UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Bachelor's degree in Data Science and Engineering

Spoken and Written Language Processing - 270225

Assignment 1 - Word Vectors

Sílvia Fàbregas
Eduard Morillo

Professors:
José Adrián Rodríguez
Carlos Escolano

May 23, 2024

Contents

Motivation

Word vectors are vector representations of words. Some years ago, many NLP systems represented words as atomic units, overlooking the semantic meaning or similarity between words. One of the aims of this project is to find embeddings that encapsulate lexico-semantic properties. "Lexico" refers to the vocabulary of a language and "semantic" refers to the meaning of these words and their combination to form meaningful and coherent sentences. The objective is to develop embeddings that encode relevant information on how the words are related, and so we are going to use the context these words usually appear in, all within a deep learning framework.

The models that have been trained are rooted in the concept introduced by J.R. Firth in 1957: "You shall know a word by the company it keeps." Therefore, to construct these word vector representations, we leverage the adjacency between terms in texts. These models are Continuous Bag Of Words (CBOW) and Skip-Gram. These models utilize different strategies to construct word embeddings. They are going to be trained for a specific task, but then we will not use them for that task. The task should enable the model to construct word embeddings that demonstrate specific desirable properties. Once the word embeddings have been computed, various characteristics and attributes have been tested and evaluated. These include the visualization of linear structures and analogies, or looking for the closest words to a target given word. These evaluations provide insight into the quality and structure of these embeddings. Mainly, we want our embeddings to capture the following two aspects:

- **Word similarity:** words with similar meanings should also have similar vector representations and be close to each other in the embedding space.
- **Word analogy:** the ability to encode analogical relationships between words. For instance, allowing algebraic operations with words like `"king" - "man" + "woman" = "queen"`.

These properties shall be employed to evaluate the models intrinsically. Intrinsic evaluation refers to assessing the quality and characteristics of word embeddings directly based on their inherent properties.

CBOW model

CBOW, a variant of the word2vec model, predicts the central word based on the context words (the bag of words). Thus, by considering all words within the context window except the central one, CBOW determines the most probable word at the center. The CBOW model architecture looks as follows:

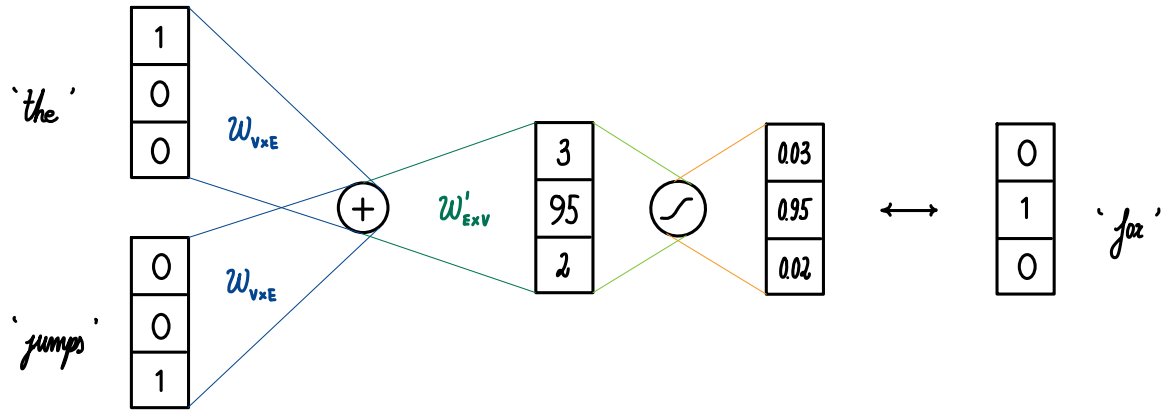

Figure 1: Diagram of the architecture of the CBOW shallow neural network, that tries to predict `fox` from the context words `the` and `jumps`

Each word has associated a fixed size vector, its embedding. It is computed as $\mathbb{1}_V \cdot W_{V\times E}$, where $W_{V\times E}$ is the embedding matrix. V the size of the vocabulary, E the embedding size and $\mathbb{1}_V$ a one-hot encoding representation of a word within the vocabulary (it selects the row associated to the word). Once the embedding of every context word has been computed, all of them are summed. The result is then multiplied by a matrix $W'_{E\times V}$, also known as the *output embedding*, to be fed into a *softmax* function, which produces a probability distribution. The word with the highest value would be selected as a prediction. The output of the *softmax* is then compared with the one-hot representation of the central word.

In Pytorch, this model can be implemented as follows:

```
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)
    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
```

In this implementation, instead of inputting the words as V-dimensional one-hot vectors, they are represented as indices $\in [0..V-1]$. It has to be noted that the *output embedding* layer has no bias, and that `lin` layer transposes `lin.weight` before performing the multiplication. The *softmax* function need not be applied, as the criterion used to compare the output with the target, `nn.CrossEntropyLoss()`, includes it.

All this been said, how will this make our embedding show the desirable properties? If two different words exhibit highly similar contexts (are usually around similar words), our model will generate similar outcomes. Therefore, under the assumption that given two similar words their contexts will probably be similar, our network will be inclined to produce similar word vectors for them.

Improving the CBOW model

The standard CBOW model sums all the context word vectors with the same weight (1). In other words, it gives equal importance to all words in the context. However, this does not necessarily have to be the case.

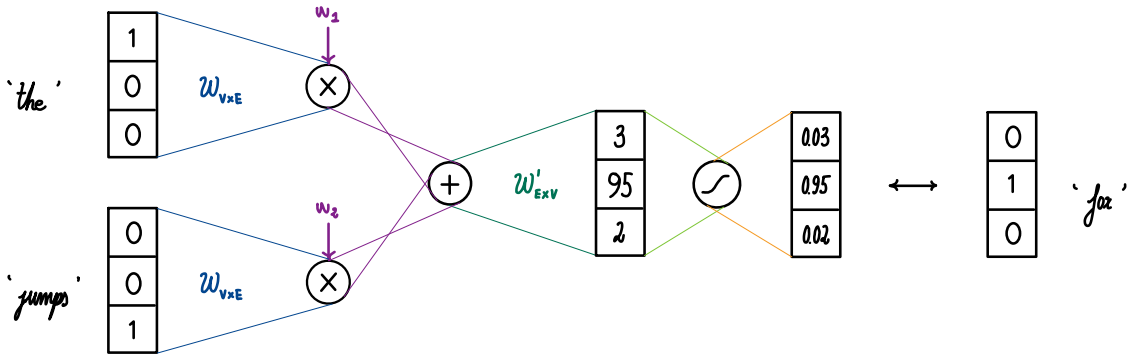

Figure 2: Diagram of the CBOW model, where each embedding is multiplied by a custom weight (w_1, w_2) before being added.

Fixed scalar weight

Multiplying the embeddings of the context words by a fixed scalar weight, $(1,2,3,3,2,1)$, will give more relevance to the words that are closer to the predicted central word. The $\times$ operator in Figure 2 would therefore correspond to scaling a vector. The implementation in Pytorch would be modified as follows.

```
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.register_buffer("position_weight",
        torch.tensor([1,2,3,3,2,1], dtype=torch.float32))
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)

```

```
    def forward(self, input):
        e = self.emb(input)
        e = torch.mul(e, self.position_weight.view(1, -1, 1))
        u = e.sum(dim=1)
        z = self.lin(u)
        return z
```

Trained scalar weight

Same rationale as with the fixed scalar weight, but the weights $w_1, w_2, \ldots, w_C$ are parameters to be optimized. The implementation would differ from the previous one in the sense that position weight would be defined as follows:

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

Trained vector weight

The weights in this case would no longer be scalars, but E-dimensional vectors to multiply with the embeddings element-wise. Therefore, the $\times$ operation would represent element-wise multiplication, $\odot$, of two vectors of the same size. The implementation in Pytorch, with all the weights initialized to 1, is to be found below.

```
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.position_weight = nn.Parameter(torch.tensor(torch.ones(1, 6,
                                        embedding_dim),dtype=torch.float32))
        self.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)

    def forward(self, input):
        e = self.emb(input)
        e = torch.mul(e, self.position_weight)
        u = e.sum(dim=1)
        z = self.lin(u)
        return z
```

Hyperparameter optimization

Finally, after having considered many combinations to weigh the input embedding vectors, hyperparameter optimization will be undergone in order to refine the models. Its performance will be studied by varying one parameter at a time, with the default ones being `embedding_dim = 100`, `batch_size = 1000`, `epochs = 4` and `optimizer = Adam`. The following list enumerates the modifications:

- `modelA` stands for the model with fixed scalar weights
- `modelB` stands for the model with trained scalar weights

- `modelC` stands for the model with trained vector weights
- `-shared` stands for using the same matrix as input and output embeddings (i.e. $W = W'$)
- `-RMSprop` stands for using the RMSprop optimizer
- `-embedX` stands for setting `embedding_dim = X`

The following table displays the train error and the validation error with two different datasets of the models trained.

Model	*Training error*	*Val. error* (Wikipedia)	*Val. error* (El Periódico)
`modelA`	33.6%	31.6%	21.8%
`modelB`	35.4%	33.1%	22.9%
`modelC`	43.3%	41.2%	31.2%
`modelB-shared`	18.2%	15.7%	9.5%
`modelC-shared`	39.5%	37.2%	27.2%
`modelC-RMSprop`	40.0%	37.8%	29.0%
`modelC-embed150`	44.6%	42.4%	32.1%
`modelC-embed200`	45.4%	43.1%	32.3%
`modelC-embed300`	46.3%	43.6%	32.8%

Table 1: Accuracies obtained with the different models.

We note an increase in accuracy for model B compared to model A, and for model C compared to both A and B. The accuracy experiences a notable decrease when sharing input and output embeddings, particularly evident in the case of model B, and to a lesser extent in the case of model C. Changing the optimizer also produces a slight decay in the accuracy. Finally, increasing the embedding size improves the accuracy, although not significantly.

The Wikipedia validation error is always 2-3% below the training error. Training has been conducted using Wikipedia documents and, consequently, the language employed is very similar. However, during validation with El Periódico, accuracy drops due to differences language use.

Skip-Gram model

As an alternative to CBOW, Skip-Gram has been implemented. In Skip-Gram, each word in the vocabulary is also associated with input and output vectors. Unlike CBOW, Skip-Gram predicts the context words based on the target word and it allows for many variations.
Many architectures could be conceived, from which we drew only the last one, which include:

- Training using all word-context pairs in the dataset. This approach would inflate our dataset size by a factor of 6 (reflecting the context window size). Moreover, we are trying to predict many outputs from one very same input, which may be confusing for the model.
- The second approach involves randomly sampling a context word for each target word during training, effectively maintaining the dataset size. Nevertheless, it operates on the same principle as the previous method, despite having reduced the data volume.
- The third strategy revolves around predicting a word and subsequently determining if the predicted word is part of the context. However, this method faces challenges as the context window typically spans 6 words, with frequently occurring words appearing in many of the windows. Consequently, the model may gravitate towards predicting only the most prevalent words, disregarding others.
- Introducing negative sampling. This strategy uses all the context words and some randomly sampled terms that are not part of the context. The input embedding of the word tries to be as close as possible to its context words and far away enough from the negatively sampled words.

The code has been acquired from the repository specified below and modified to our dataset and use (e.g. hyperparameters) [1].

A method for computing the cosine_similarity has been implemented which will be used at training time to validate the model by finding the most similar words to the target word post-iteration.

```
def cosine_similarity(embedding, n_valid_words=16, valid_window=100):

    all_embeddings = embedding.weight  # (n_vocab, n_embed)
    # sim = (a . b) / |a||b|
    # (1, n_vocab)
    magnitudes = all_embeddings.pow(2).sum(dim=1).sqrt().unsqueeze(0)

    # Pick validation words from 2 ranges: (0, window):
    # common words & (1000, 1000+window): uncommon words
    valid_words = random.sample(range(valid_window), n_valid_words//2) + \
    random.sample(range(1000, 1000+valid_window), n_valid_words//2)
    # (n_valid_words, 1)
    valid_words = torch.LongTensor(np.array(valid_words)).to(device)
```

[1] https://github.com/lukysummer/Skip-Gram_with_NegativeSampling_Pytorch/tree/master

```
14
15      # (n_valid_words, n_embed)
16      valid_embeddings = embedding(valid_words)
17
18      # (n_valid_words, n_embed) * (n_embed, n_vocab) -->
19      # (n_valid_words, n_vocab) / 1, n_vocab)
20      # (n_valid_words, n_vocab)
21      similarities = torch.mm(valid_embeddings, all_embeddings.t()) / magnitudes
22
23      return valid_words, similarities
```

The distribution from which to draw noise samples, $P_n(w)$, is a free parameter. In the implementation, as recommended in "Distributed Representations of Words and Phrases and their Compositionality", by Mikolov et al., the unigram distribution raised to the 3/4 power is used. It can be computed as follows.

```
freq = Counter(data[0][1])
freq_ratio = {word: (cnt/len(vocab.token2idx)) for word, cnt in freq.items()}
freq_ratio = np.array(sorted(freq_ratio.values(), reverse=True))
unigram_dist = freq_ratio / freq_ratio.sum()
noise_dist = torch.from_numpy(unigram_dist**0.75 / np.sum(unigram_dist**0.75))
```

The implementation of the model can be found below.

```
class Skip-GramNeg(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, noise_dist):
        super().__init__()

        self.num_embeddings = num_embeddings
        self.embedding_dim = embedding_dim
        self.noise_dist = noise_dist

        self.in_emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)
        self.out_emb = nn.Embedding(num_embeddings, embedding_dim, padding_idx=0)

        # Initialize both embedding tables with uniform distribution
        self.in_emb.weight.data.uniform_(-1, 1)
        self.out_emb.weight.data.uniform_(-1, 1)


    def forward_input(self, input_words):
        input_vectors = self.in_emb(input_words)
        return input_vectors  # input vector embeddings


    def forward_target(self, output_words):
        output_vectors = self.out_emb(output_words)
        return output_vectors  # output vector embeddings
```

```
25
26      def forward_noise(self, batch_size, n_samples=5):
27          noise_dist = self.noise_dist
28
29          noise_words = torch.multinomial(input       = noise_dist,
30                                          num_samples = batch_size*n_samples,
31                                          replacement = True)
32          noise_words = noise_words.to(device)
33
34          # use context matrix for embedding noise samples
35          noise_vectors = self.out_emb(noise_words).view(batch_size,
36                                  n_samples, self.embedding_dim)
37
38          return noise_vectors
```

Since we are using Negative Sampling (drawing random noise words to form incorrect target pairs), the model tries to minimize a loss which combines both the loss from the target (context) words and the loss from the noise samples from the vocabulary. The implementation is to be found below.

```
1  class NegativeSamplingLoss(nn.Module):
2      def __init__(self):
3          super().__init__()
4
5      def forward(self, input_vectors, output_vectors, noise_vectors):
6
7          batch_size, embedding_dim = input_vectors.shape
8
9          # batch of column vectors
10         input_vectors = input_vectors.view(batch_size, embedding_dim, 1)
11         # batch of row vectors
12         output_vectors = output_vectors.view(batch_size, 1, embedding_dim)
13
14         # log-sigmoid loss for correct pairs
15         out_loss = torch.bmm(output_vectors, input_vectors).sigmoid()\
16         .log().squeeze()
17
18         # log-sigmoid loss for incorrect pairs
19         noise_loss = torch.bmm(noise_vectors.neg(), input_vectors)\
20         .sigmoid().log()
21         # sum the losses over the sample of noise vectors
22         noise_loss = noise_loss.squeeze().sum(1)
23
24         # average batch loss
25         return -(out_loss + noise_loss).mean()
```

Intrinsic evaluation

In order to evaluate the performance of the distinct models, the most important aspect to consider is not the accuracy of the models, but the adequacy of the vector embedding: an intrinsic evaluation shall be performed. As previously noted, this evaluation encompasses both word similarity and word analogies.
A `WordVectors` class has been implemented with the methods `most_similar` and `analogy` to check these properties.

```
class WordVectors:
    def __init__(self, vectors, vocabulary):
        self.vocabulary = vocabulary
        self.vectors = vectors
        self.vectors_norms = np.linalg.norm(vectors, axis=1)

    def most_similar(self, word, topn=10):
        // ...

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

Methods

Most similar

Ideally, similar words should be closely positioned within the embedding space, while words with disparate semantic meanings should be distanced from each other. The method *most_similar* has been implemented in order to find the closest words in the embedding given an input word (see below).

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

    # word -> index
    word_index = self.vocabulary.get_index(word)

    # retrieve the vector
    word_vector = self.vectors[word_index]

    # compute cosine similarity and yield top(n+1) results
    cos_sim = np.dot(self.vectors[1:], word_vector)/
                    (self.vectors_norms[1:]/self.vectors_norms[word_index])
    top_indexes = np.argsort(cos_sim)[::-1][1:topn+1]

    # index -> word
    most_similar_words = []
    for index in top_indexes:
        most_similar_words.append((self.vocabulary.get_token(index + 1),
                                   cos_sim[index]))
```

```
19
20      return most_similar_words
```

Analogy

Moreover, our embedding should exhibit certain properties: it should support meaningful algebraic operations with words. Consequently, the result of adding two word vectors should yield a coherent representation. Most notably, the analogy operation should return consistent outcomes. It follows the format:

$$word_1 + word_2 - word_3 = word_4$$

See the implementation below.

```
1  def analogy(self, x1, x2, y1, topn=5, keep_all=False):
2      # If keep_all is False we remove the input words (x1, x2, y1) from the
3      returned closed words
4
5      y1_index = self.vocabulary.get_index(y1)
6      x1_index = self.vocabulary.get_index(x1)
7      x2_index = self.vocabulary.get_index(x2)
8
9      y2_vec = self.vectors[y1_index] + (self.vectors[x2_index] -
10             self.vectors[x1_index])
11
12     cos_sim = np.dot(self.vectors[1:], y2_vec)/
13                  (self.vectors_norms[1:]*np.linalg.norm(y2_vec))
14     top_indexes = np.argsort(cos_sim)[::-1][0:topn + 3*int(1 - keep_all)]
15
16     best_analogies = []
17
18     if keep_all == False:
19         j = 0
20         for index in top_indexes:
21             if (index + 1) not in [y1_index, x1_index, x2_index]:
22                 best_analogies.append((self.vocabulary.get_token(index + 1),
23                                        cos_sim[index]))
24                 j = j + 1
25             if j == topn:
26                 break
27     else:
28         for index in top_indexes:
29             best_analogies.append((self.vocabulary.get_token(index + 1),
30                                    cos_sim[index]))
31
32
33     return best_analogies
```

Good/Bad examples

We shall perform the intrinsic evaluation both on the best CBOW model (modelC-embed300) and the Skip-Gram model.
Both the performance of the Skip-Gram model and the best CBOW model at identifying the most similar words are remarkable, consistently providing results with infallible precision. They return words that exhibit either direct synonymity and closely related semantic meaning or are often found together in sentences.

- They are very good with nouns with very unequivocal meanings, like areas of knowledge. For instance, *política* returns *diplomàtica, econòmica, social, democràtica, feminista* for the Skip-Gram and *ideològica, diplomàtica, religiosa, social, militar* for the CBOW, and *farmacologia* returns *fisiologia*, *biologia*, *zoologia* and *psicologia* for both models.

- *domèstic* is most similar to *consum*, *animal*, *entreteniment* in the Skip-Gram. These are words that tend to be found next to *domèstic*. On the other hand, the CBOW model seems to find words more similar in meaning, like *casolà*, *quotidià* or *laboral*.

- Sometimes the models also return antonyms of the words. For the word *tristor*, the obtained most similar terms are *vitalitat, mental, agilitat, genialitat, predilecció* for the Skip-Gram and *nostàlgia, simpatia, fascinació, sensibilitat, frustració* for the CBOW.

- When inputted the word *noia*, the most similar words for the Skip-Gram are *nena, mare, amiga, **prostituta**, noi* and for the CBOW are *nena, noi, dona, **prostituta, bruixa***. Our model is perpetuating a common association of meaning, linking the depiction of a woman to that of a derogatory term like "whore" or "witch". The word *dona* returns *mare, muller, noia, esposa* whereas the word *home* returns *individu, noi, dona, heroi, assassí, treballador* among others. The woman is often portrayed in relation to others, such as sons, daughters, or a husband, while the man is depicted more as an individual, perhaps even in roles like a *killer*.

- Curiously, in the CBOW model, the word *Franco* is most close to *Macías, Pavía, Narváez, Lavalle, Goded*. *Francisco Macías* was an Equatorial Guinean politician who admired Franco so dearly he changed his african name to Francisco to match Franco; Pavía is the name of an italian city and also of the Pavía battle, which was faught by *Carlomagno, rey de los francos*; Narváez (Spain), Lavalle (Argentina) and Goded (Spain) where all from the military.

- Certain words carry societal stigma, which is projected into the embedding. For example, the word *embaràs* is most close with *avortament, estrès, abstinència, abús, empresonament, abandonament*, very negative words. In fact, the closest words to *tabú* in the Skip-Gram embedding are *homosexualitat, racisme, sexual*. TThis reflects significant aspects of our society and to which topics are viewed as more taboo issues.

In word similarity tasks, both models appear to be highly accurate in their performance. We shall see if they differ in producing as good analogies.

- **Gender**: *rei* − *home* + *dona*: *reina* for both Skip-Gram and CBOW.

- **Geography**: *París - França + Itàlia*: *Roma* for Skip-Gram and *Milà* for CBOW (Rome is the fourth option).

- **Verbs**: $cantava - cantar + estar$: *trobava* for Skip-Gram and *estigué* for CBOW. In fact, the second option given by CBOW is *estava*. The CBOW model seems to deal better with verbs.
- **Antonyms**: $ric - pobre + malalt$ returns *robertsita*, which is a mineral from the class of phosphate minerals, *riques, Öland, Mineralogy, Diocesis* (mix of results of CBOW and Skip-Gram). For simpler antonyms it produces better results: *nit - dia + blanc* returns *negre* and *nit - dia + comprar* returns *vendre*. This is actually an exception. Many other bad analogies may be easily found. *nit - dia + sinònim = fada* or *nit - dia + petit = petita*.
- **Cause and consequence**: $menjar - set + gana$. The Skip-Gram returns *menjar* whereas the CBOW returns a synonym of *gana, fam*. The Skip-Gram fails too. When inputted $plorar - tristor + felicitat$ the most similar word it finds is *bebè*.
- **Time**: *primavera - Març + Desembre*. The CBOW model seems to be better with verbs and time events, but it wrongly returns *tardor*. However, Skip-Gram predicts *jubilació*, which is not close at all. Another bad example would be $setmana - dilluns + mes$ which would logically return *any*. Instead, both models return other month names like *juliol, agost*, etc.
- **Curious examples**. Both the CBOW and the Skip-Gram model suggest that $plaer - home + dona = frustració$ and that *pesat - francès + espanyol = lleuger*.

In summary, while both models demonstrate proficiency in word similarity tasks by effectively grouping similar words together, they may not fully leverage algebraic properties and adding and subtracting words is often devoid of meaning.

Visualization

We employed t-SNE (t-Distributed Stochastic Neighbor Embedding) to visualize both models (Skip-Gram, see Figure 3, and CBOW, see Figure 4). We plotted 380 words. Notably, we observed distinct clusters emerging, particularly ones corresponding to months (bottom-left for Skip-Gram and top-right for CBOW) and numbers (bottom-left for Skip-Gram and top-left for CBOW). Moreover, words with the same lexical root are almost always overlapping. See for instance (Skip-Gram) *morir* and *mort*, *empresa* and *empreses*, *Sant* and *Santa*. Some others are not so coherent, as *família* and *famílies*, which are very far from one another. *famílies* is close to other plural words that refer to groups of people and *família* is next to *casa*. Moreover, most of the verbs are located at the bottom-right (Skip-Gram).
If we do the t-SNE with 1000 words (CBOW, see Figure 5) it gets more cluttered, but new clusters emerge: proper nouns (like *Pau, Carles, Francesc*), verbs in the infinitive, numbers that represent years, or other smaller clusters like country names or languages.

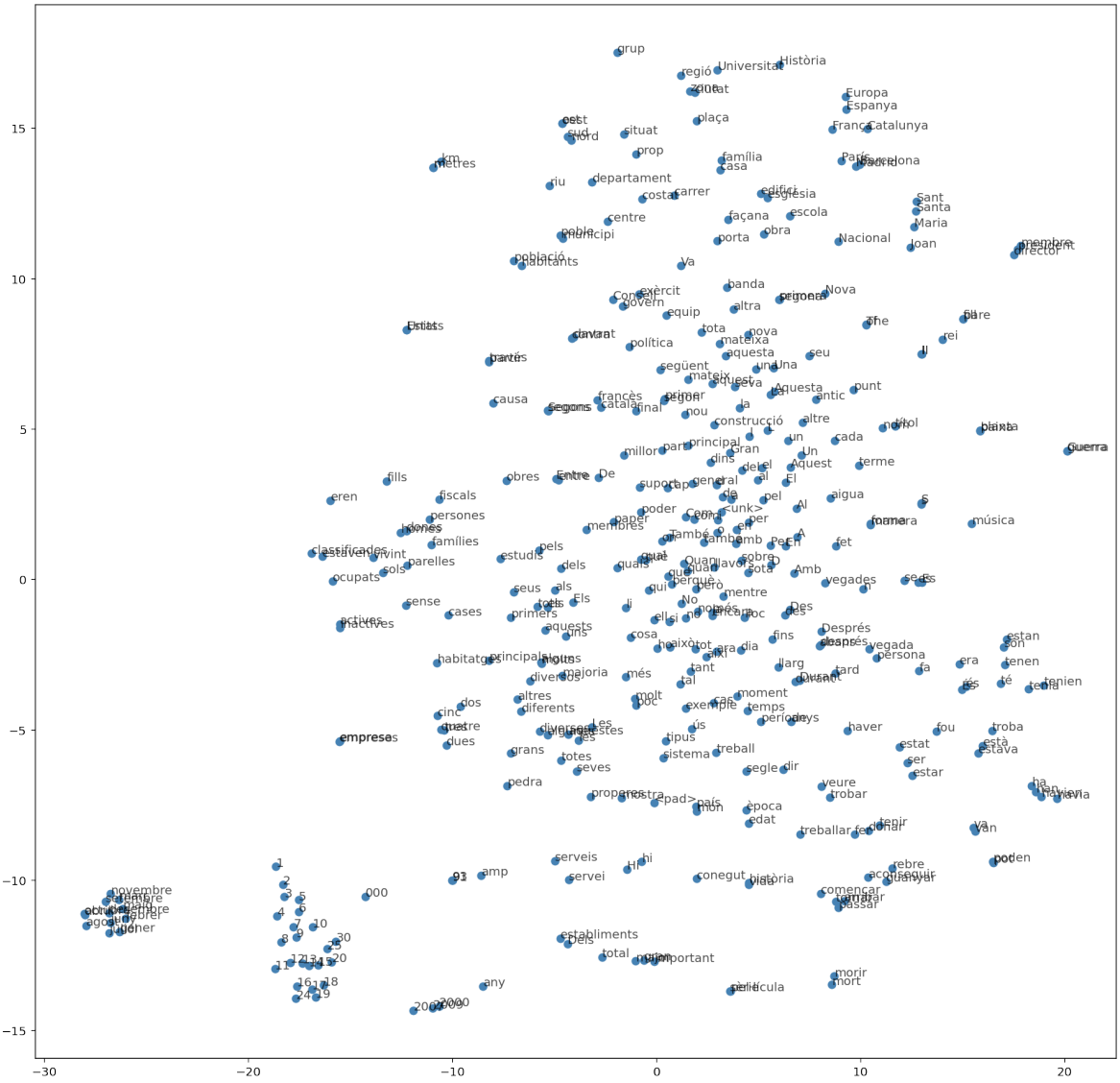

Figure 3: *t-SNE* of the embeddings computed with the Skip-Gram method implemented previously

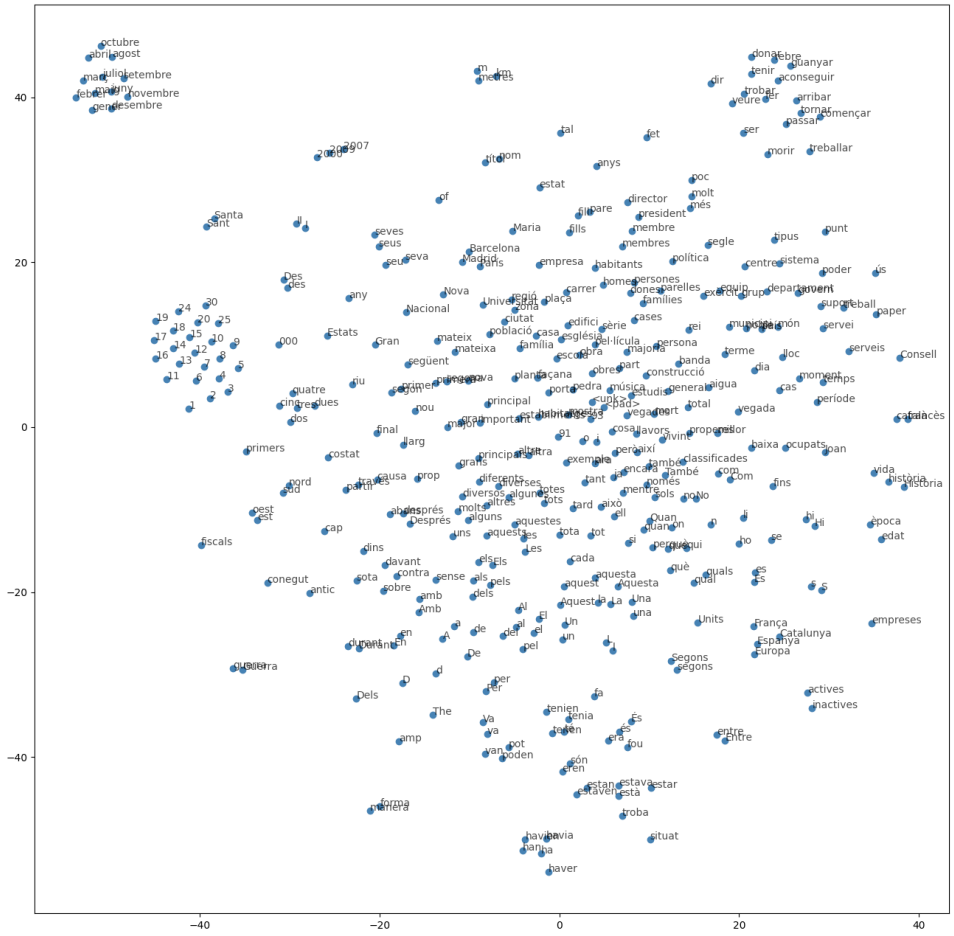

Figure 4: *t-SNE* of the embeddings computed with the CBOW method implemented previously

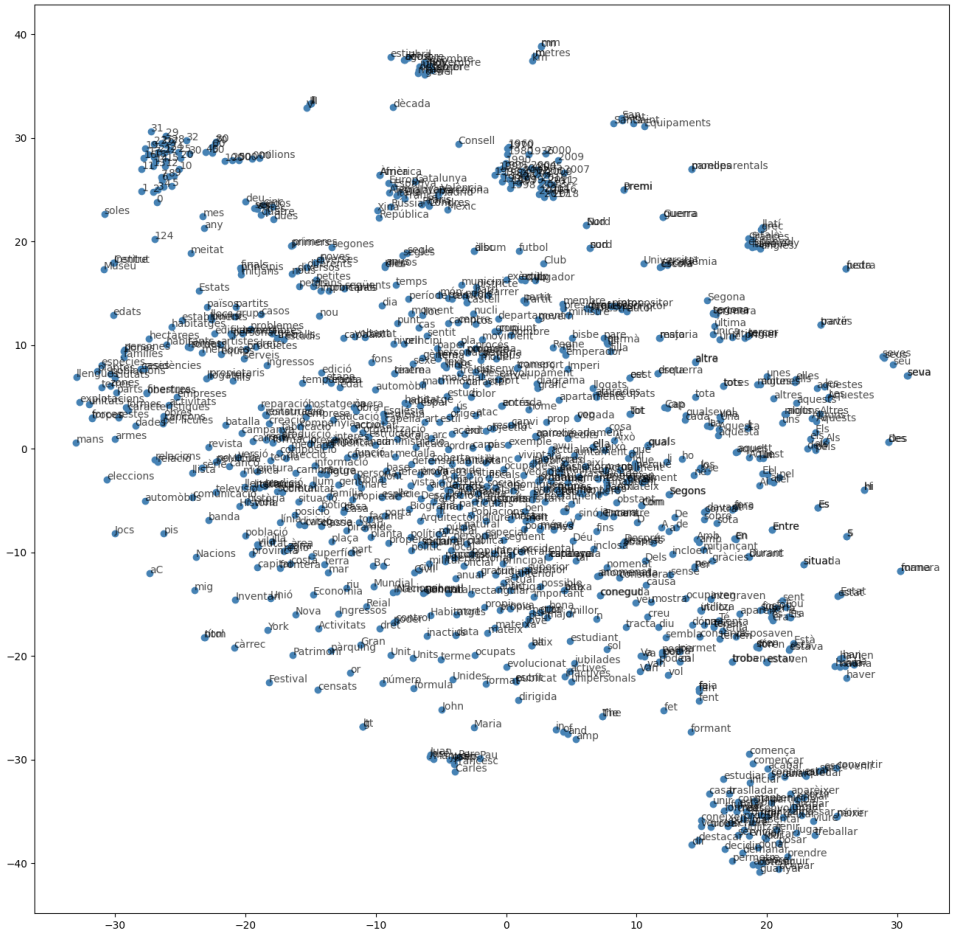

Figure 5: *t-SNE* of the embeddings computed with the CBOW method implemented previously (1000 words)