece intuitivo que cada palabra según su posición en el contexto tenga un peso, pero al final las palabras son vectores de embbedings y forzar que todas las posiciones del vector valgan lo mismo puede que este limitando al modelo. Es por eso que proponemos un nuevo modelo llamado CBOW Trained Vector Weight (TVW), en el cual se aprenderan un total de 6· embeding_dim parametros.

Essto equivale a introducir la siguiente linea de código en el método _init_() de la clase **CBOW**:

```
1        self.position_weight = nn.Parameter(torch.tensor([[1]*embedding_dim,
         ↪  [2]*embedding_dim,[3]*embedding_dim, [3]*embedding_dim,
         ↪  [2]*embedding_dim,[1]*embedding_dim], dtype=torch.float32))
```

De primeras nos parecia poco intuitivo el modelo TVW, pero es evidente que las palabras estan representadas por embeddings y su estructura es entrenada por el propio modelo, así que no es descabellado pensar que parametrizar cada posición de las palabras de entrada será mejor. Los resultados lo corroboran, el modelo TVW ha ganado mucha precisión, tal y como se muestra en la *Tabla 3*

Modelo	**Training %**	**Validation %**	**Test %**
Base	26.2	23.9	15.4
FSW	33.6	31.4	21.4
TSW	35.4	33.3	22.8
TVW	43.4	41.3	30.6

Table 3: Comparación de acuracy del modelo TSW con los modelos anteriores

Si analizamos los pesos de las palabras de embedding podemos ver como hay algunas posiciones que se consideran más imporantes, como por ejemplo la posición 0, 9 i 50. También podemos apreciar como hay valores altos (oscuros) en una posición muy especifica de un embedding mientras que en los otros se mantienen valores más bajos (claros), como seria el caso de la posición 69 de la pablabra 3. Esto nos puede sugerir un overfitting, que puede llevar a problemas cuando salgamos de los casos de training.

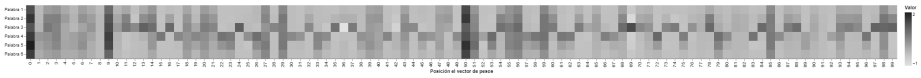

Figure 3: Pesos de los embeddings de las palabras de contexto

2.4 Hyperparameter optimization

En este apartado, estudiaremos los parámetros con los cuales estamos entrenando a los modelos. Como modelo base usaremos el TVW.

2.4.1 Embedding size

El tamaño de embedding juega un papel crucial en el rendimiento y la eficiencia computacional de los modelos. Exploraremos cómo ajustar este parámetro para optimizar el rendimiento del modelo.

Los tamaños de las matrices **E** y **W** dependen directamente del tamaño de embedding, por lo tanto el número de parametros a entrenar también se verá afectado.

Hemos probado con dos embeddings distintos, uno muy pequeño y otro muy grande para poder ver bien las implicaciones en el tiempo de ejecución y la eficiencia.

Emb size	**Train %**	**Valid %**	**Test %**	**Tiempo** (hora:min)
10	30.0	27.2	21.3	1:22
100	43.4	41.3	30.6	2.15
1000	48.9	45.0	33.3	10:46

Table 4: Comparación de acuracy y tiempo de ejecución con distintos tamaños de embedding

Podemos concluir que incrementar el tamaño del embedding resultará en una mayor precisión, dado que el modelo será más capaz al tener más parámetros. Sin embargo, nos vemos limitados por el tiempo de ejecución de Kaggle, que es de 12 horas, por lo que no podemos aumentarlo demasiado. Además, es importante tener en cuenta que el crecimiento no es lineal en órdenes de magnitud. Por ejemplo, el cambio en precisión y el tiempo necesario para entrenar se duplican al aumentar el tamaño del embedding de 10 a 100, pero al pasar de 100 a 1000, el cambio en precisión, aunque notable, es menor y el tiempo de entrenamiento se multiplica por 5.

2.4.2 Batch size

El tamaño del batch size afecta la velocidad de convergencia y la estabilidad del entrenamiento. En esta sección, discutiremos cómo seleccionar el tamaño del lote óptimo para mejorar el proceso de entrenamiento.

En este caso, la relación con el tiempo de ejecución es claro, cuanto mayor sea el batch size, más rápido entrenará los datos, però los saltos de entrenamiento

serán más abruptos.

Para ver diferentes comportamientos hemos escogido, de nuevo, valores extremos, pero hemos tenido una limitación: la RAM que permite el Kaggle. Pues valores muy altos no son procesados.

Batch size	**Train %**	**Valid %**	**Test %**	**Tiempo** (hora:min)
100	40.7	38.1	29.3	6:03
1000	43.4	41.3	30.6	2.15
5000	43.0	41.1	30.6	1:52

Table 5: Comparación de acuracy y tiempo de ejecución con distintos tamaños de batch

Podemos concluir que a medida que disminuye el batch size, el tiempo de ejecución tiende a aumentar, sin que esto conlleve una mejora en la precisión. De hecho, observamos un fenómeno contrario: a menor batch size, la precisión en test disminuye. Por otro lado, al aumentar el batch size, dentro de los límites permitidos por la memoria RAM de Kaggle, el tiempo de ejecución disminuye sin que se vea afectada la precisión.

2.4.3 Learning rate

El learning rate influye en la rapidez con la que un modelo converge durante el entrenamiento. Analizaremos valores del learning rate y su eficiencia en el entrenamiento y test.

Probaremos valores de diferentes ordenes para ver cuales funcionan mejor en el módelo. Hemos podido observar que un learning rate de 0.001 es el que maximiza la precisión y converge de forma más rápida. Los resultados se pueden apreciar en la *Tabla 6*.

Learning Rate	**Train %**	**Valid %**	**Test %**
0.01	42.1	39.6	29.8
0.001	43.4	41.3	30.6
0.0001	38.4	36.9	27.3

Table 6: Comparación de acuracy con distintos learning rates

2.4.4 Number of epochs

El número de epochs determina la cantidad de veces que el modelo verá el conjunto de datos durante el entrenamiento. En esta sección, consideraremos

cómo seleccionar el número óptimo de épocas para evitar el sobreajuste o el subajuste del modelo.

Epochs	Train %	Valid %	Test %
1	34.1	37.5	27.9
2	40.8	39.7	29.6
3	42.2	40.5	30.4
4	43.0	41.0	30.8
5	43.5	41.3	31.0
6	43.8	41.5	31.0
7	44.1	41.8	31.5
8	44.3	41.9	31.5
9	44.4	41.9	31.4
10	44.6	42.0	31.6

Table 7: Comparación de acuracy con distintos learning rates

En la *Tabla 7*, observamos que conforme aumenta el número de épocas, la precisión tiende a incrementarse, aunque no de manera uniforme. Se percibe que a partir de las 7 épocas, la precisión en el test se estabiliza alrededor del 31.5%. Sin embargo, es importante tener en cuenta que el incremento en el número de épocas está directamente asociado con un aumento en el tiempo de ejecución.

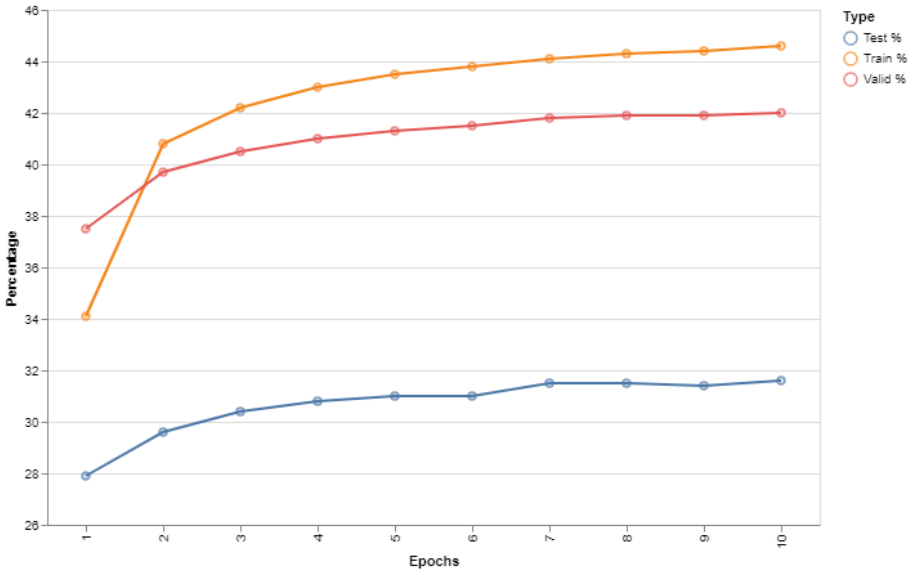

Figure 4: Precisión en entrenamiento, validación y prueba por época

En la *Figura 4*, podemos observar de manera más clara la tendencia que

estamos discutiendo. A partir de la época 7, se observa un aumento en la precisión del entrenamiento, pero no en el test, lo cual sugiere la presencia de un posible sobreajuste (overfitting). Esto significa que el modelo está aprendiendo demasiado los detalles específicos de los datos de entrenamiento y no generaliza bien a nuevos datos.

2.4.5 Sharing input/output embeddings

Compartir embeddings de entrada/salida puede ser beneficioso en ciertos contextos para mejorar la eficiencia y la generalización del modelo. Exploraremos cómo implementar esta técnica y sus implicaciones en el rendimiento del modelo.

2.5 Other models

Se puede apreciar que las dimensiones de la matriz de embeddings y de la matriz lineal son inversas entre sí, lo que sugiere que nuestro modelo se orienta hacia la obtención de dos representaciones distintas de los embeddings de palabras. Con este entendimiento, hemos optado por la estrategia de "weight tying", en la cual los pesos de ambas matrices se unifican. Específicamente, esto implica que la matriz lineal se convierte en la transposición de la matriz de embeddings. Este enfoque promete no solo mejorar la capacidad de generalización del modelo, sino también reducir el tiempo necesario para su entrenamiento.

La implementación de esta técnica requiere una modificación específica en el proceso de 'forward pass'. Esto se traduce en ajustar cómo los datos fluyen a través del modelo durante la fase de entrenamiento, asegurando que las actualizaciones de pesos se realicen de manera coherente con nuestra nueva configuración de "weight tying".

```
 def forward(self, input):
# input shape is (B, W)
e = self.emb(input)
# e shape is (B, W, E)
u = (e*self.position_weight.view(1,-1,e.size(-1))).sum(dim=1)
# Here, F.linear function is used to apply the linear transformation
z = F.linear(u, self.emb.weight)
# z shape is (B, V)
return z
```

Configuración	Tiempo de Entrenamiento (s)	Precisión de Prueba
Parámetros No Compartidos	8000	0.3003
Parámetros Compartidos	7000	0.255

Table 8: Comparación del rendimiento del modelo con parámetros compartidos y no compartidos.

Aunque el tiempo de entrenamiento se reduce con parámetros compartidos, esta ventaja no compensa la pérdida de precisión en el conjunto de prueba. Esto sugiere que el modelo, con pesos independientes, capta mejor el significado de las palabras, haciendo que el ahorro de tiempo no compense el decremento en rendimiento.

3 Analyze the generated word vectors

El objetivo de esta sección es comprender cómo representaciones vectoriales de palabras pueden capturar relaciones semánticas y sintácticas dentro de un corpus de texto.

3.1 WordVector Class

La classe WordVectors que hemos implementado es una estructura de datos diseñada para gestionar y operar sobre vectores de palabras. Estos vectores, también conocidos como embeddings, son representaciones numéricas de palabras en un espacio de alta dimensión(en nuestro caso 100) donde palabras con significados o contextos similares tienden a estar más cerca entre sí. Esta clase ofrece funcionalidades para calcular similitudes entre palabras y encontrar analogías.

3.1.1 most_similar

Busca las palabras más similares a una palabra dada, basándose en la similitud coseno de sus vectores. Cabe recalcar que ignora la palabra objetivo estableciendo su similitud a 0 para evitar que se devuelva a sí misma como la más similar.

```
def most_similar(self, word, topn=10):

        word_index = self.vocabulary.get_index(word)
        word_vec = self.vectors[word_index]
        similarities = np.array([self._cosine_similarity(word_vec, vec) for vec
        ↪  in self.vectors])
        similarities[word_index] = 0
        most_similar_indices = similarities.argsort()[-topn-1:-1][::-1]

        return [(self.vocabulary.get_token(i), similarities[i]) for i in
        ↪  most_similar_indices]
```

3.1.2 analogies

Resuelve analogías del tipo ”x1 está a x2 como y1 está a ?” utilizando los vectores de palabras. Este método se basa en la idea de que las relaciones semánticas entre palabras pueden representarse como operaciones vectoriales.

Por ejemplo, "rey" está a "reina" como "hombre" está a "mujer" se modela encontrando el vector que resulta de la operación vector("reina") - vector("rey") + vector("hombre") y buscando el embedding más cercano a este resultado.

```
def analogy(self, x1, x2, y1, topn=5, keep_all=False):

        x1_vec = self.vectors[self.vocabulary.get_index(x1)]
        x2_vec = self.vectors[self.vocabulary.get_index(x2)]
        y1_vec = self.vectors[self.vocabulary.get_index(y1)]

        analogy_vec = y1_vec - x1_vec + x2_vec
        similarities = np.array([self._cosine_similarity(analogy_vec, vec) for
        ↪  vec in self.vectors])

        if not keep_all:
            for word in [x1, x2, y1]:
                word_index = self.vocabulary.get_index(word)
                similarities[word_index] = 0

        most_similar_indices = similarities.argsort()[-topn:][::-1]
        return [(self.vocabulary.get_token(i), similarities[i]) for i in
        ↪  most_similar_indices]
```

3.2 Intrinsic Evaluation

Ahora utilizaremos la evaluación íntrinsica como método para para medir la calidad de las representaciones vectoriales de las palabras. Esta evaluación se basa en cuantificar la similitud entre palabras y analogias y comprovar que coincide con la similitud real semántica de las palabras.

```
model1.most_similar('català')
```

Idioma	Similitud
Valencià	0.8949
Basc	0.8039
Mallorquí	0.8004
Gallec	0.7902
Navarrès	0.7643
Francès	0.7365
Neerlandès	0.7226
Rossellonès	0.7097
Aragonès	0.7074
Andalús	0.6925

Table 9: Similitud de idiomas

Podemos ver que la similitud del coseno obtiene las palabras más similares a la nuestra en términos de su campo semántico. En este caso, el campo semántico de 'català' es el idioma.

También podemos canviar la palabra para obtener otras de un campo semántico diferente, como el de regiones o paises.

```
model2.most_similar('Catalunya')
```

Región	Similitud
Turquia	0.8549
Portugal	0.8517
Mèxic	0.8505
València	0.8465
Grècia	0.8390
Europa	0.8344
Brasil	0.8297
Rússia	0.8287
Bèlgica	0.8257
Girona	0.8252

Table 10: Similitud de regiones

Aunque que Turquia sea la palabra más parecida a Catalunya no tiene un sentido semántico podemos ver que en global las palabras más parecidas si que estan del mismo campo semántico.
Luego, tambien podemos hacer una evaluación intrínseca midiendo las analogias que se nos ocurran.

```
model1.analogy('home', 'dona', 'rei')
```

Palabra	**Similitud**
reina	0.603
princesa	0.581
regent	0.573
tsarina	0.536

Table 11: Puntuación de analogias

No obstante, los embeddings aprenden el significado de las palabras de los textos, pero también replican sesgos y estereotipos presentes en estos. Aunque

pueden modelar similitudes relacionales, como en la analogía hombre:mujer::rey:reina, también reflejan estereotipos de género. Aquí tenemos un ejemplo.

```
model1.analogy('home', 'dona', 'banquer')
```

Palabra	**Similitud**
cuinera	0.584
cineasta	0.563
metgessa	0.562
psicoanalista	0.560

Table 12: Puntuación de analogias sesgadas

3.3 Clustering Properties

Para evaluar la calidad de las representaciones vectoriales de palabras, o embeddings, una técnica eficaz es el clustering visual. Este enfoque implica proyectar los embeddings de palabras en un espacio bidimensional (2-D) para observar cómo se agrupan visualmente. El objetivo es comprobar si la disposición espacial de las palabras refleja adecuadamente sus relaciones semánticas, es decir, si palabras con significados similares se ubican cercanamente en el espacio visual.

El método de clustering que hemos aplicado inicialmente es el Análisis de Componentes Principales(PCA)

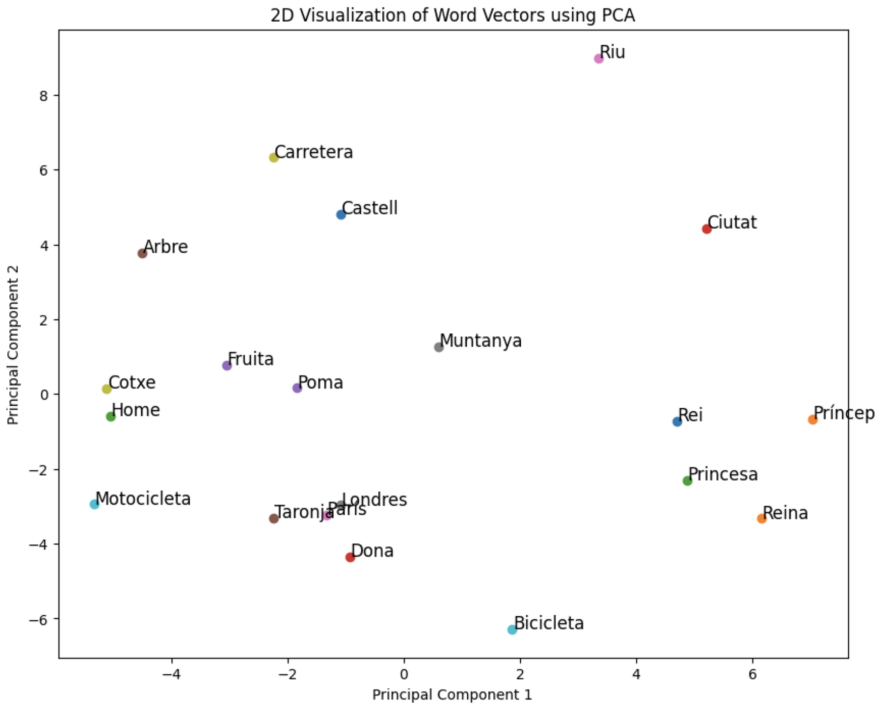

Figure 5: PCA clustering.

En la imagen podemos apreciar como las palabras similares(están dentro del mismo campo semántico) estan más juntas que palabras que no tienen ninguna relación entre si. Un ejemplo muy claro es de Rei, Princesa, Reina, Príncep que estan muy cerca todos ya que estan dentro del mismo campo semántico de 'familia real'.

En la imagen tambien podemos ver como otras palabras que estan dentro del mismo campo semántico estan más juntas. Por ejemplo, Fruita y Poma. O Londres Y Paris.

Tmbien podemos ver en la imagen una representacion de lo que hemos comentado anteriormente, que rei-reina = home-dona

Podemos tambien mostrar en la imagen la similidad de diferentes palabras a una palabra central.

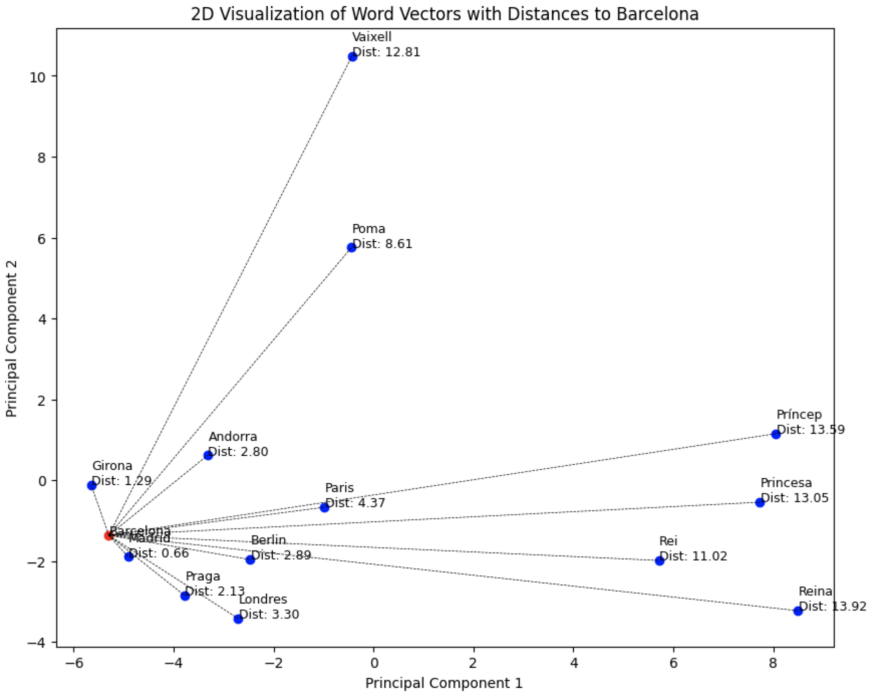

Figure 6: PCA clustering.

En esta imagen podemos ver que la palabras mas similares a Barcelona son esas de su mismo campo semántico(ciudades europeas) y forman un cluster. Por lo contrario, otras palabras, como 'Poma', que no guardan ninguna relacion con nuestra palabra, estan mas alejadas.

3.4 Prediction accuracy

La precisión en la predicción de un modelo de embeddings se basa en la capacidad del modelo para calcular la palabra dado el contexto de un texto que nunca ha visto antes. Hemos utilizado dos modelos para calcular la precisión de predicción. El primer modelo es el que implementamos con pesos [1,2,3,3,2,1] para las palabras de contexto en el entrenamiento CBOW. El otro modelo es aquel en el cual, en lugar de un peso para cada palabra de contexto, optimizamos un vector para cada posición. Aquí podemos ver los resultados:

Modelo	**Precisión**
Primer modelo(pesos fijados)	0.21
Segundo modelo(pesos aprendidos)	0.303

Table 13: Puntuación de modelos de embeddings

Podemos observar que el segundo modelo muestra una mayor capacidad de predicción que el primero. Esto puede atribuirse a la optimización de vectores individuales para cada posición de las palabras de contexto, lo que permite al modelo capturar y aprender relaciones más ricas y variadas entre las palabras, mejorando así su habilidad para anticipar correctamente la palabra siguiente en diferentes situaciones y contextos.