CATALAN WORD VECTORS. TRAINING AND ANALYSIS

PROCESSAMENT DEL LLENGUATGE ORAL I ESCRIT
ASSIGNMENT 1

GRAU EN CIÈNCIA I ENGINYERIA DE DADES
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Lucía De Pineda Alabart - 49880647H

Aina Luis Vidal - 26594595V

March 2023

1 Introduction

This assignment consists on the creation, analysis and evaluation of word vectors, using a database created from *Wikipedia* articles. Word embeddings are a form of representations where each word is mapped to a numeric vector. They are key for many natural language processing systems, so it is essential to know how they work. The main ideas are that words with similar meaning should have similar vectors and the fact that the environment of a word gives meaning to it. In relation to this, the model we will use in this assignment takes into account the context words to make predictions. Specifically, we will train and improve a CBOW (Continuous bag-of-words) model and later on, analyze the word vectors generated. To do that, we will work on a Kaggle Environment, which allow us to share our work and our data and to have access to GPU resources.

2 Task 1: Improve CBOW Model

The CBOW model aims to predict a word using the context words. The standard CBOW model sums all the context word vectors with the same weight. The goal of this task is to improve the model assigning different weights to each context word vector. We will use different methods to assign these weights and evaluate the performance.

We are given a definition of a class implementing the standard CBOW model, which is the following:

```python
class CBOW(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        u = e.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

Figure 1: Definition of standard CBOW class

We are asked to modify this class in order to change the distribution of weights. We will create and analyze three different models.

2.1 Model A. Fixed scalar weight

The first improvement we are suggested to do is to compute a weighted sum of the context words with a fixed scalar weight. In other words, every position of the embedding representation of a word is multiplied by the same fixed value and consequently, only the embedding representations are learned along the training process of the model. We have decided to use $(1, 2, 3, 3, 2, 1)$ as the scalar weights, allowing words that are next to the centered word (the target one) to have more weight than the ones that occupying a distant position. The code is the following:

```python
class CBOW_fixed_weight(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1], dtype=torch.float32).view(1,6,1))

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        w = e*self.position_weight
        u = w.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

Figure 2: Code for Model A implementation

2.2 Model B. Trained scalar weight

Secondly, we are suggested to compute a weighted sum of the context words using trained scalar weights. In other words, every position of the embedding representation of a word is multiplied by the same value which is learned along the training process and consequently, not only are these values learned, but also the embedding representations. Initially, as the window size is defined to be of length 7, we have decided to use a tensor having random values of 6 positions. The implementation is:

```
class CBOW_trained_scalar_weight(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.position_weight = nn.Parameter(torch.rand(1,6,1))

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        w = e*self.position_weight
        u = w.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

Figure 3: Code for Model B implementation

2.3 Model C. Trained vector weight

Finally, we are suggested to compute a weighted sum of the context words using trained vector weights. In other words, every position of the embedding representation of a word is multiplied by a value which is learned along the training process. In this case, each position of each embedding representation can have associated a different weight and consequently, during the training process not only are the embedding representation values learned, but also the 6x100 (*window_size-1* x *embedding_dim*) weights. Initially, we have decided to use a tensor of dimensions $(6, 100)$ having random values. The result modification is:

```
class CBOW_trained_vector_weight(nn.Module):
    def __init__(self, num_embeddings, embedding_dim):
        super().__init__()
        self.emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        self.position_weight = nn.Parameter(torch.rand(1,6,100))

    # B = Batch size
    # W = Number of context words (left + right)
    # E = embedding_dim
    # V = num_embeddings (number of words)
    def forward(self, input):
        # input shape is (B, W)
        e = self.emb(input)
        # e shape is (B, W, E)
        w = e*self.position_weight
        u = w.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

Figure 4: Code for Model C implementation

2.4 Evaluation of the models

Once we have implemented the three different models, we evaluate their accuracy. To do that, we train and validate the models with a dataset from *Wikipedia* and then use another validation dataset from *El Periódico*. First of all, we analyze the accuracy of the standard CBOW model:

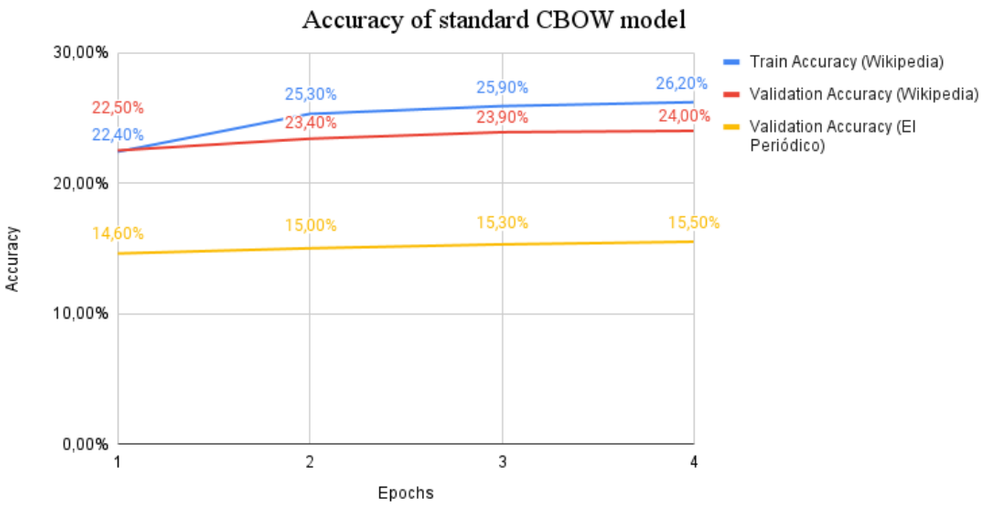

Figure 5: Accuracy of standard CBOW model

We can see that the validation accuracy of the Wikipedia dataset reaches 24% in the last epoch, while the accuracy of *El Periódico* dataset goes up to 15,5%.

Afterwards, we evaluate the accuracy of the Model A (fixed scalar weights):

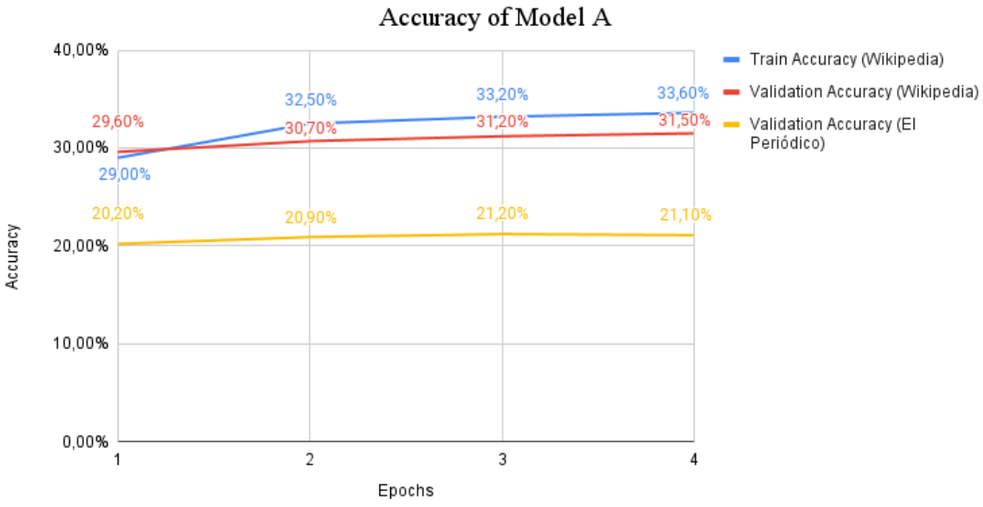

Figure 6: Accuracy of Model A

In this case, the validation accuracy of the Wikipedia dataset gets to 31,5% and 21,1% for the *El Periódico* dataset. We can see an improvement of approximately 6% for both datasets when we use fixed scalar weights, which is what we would expect when giving more importance to the context words closest to the input.

Next, we have Model B, which uses trained scalar weights for each position. The evaluated accuracy is:

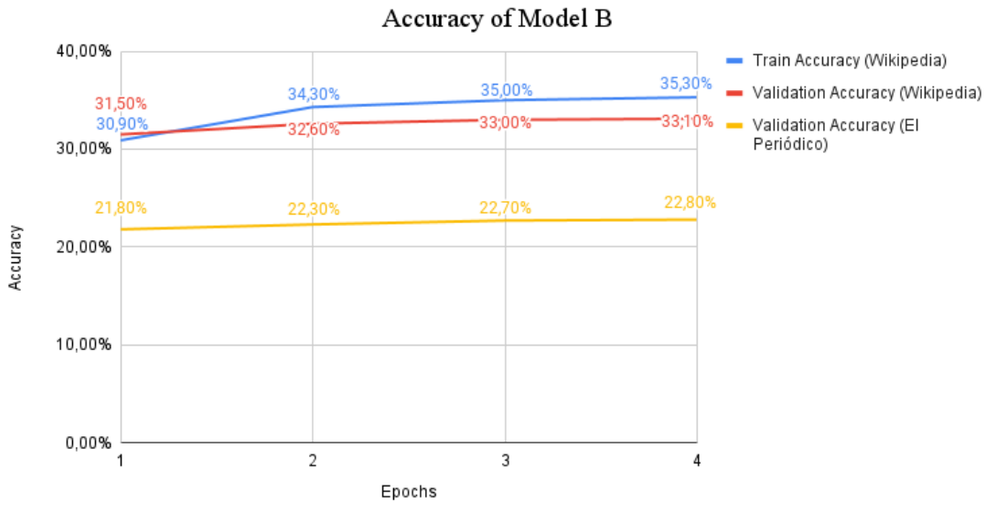

Figure 7: Accuracy of Model B

With this model the accuracy increases even more, going up to 33,1% for the validation accuracy of the Wikipedia dataset and 22,8% for *El Periódico*'s. This increment in the performance is due to the training of the weights, which reduces the loss and increases the accuracy at each iteration.

We have considered of great interest to analyse which are the values of the weights that the model has learned along the training process. As we have initialized the weights as random values, we expect that the weights converge to some values such that the words closer to the word to predict have more weight than the ones occupying more distant positions. In the following graph we can see how the values converge.

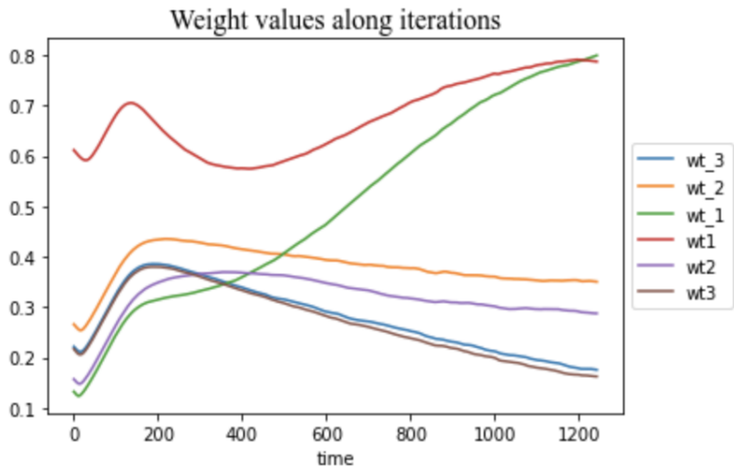

Figure 8: Weight values along iterations

We would like to note that we have had a little problem because we have run out of GPU minutes and, as a consequence, we have run the training of the model printing the weights using a CPU. We have not been able to see which are the values of the weights along every iteration and on every epoch. However, we can draw many conclusions. Firstly, as expected, the weights corresponding to the words closest to the one we want to predict are given more importance, having values around 0.8, and when going away from the center, the weights are reduced having values of 0.35 when we move 2 position away and 0.18 when moving 3 positions. In addition, we can see a symmetric behaviour as the weights w_{t-3}, w_{t-2} and w_{t-1} are similar to w_{t+3}, w_{t+2} and w_{t+1}, respectively.

Lastly, we analyze the last model, which uses a trained vector weight for each position.

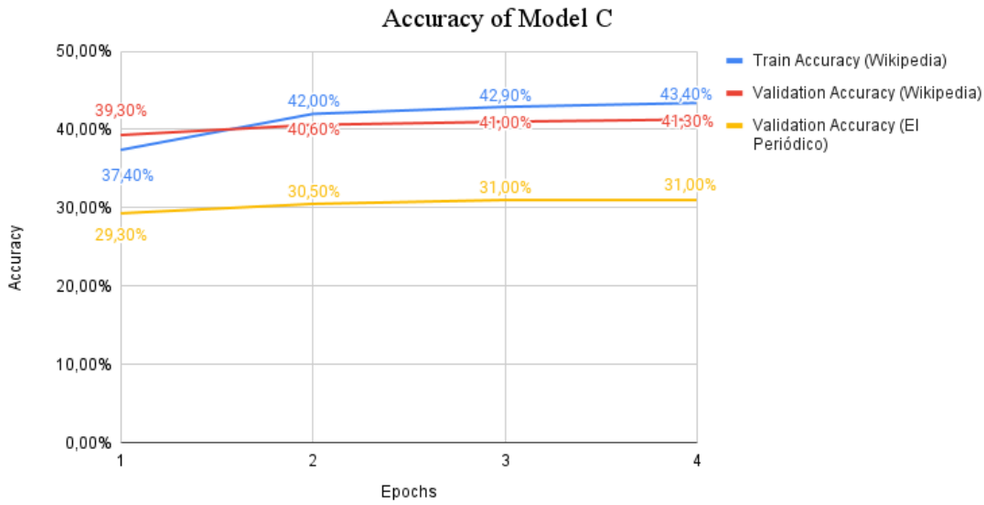

Figure 9: Accuracy of Model C

This model presents the highest accuracy of all the models, as not only it trains the weights for each specific context word, but also for each embedding position. The validation accuracy for Wikipedia and *El Periódico* datasets reach 41,3% and 31% accuracy, respectively. However, the computation cost is also more expensive.

2.5 Use of hyperparameter optimization

As optional work, we study the performance of the CBOW model as a function of one of its parameters. We choose the parameter of embedding size and we will use the Model C, as it is the one with the best accuracy. The embedding size we worked with in the previous sections is 100, so we will analyze the performance with embedding size 50, 150 and 200:

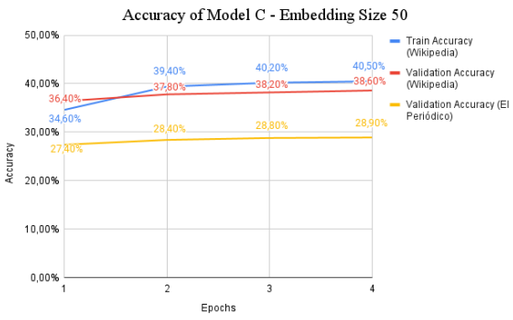

Figure 10: Accuracy with embedding size 50

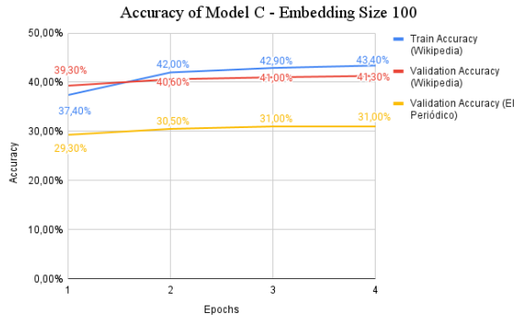

Figure 11: Accuracy with embedding size 100

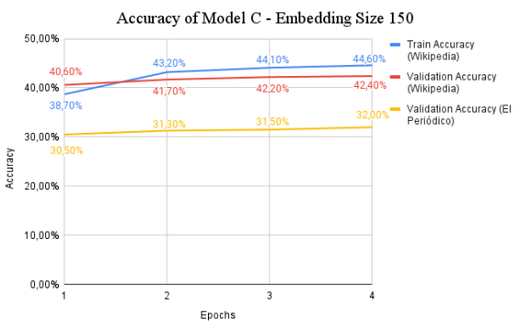

Figure 12: Accuracy with embedding size 150

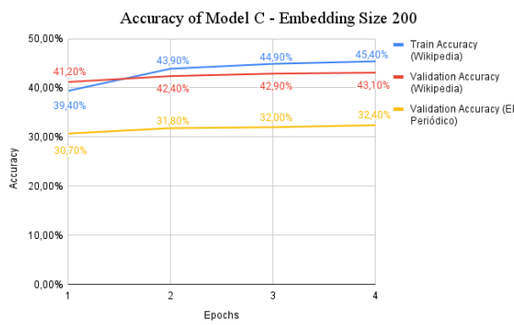

Figure 13: Accuracy with embedding size 200

What we can conclude looking at these plots is the fact that the bigger the embedding size, the higher the accuracy. Increasing the embedding size from 100 to 200 causes a rise of approximately 2% of the accuracy. However, it should also be noted that the execution time is also significantly higher as the embedding size grows. Therefore, both things should be taken into consideration when choosing the embedding size, depending on our interests. In addition, more hyperparameters (such as batch size, number of epochs or learning rate) could be studied to improve the performance of the model.

In section 3.4, we have explained why these values of accuracy make sense. Also, we have submitted our best trained model, which is the one obtained using `Model C`, to the Kaggle competition, in order to see which accuracy we obtain when we use an out-of-domain test set (*El Periódico*).

3 Task 2: Word Vectors Evaluation

The second task consists on analyzing the word vectors generated by the CBOW model. To do that, we implement the *WordVector* class, defining two methods: *most_similar* method to find the closest vectors and *analogies* to find word analogies. We will also work with the *cosine_similarity* and *operation* methods, which give us crucial information about vectors. Moreover, to generate the word vectors we will use the CBOW model with the highest accuracy, which is Model C (as we demonstrated before), as it will produce better vectors.

In this task, there are two possible models to be used: one that uses the *input word vectors* and a second one that uses the *output word vectors* as representation of the words, both obtained with the training process of the CBOW model. While the former corresponds to the embedding matrix which contains the vector representations of words in a high-dimensional space, the latter is the linear transformation matrix which is used to transform the vector representation of the input words into a probability distribution over the whole vocabulary.

On the one hand, it is true that the embedding matrix clearly has the representation of each word in the vocabulary. It is trained to keep the semantic and syntactic relationships between words as the aim of the CBOW model is to adjust this matrix in a way that we predict the co-occurrence patterns. On the other hand, the linear matrix is just a transformation in the dimensional space and it is not trained to capture the semantic and syntactic relationships between words directly. However, this matrix can also be seen as a version of the embedding representations and although it has more sense for us to use the embedding matrix for computing similar words and analogies, using the linear one can also give good results, specially when computing analogies.

3.1 Most_similar method

This method aims to find the closest vectors to a specific vector. Therefore, it is supposed to return similar words to the input. The implementation is the following:

```
def most_similar(self, word, topn=10):
    word_id = self.vocabulary.get_index(word)
    sim_word = []
    for w2_id in range (self.voc_size):
        if w2_id != word_id:
            sim_word.append((self.vocabulary.get_token(w2_id), self.cosine_similarity(
                self.vectors[word_id], self.vectors[w2_id])))
    sim_word.sort(key = lambda x: -x[1])
    return sim_word[0:topn]
```

Figure 14: Implementation of most_similar method

The function is quite simple, first it computes the cosine similarity of the input word with each word from the vocabulary. Then, it sorts them and only returns the top N words, which is a parameter of the function (the default is 10).
To evaluate its performance, we try different words and analyze the results, which are represented in Figure 15. We have considered the words *català, casa* and *banc*. As we can see, when we use `Model 1` (input word vectors), for the first two examples the results are excellent, as the words returned are really similar in meaning. However, for the third example, as it is a word with multiple meanings, we obtain a more diverse output. The results make sense but they refer to different meanings of the word. On the other hand, when using `Model 2` (output word vectors), just for the first example the results are correct. For both the *casa* and *banc* words, the returned solution is not correct. As a conclusion, using `Model 1` allows us to obtain better solutions.

```
model1.most_similar('català', 5)
```

```
[('valencià', 0.8966472396059136),
 ('basc', 0.8486368720995038),
 ('gallec', 0.7824255608434053),
 ('mallorquí', 0.7668336708113344),
 ('castellà', 0.7572442597077931)]
```

```
model2.most_similar('català', 5)
```

```
[('francès', 0.9044073668905137),
 ('anglès', 0.8977542690649529),
 ('espanyol', 0.8971587870220512),
 ('alemany', 0.8933394091887107),
 ('castellà', 0.8753904387013661)]
```

```
model1.most_similar('casa', 5)
```

```
[('finca', 0.7651913984842528),
 ('masia', 0.7227804951501101),
 ('barraca', 0.6704054216869894),
 ('mansió', 0.6560298009694184),
 ('cabana', 0.652651306609716)]
```

```
model2.most_similar('casa', 5)
```

```
[('Roma', 0.8512310563115476),
 ('ciutat', 0.8459281768182569),
 ('propietat', 0.8358765003548796),
 ('dona', 0.8340125656262013),
 ('Rússia', 0.8230945573975121)]
```

```
model1.most_similar('banc', 5)
```

```
[('buc', 0.670424478907233),
 ('moll', 0.6641065722338934),
 ('tanc', 0.655862803217196),
 ('corredor', 0.6552406056554092),
 ('zoo', 0.6444781640060616)]
```

```
model2.most_similar('banc', 5)
```

```
[('corredor', 0.9174292408547489),
 ('bloc', 0.9044860993928407),
 ('desert', 0.9006056210003588),
 ('tribunal', 0.9005192048737988),
 ('magatzem', 0.8987332491977925)]
```

Figure 15: Most_similar method examples

3.2 Analogies method

The *analogies* method finds analogies between words using the cosine similarity and the difference between vectors. The code for the function is:

```python
def analogy(self, x1, x2, y1, topn=5, keep_all=False):
    # If keep_all is False we remove the input words (x1, x2, y1) from the returned closed words
    x1_id = self.vocabulary.get_index(x1)
    x2_id = self.vocabulary.get_index(x2)
    y1_id = self.vocabulary.get_index(y1)
    vec_x1 = self.vectors[x1_id]
    vec_x2 = self.vectors[x2_id]
    vec_y1 = self.vectors[y1_id]
    dif_vec = self.operation(vec_x2, vec_x1, 'sub')
    vec_ref = self.operation(dif_vec, vec_y1, 'sum')
    analogy_word = []
    for y2_id in range (self.voc_size):
        if keep_all or (y2_id!=x1_id and y2_id!=x2_id and y2_id!=y1_id):
            vec_y2 = self.vectors[y2_id]
            analogy_word.append((self.vocabulary.get_token(y2_id), self.cosine_similarity(vec_ref, vec_y2)))
    analogy_word.sort(key = lambda x: -x[1])
    return analogy_word[0:topn]
```

Figure 16: Implementation of analogies method

This function takes 3 words as mandatory input parameters and 2 more additional. The first 3 words are named `x1, x2, y1` and we want to obtain the word `y2` such that is at the same distance from `y1` as `x2` is from `x1`. The 2 other parameters refer to the number of words we want to obtain (the top N more similar words that fulfill the condition) and whether we can include the words `x1, x2, y1` in the returned words.

To do so, we have chosen to look for the word `y2` which is the one closest to the representation of the embedding obtained when adding the difference between `x1` and `x2` to `y1`. In other words, if the distance between the two vectors is understood as a transformation vector, we want to obtain closest to the point we reach when we apply this transformation vector to the `y1` word. So, firstly the difference vector between words `x1` and `x2` is computed and then, it is summed to the `y1` word, obtaining the

`vec_ref` which is the exact representation we obtain when be apply the same difference that exists between `x1` and `x2` to `y1`. Then, for every possible word `y2` we compute the cosine similarity between itself and `vec_ref` and we finally return the top N words having the highest similarity.

To evaluate the performance of the *analogy* method, we try different words and analyze the results, which are represented in Figure 18.

```
model1.analogy('França', 'francès', 'Polònia')
```

```
[('polonès', 0.7302620427345439),
 ('rus', 0.7294077928728961),
 ('suec', 0.7001087547228407),
 ('romanès', 0.6946904401115087),
 ('alemany', 0.6920949504835143)]
```

```
model2.analogy('França', 'francès', 'Polònia')
```

```
[('polonès', 0.9190087000477082),
 ('rus', 0.9039990033813486),
 ('suec', 0.8966731891423321),
 ('romanès', 0.8924005179177729),
 ('basc', 0.892397117080707)]
```

```
model1.analogy('un', 'dos', 'tres')
```

```
[('cinc', 0.7621840906492645),
 ('quatre', 0.7490343912650385),
 ('sis', 0.7413847423712532),
 ('vuit', 0.73054059664786),
 ('dotze', 0.7211573624048647)]
```

```
model2.analogy('un', 'dos', 'tres')
```

```
[('dues', 0.8936959905472144),
 ('quatre', 0.889503843418989),
 ('set', 0.8464327196388995),
 ('4', 0.8413765528566006),
 ('10', 0.8401214338331634)]
```

```
model1.analogy('dilluns', 'dimarts', 'dimecres')
```

```
[('dijous', 0.6019801514300591),
 ('divendres', 0.5973331932611301),
 ('dissabte', 0.5374500547453094),
 ('dissabtes', 0.46155273729181684),
 ('dia', 0.45135969716670593)]
```

```
model2.analogy('dilluns', 'dimarts', 'dimecres')
```

```
[('divendres', 0.9539057830883041),
 ('dijous', 0.952255944278861),
 ('dissabte', 0.9471096614130475),
 ('dissabtes', 0.9386328357351887),
 ('diumenges', 0.932970226481075)]
```

Figure 17: Analogy method examples

We have considered three examples. In the first one we would like the method to return the language of the country we are giving while in the other two examples we would like to return the next word of a sequence (numbers or day of the week). In all three cases and using any model, the solutions obtained are correct. However, we would like to highlight that the similarities obtained using `Model 2` are higher than the ones obtained when using `Model 1`. As a conclusion, as in the analogies process we are using the representation in the space and as when adding the difference vector we are doing a transformation in this dimensional space, it has more sense to use the linear transformation matrix.

3.3 Visualization of word clusters and word analogies

To finish our analysis of word vectors, we visualize the word embeddings in two dimensions. This way, we are able to study the clustering properties and how the word analogies work.

To reduce the word embeddings dimensionality (which is 100) and be able to plot them in 2D, we use PCA (Principal Components Analysis). PCA is a dimensionality-reduction method that transforms a large set of variables into a smaller one that still contains most of the information in the large set.

However, if we tried to plot all the words in our vocabulary, we were not able to see anything, as the overlap was too big and we could not obtain any conclusion. To fix that, we decided to use a short text extracted from the Internet to analyze the cluster properties. For the analogies visualization, we just chose a specific analogy and plotted it.

Starting with the word analogies, the words used were: *dos, 2, tres, 3, quatre, 4, cinc* and *5*. The idea is to see graphically if the difference between each pair of words (the two representations of each number) is similar. To do that, we represent each word with their embedding in two dimensions. The result is:

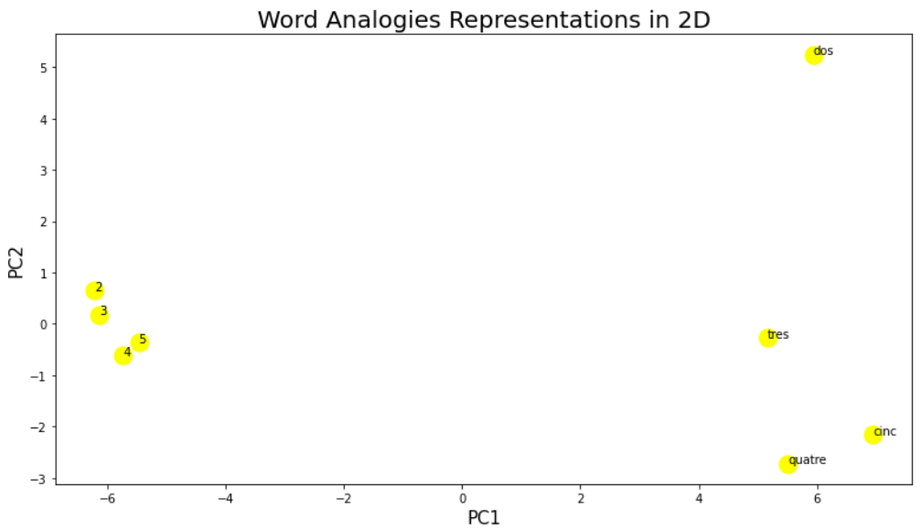

Figure 18: Word Analogies Representations in 2D

Looking at the plot we can see how the embeddings of the numbers are really similar, while the words representing the numbers are approximately close too. This is what we would expect, as similar words should have similar embeddings. Moreover, the difference between each analogy is comparable too, specially in the first principal component (PC1, the horizontal axis). Regarding the PC2, the word *dos* stands out, as the direction of the analogy is upwards, while all the others are downwards. This could be due to the fact that the original embeddings have 100 dimensions, and trying to reduce them to only two is challenging and some information is inevitably lost. Nevertheless, it works significantly well and it shows how the difference between each analogy is quite similar.

Secondly, to visualize the word clustering properties, we have selected the content of the *Wikipedia* page that explains the fable of *The Three Little Pigs* with the aim of just representing in 2D the words that are present in this fragment. The representation obtained is the following:

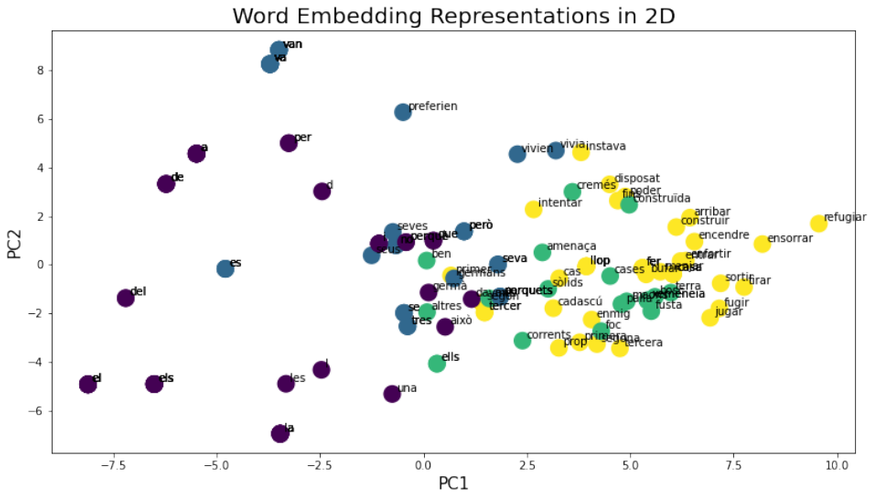

Figure 19: Word Embedding Representations in 2D

If we look carefully at the position of the words, we can say that it exists a spatial separation done considering the syntactic properties of the words as most of the verbs are in the right while pronouns and articles are in the bottom left. Also, the semantic properties have an impact over the position of the points as words having the same meaning are placed together: different conjugations of the same verb occupy a close position such as *va* - *van* (top left) or *vivien* - *vivia* (middle right) and also the

singular and plural version of the same word are also placed together as *germà* - *germans* (middle). On the other hand, if we analyse the color of the points, which represents the cluster the word belong to, we can see that colors are also visibly separated so the same characteristics mentioned before are also achieved when doing the separation of the different clusters. In conclusion, although the used embeddings are of size 100, using the PCA analysis we can see that using only 2 components we can do a good representation of the words and achieving the expected separation.

3.4 Prediction accuracy

As a final section of this assignment, we have decided to submit our best trained model, which is the one obtained using `Model C`, to the competition so as to see which accuracy we obtain when we use an out-of-domain test set (*El Periódico*). When doing so, we obtain an accuracy of the 30.35% which is approximately the same as the one we obtained using the validation test. As the accuracy has not been reduced, we don't experiment overfitting and, we can conclude that our model is good.

Overall, the values of accuracy obtained are not very high as we normally expect to have values around 80% or higher. However, in this case we are not training the CBOW model to obtain perfect predictions as it seems impossible to obtain the good prediction if we are considering more than 100.000 possible words. On the other hand, we are training the model because we are interested in the representation of the embeddings that it is learning. As we have seen when computing similar words and analogies and when doing the visualizations, the goal is achieved so we can conclude that although the accuracy is low as the predictions might not be perfect, the representations of the word vectors are good.

4 Conclusions

Taking everything we have learned in this assignment into consideration, we conclude that word embeddings are crucial for natural language processing. They give us a large amount of tools to analyze text and obtain useful insights.

Specifically, we focused on the prediction of words using its context. We learned how the CBOW model works and how we can improve it by a training process for the weights or hyperparameters tuning, among other methods. We discovered that using a trained vector weight for each context word gave us the highest prediction accuracy (around 30%), which is a reasonable result considering the large vocabulary size.

Once the word vectors were generated, we also implemented two methods (*most_similar* and *analogies*) to further study the word embeddings and the relationship between one another. We also represented them visually and found out how similar words do in fact have similar embeddings, which is what is supposed to happen. Moreover, we learned how the difference between each pair of analogies is approximately the same even in two dimensions. Finally, words have cluster properties that could lead to a further analysis.

In conclusion, this assignment has helped us to understand the behaviour of word vectors, as well as their value in natural language processing.