Assignment 1. Catalan Word Vectors. Training and analysis

Alex Carrillo and Robert Tura
Spoken and Written Language Processing

March 9, 2020
Universitat Politècnica de Catalunya

1 Introduction

The *word2vect* technique is a predictive deep learning based model to compute and generate high quality and continuous dense vector representations of words, which capture contextual and semantic similarity. Essentially, these models take in a textual corpus, create a vocabulary of words and generate dense word embeddings for each word in the vector space representing that vocabulary.

There are two different model architectures which can be leveraged by word2vect to create these word embedding representations. These include, the *Continuous Bag of Words (CBOW)* model and the *Skip-gram* model. In this report we focus on the CBOW model, which learns word representations by predicting a word according to its context defined as a symmetric window containing all the surrounding words.

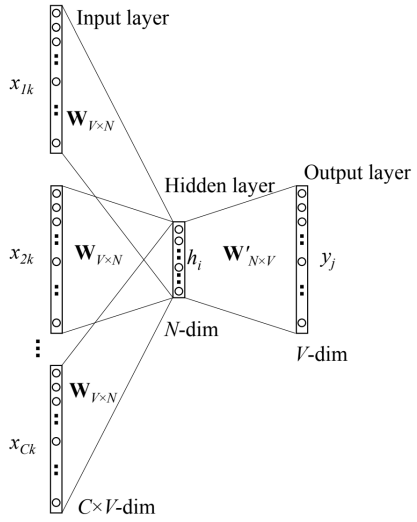

Figure 1: Continuous Bag of Words (CBOW) model.

2 Improving the CBOW model

The standard CBOW model sums all the context word vectors with the same weight, but when you are analyzing the sentence: "La Lara _____ francès bé" we would say that the first and the last words are meaningless to know the middle word is "parla", but if the word was like this: "La Lara _____ poc francès" we see that now the last word has turn to be more important than it was in the other sentence.

As we said earlier, the key to CBOW is to get enough context and not too much, but also, to accentuate certain parts. The basic point of windowing is that it captures the language structure from the statistical point of view. The bigger the window is, the more context you have to work with; optimum length really depends on the application. If the window is too short, you may fail to capture important differences. On the other hand, if it is too long, you may fail to capture the "general knowledge" and only stick to particular cases. Our first approach will be guided by a simple human insight to give less weight the farther the words are.

2.1 Weighted sum of context words

2.1.1 Fixed scalar weight

This first approach is easy and non variant throughout the process of training, we give 1/6 of weight to the first and the last words and 1/3 to the second and third. As this is human based, we have this hypothesis that closer words have more significance in general. The code that allows us to weight the inputs is the next:

```
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)
        # a) A fixed scalar weight (1,2,2,1) to give more weight to the words
        # that are closer to the predicted central word.
        self.register_buffer('position_weight', torch.tensor([1,2,2,1],
                                    dtype=torch.float32).view(1,4,1))

    def forward(self, inputs):
        U = (self.emb(inputs)*self.position_weight).sum(dim=1)
        V = self.lin(U)
        return V
```

As we thought this new approach improves the accuracy of our model, but not as much as we expected but this seems to be a nice place to grow from.

2.1.2 Trained scalar weight

Next step is to benefit from the fact that we are using neural networks and that the main action in this is to backpropagate the values of its parameters and improve them each time one cycle is done. So we'll use this perk in our favour including this new weights into the cycle of training in the backpropagation. The code to do so is:

```
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) A trained scalar weight for each position.
        self.position_weight = nn.Parameter(torch.tensor([1,2,2,1],
                                    dtype=torch.float32).view(1,4,1))

    def forward(self, inputs):
        U = (self.emb(inputs)*self.position_weight).sum(dim=1)
        V = self.lin(U)
        return V
```

As before the accuracy has improved, this time even more, but it's not still enough to be happy about it for there is still place for us to tweak the model in order to predict with better accuracy, this idea focuses not on training a single value, but to use a vector of values what then will be element-wise multiplied by the corresponding position to be used for the specific case.

2.1.3 Trained vector weight

This is the final approach we tried for the CBOW algorithm, and the one that had the best results. The idea is based of the first and the second but taking it to the next level, as before we are starting some values (in this specific case 1,1,1,1) and then we are training them as we backpropagate the whole network, and we don't just use one vector for the whole embedded chunk, we are creating a separate vector for each.

```
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)
        # c) A trained vector weight for each position. Each word vector is
        # element-wise multiplied by the corresponding position-dependen weight,
        # and then added with the rest of the weighted word vectors.
        self.position_weight = nn.Parameter(torch.ones([4,100],
                                            dtype=torch.float32))

    def forward(self, inputs):
        U = (self.emb(inputs)*self.position_weight).sum(dim=1)
        V = self.lin(U)
        return V
```

This allows us to go even further than before in terms of accuracy, but also lets us have more parameters and tricks up the sleeve to tweak the algorithm.

2.2 Hyperparameter optimization

With our last approach, we then started to play with the fixed parameters of the network. The first variable we tried to change was the number of epochs, which resulted in a gain of accuracy but giving up a lot of computation time (and Kaggle has some time limitations).

The second parameter we used to improve the network was the batch size, which resulted in a very subtle improvement but a nice parameter to lower the execution time taking into account that the accuracy loss was minimal.

And finally we ended up changing the embedding dimension, what resulted in a clear game changer. As the embedding dimension increases, the amount of word combinations that are imputed to the network at once is increased also what means more computation time but also more accuracy for the fact that there is "more training" at each step.

2.3 Other methods to obtain word embeddings

To implement a skip-gram algorithm we found the library `gensim`, that allows you to create word embeddings with CBOW as it is done here with PyTorch's functions, but also its able to do skip-gram. The way to call the model would be:

```
import gensim.models as gs
model = gs.Word2Vec(data, size = embedding_dim, window = 5 , sg = 1, min_count = 5)
```

(`window` parameter stands for how far should the embedding go 2,1,0,-1,-2, `sg` is a flag to call the skip gram algorithm, and `min count` is basically a threshold for non frequent words)

This library seems good and lets you train with more data and improve the word embedding. But there is an issue we encountered: when trying the code with our Notebook in Kaggle we were unable to perform this word embedding due to class type errors. We decided to go forward and work on the rest of the assignment, for we knew what we were supposed to perform but we had issues with the code itself.

3 Performance of the improved CBOW model

In order to measure the quality of the resulting vector representations we complete an intrinsic evaluation to test, perform and find good and bad examples of closest words and analogies. This way of analyzing the behavior of the CBOW word vectors in terms of words with multiple meanings, synonyms and antonyms, word frequency, different types of analogies or bias (gender, race, sexual orientation, etc.) should be descriptive enough to reflect the quality of our trained model.

Broadly, we expect that no only similar words will tend to be close to each other, but that words may have multiple degrees of similarity. This could mean that, for example, although nouns can have different endings, if we search for similar words in a subspace of the vector space, it is possible to find words that have similar endings.

3.1 The vector space

Somewhat surprisingly, when these type of word vector architectures were created, it was found that similarity of word representations goes beyond simple syntactic regularities. The word2vec objective function causes words that occur in similar contexts to have similar embeddings. However, this explanation is very "handwavy" and, for what we have searched there is no clear answer. In our case, as a result of the outcomes obtained we have concluded that we might consider a word embedding to be "intuitive" when it has the following properties:

(i) **Semantically similar** words (like *car, truck, sedan, vehicle*) are close to each other

(ii) **Embedding arithmetic** can be done: $vect(\text{"king"}) - vect(\text{"man"}) + vect(\text{"woman"}) \approx vect(\text{"queen"})$

Moreover, for what we have intuited, the performance of word2vec, and therefore the fact of similar embeddings distance, is not a result of the models *per se*, but of the choice of specific hyperparameters and word2vec architecture, so each approach yields particular performances.

3.2 The *WordVector* class

In the following section we describe the implementation details of the main methods of the *WordVector* class skeleton we built.

- The `.most_similar()` method makes use of the **cosine similarity** (1) method (normalized dot product) to retrieve the n closest vectors of the subspace, hence, the n most similar words in our vocabulary.

$$cos(w_x, w_y) = \frac{w_x \cdot w_y}{\|w_x\|\|w_y\|} \tag{1}$$

- The `.analogy()` method performs a simple **algebraic operation** with the vector representation of words. It only computes a subtraction and an addition (2) between three terms to find a word that is similar to the third given (v_3) in the same sense as the first (v_1) is similar to the second one (v_2). Then, it searches for the words closest to y_{sim} by making use of the previous `.most_similar()` method.

$$y_{\text{sim}} = v_2 - v_1 + v_3 \tag{2}$$

- The `.closest_words_scatterplot()` lays out a scatter plot in the 2-dimensional plane of the results generated from the previous methods. This is accomplished by performing a ***Principal Component Analysis*** **(PCA)** to the vector datapoints. In this manner, the word vector embedding size (= 100 in this report) is reduced to a dimensionality of 2 (the two principal components).
- The `.clusters()` method performs a **TSNE** dimension reduction and, with the new obtained points, it uses the ***KMeans*** algorithm in order to find some clusters among data. This way, it is displayed a plot of the output clusters by color.

The full code details of the *WordVector* class can be found in the Appendix (A).

3.3 Finding similar and analogous words

Once we have built our *WordVector* class, we can test the implemented functions in order to see how accurate they are. The following Table 1 has been generated using the `.most_similar()` method:

	Most similar words by decreasing similarity
Nouns	
tardor	primavera, estiu, darreria, nit, vigília, carronya
missatge	text, senyal, comentari, discurs, diàleg, telegrama
pastor	predicador, capellà, noi, gos, missioner, clergue
entrebanc	obstacle, impuls, afer, inconvenient, esdeveniment, problema
Verbs	
vindria	ve, procedeix, venia, vingué, derivaria, procedia
aconseguir	assolir, obtenir, obtindre, assegurar, aprofitar, atènyer
proposaren	proposen, proposava, proposà, preveia, presentaren, demanaren
admet	admeten, especifica, accepta, implementa, implica, utilitza
Adjectives	
optimista	pessimista, ambigua, realista, cruel, subjective, punyent
gran	enorme, considerable, notable, major, petita, immensa
majestuosa	magnífica, monumental, imponent, grandiosa, mirador, grotescos
suau	agradable, abrupte, lleu, fort, lleuger, àgil

Table 1: Examples of most similar words of three types of word categories (nouns, verbs and adjectives). The highlighted word corresponds to the most similar word in terms of cosine similarity, followed by the other five most similar ones ordered by decreasing similarity.

As we can observe, the overall outputs of the model are good in broad strokes. Though, it is clear that depending on the word category we get different outcomes. One might think that the most similar words of a given one are basically synonyms. We have seen that this is a particular behavior of nouns, as we get many synonyms in the retrieved words. However, for the word `tardor` we only obtain related words but no synonym.

Now, for verbs we do not get a consistent answer: for some of them we obtain tenses from the same verb and for some others we get nearly the same tenses but from different verbs. Moreover, seems that irregular verbs have a hard time finding similar words. Finally, for the last category of words, adjectives, we get a mixture of: synonymous words for some like `gran` or `enorme`, and antonyms for some like `optimista` or `suau`, for instance.

Following, we make use of the `.analogy()` method to check some syntactic relations in the Table 2 which can be used as a kind of benchmark to test the accuracy of the model:

Syntactic relationships

Type	First pair	Second pair	Word	Guess
Pronouns	nosaltres	nostre	vosaltres	vostre
	aquest	aquell	aquesta	aquella
	mi	meu	tu	teu
Participles	sentir	sentit	llegir	pla
	cuinar	cuinat	fer	fa
	cantar	cantat	ballar	doblat
Genres	metge	metgessa	alcalde	alcaldessa
	jutge	jutgessa	secretari	secretària
	mestre	mestra	ministre	ministra

Table 2: Examples of syntactic relationships between two pairs of words of three types of relationships (pronouns, participles and genres). The highlighted word corresponds to the analogous word (in the same way the first pair word is analogous to the second pair word) guessed by the model.

It should be stated that the words in the table above have been selected with the aim of showing analogies from a syntactic level. This refers to the set of rules that govern the structure of a sentence in a given language, including order, stems and root words.

From the results, we conclude that the model is able to correctly find word analogies between different pronouns in an accurate way. In the same sense, it does not have problems at all when guessing gender inflexions, probably as most terms only change the suffix "-a". However, the model is a mess when finding word analogies between verb participles. We think this is due to the morphology of verbs, which some become irregular in some tenses.

Finally we examine some semantic relationships with the same `.analogy()` method used in the previous test. Now, we focus on the real significance of the word pairs in the pure sense of meaning. The results are somewhat surprising as these type of analogies do not follow grounded rules of a language structure, but are derived from human comprehension.

Semantic relationships

Type	First pair	Second pair	Word	Guess
Opposites	ample	estret	llarg	costat
	petit	gran	alt	major
	ric	pobre	abundant	escassa
Cities	Espanya	Madrid	Catalunya	Barcelona
	Alemanya	Berlín	Itàlia	Nàpols
	Amèrica	Washington	Canadà	Quebec
Season/time	gener	hivern	agost	estiu
	mati	tarda	vespre	nit
	hivern	fred	estiu	sec

Table 3: Examples of semantic relationships between two pairs of words of three types of relations (opposites, cities and season/time). The highlighted word corresponds to the analogous word (in the same way the first pair word is analogous to the second pair word) guessed.

From the table above we observe that a good fit of semantic analogies is obtained when some context relationship is provided. For example, when testing opposites words we do not get good results as the input words we are feeding only have an opposite meaning relationship but they do not share a context themselves. Moving on to a more connected category of words, cities, we get a good fit between analogies, as they keep a straightforward semantic relationship (although the best guesses should be the real capitals). In the same way, we obtain good results for season and time words, as they share a clear context between them.

3.4 Prediction accuracy

With all the changes to our code we got a 43.2 percent of accuracy in the training set with a loss of 3.16. In the test set we got a 31.96 percent of accuracy which lead us to be fifths in the leaderboard!

3.5 Visualizing word similarities, analogies and clustering properties

3.5.1 Similar words

In the plots below we show some of the results stated in the previous sections. Firstly, regarding the `.most_similar()` method, we get the following:

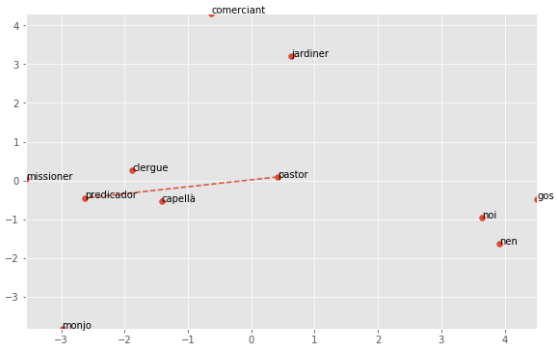

Figure 2: Word 'pastor' (noun).

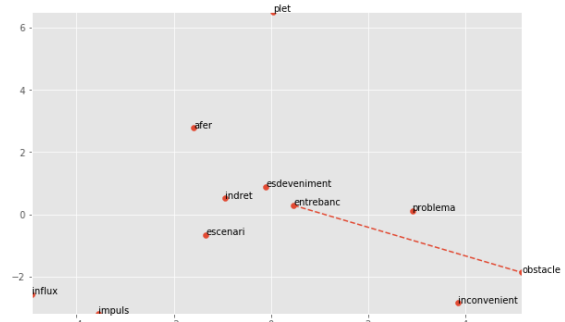

Figure 3: Word 'entrebanc' (noun).

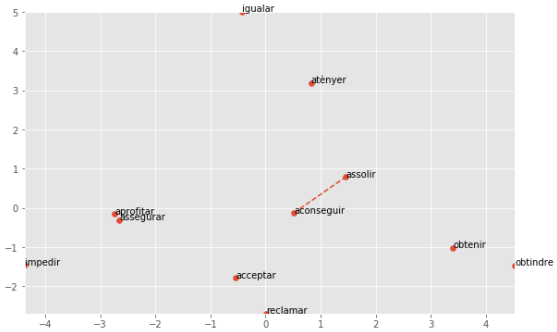

Figure 4: Word 'aconseguir' (verb)

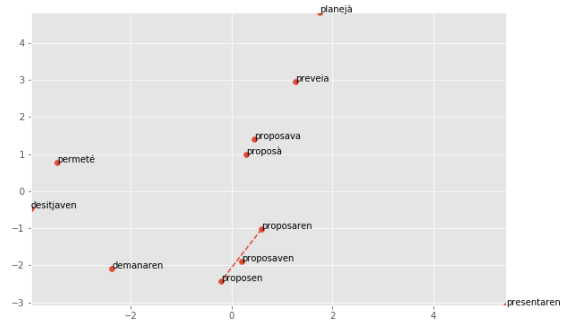

Figure 5: Word 'proposaren' (verb)

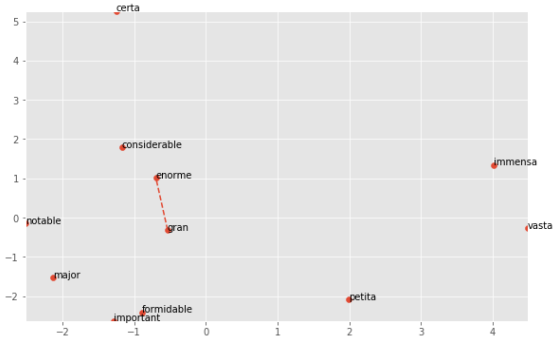

Figure 6: Word 'gran' (adjective)

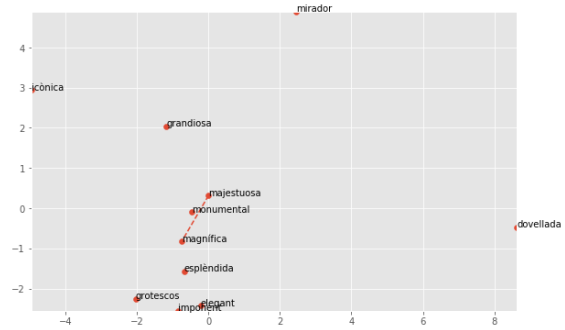

Figure 7: 'majestuosa' (adjective).

As it can be seen, in general, similar words are keep in the same nearby area, mostly synonyms and closely related words regarding meaning. In contrast, those words who share opposites meaning, *i.e.* antonyms, are mainly farther away in space than the given word.

3.5.2 Syntactic analogies

Next, we display some plots of word analogies by making use of the `.analogy()` method, in each of the three chosen syntactic relationships; pronouns, participles and genres:

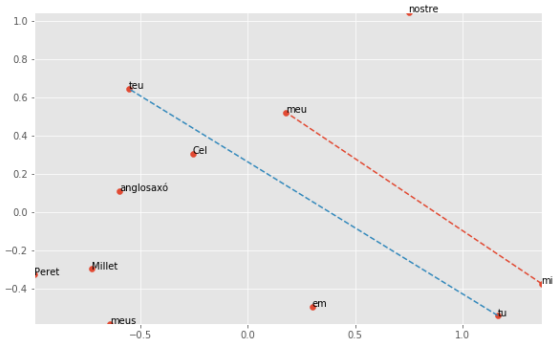

Figure 8: mi: meu (pronouns)

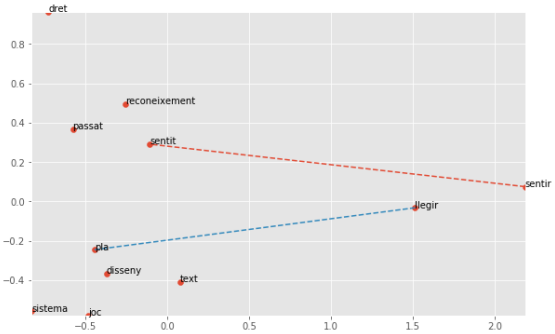

Figure 9: sentir: sentit (participles)

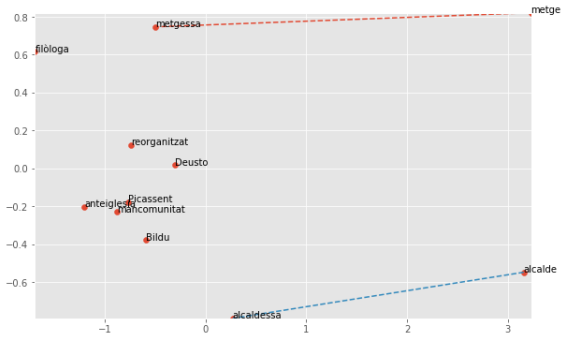

Figure 10: metge: metgessa (genres)

As we said in the previous sections, we observe a linearity between those terms which produce an accurate analogy (like for pronouns and genres), whereas on those words for which the associated analogy is not that good, the parallelism is lost and the word pairs become closest to orthogonality (as it can be seen in the participles).

3.5.3 Semantic relationships

Lastly, we show some other plots of word analogies but now focusing on the real significance of word pairs, that is, semantic relationships:

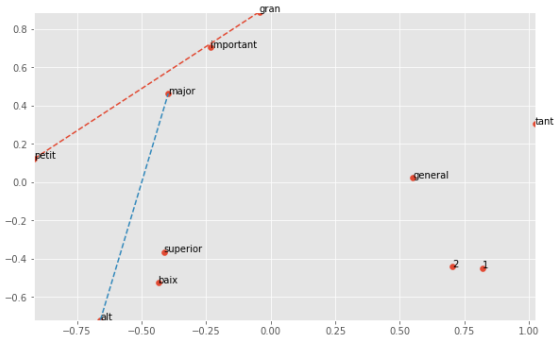

Figure 11: petit: gran (opposites)

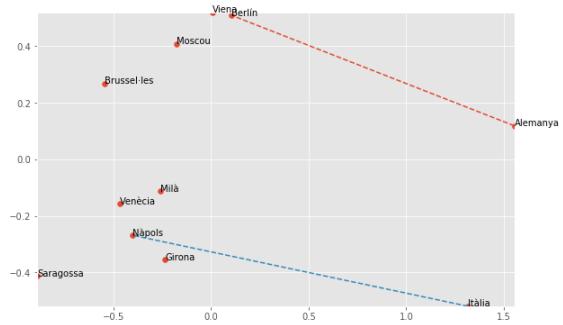

Figure 12: DE: Berlín (cities)

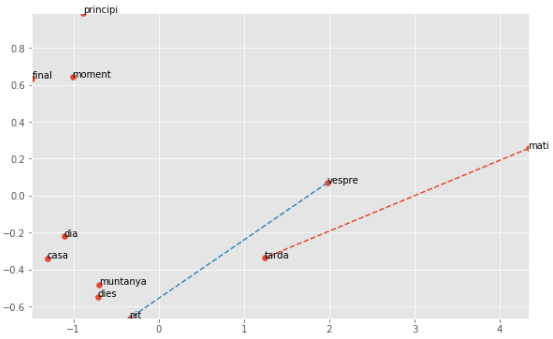

Figure 13: mati: tarda (season)

Again, we perceive the quality of the obtained analogies at a glance. For instance, we said that for opposites words the guess is not good at all, as no semantic context is given for these words. Hence, the linearity between terms is lost at the plot. In contrast, both cities and season categories have a good behavior overall, the words appear in a parallel layout, which make sense.

3.5.4 Clustering

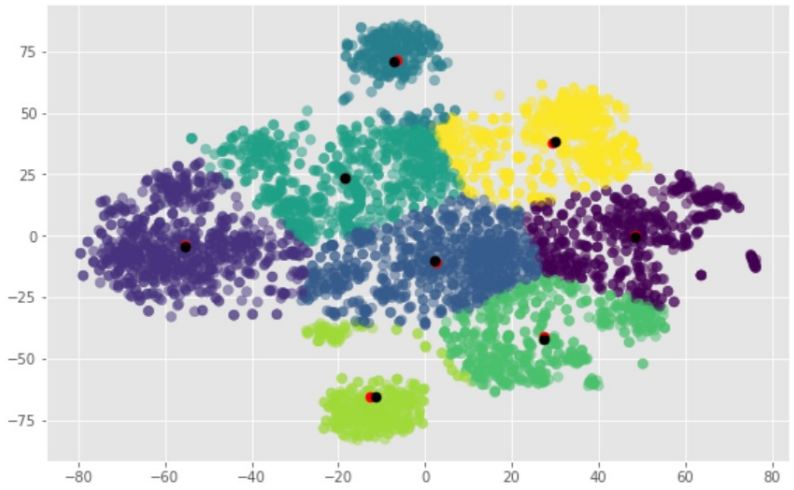

Figure 14: 2D-Visualization (T-SNE)

And here finally, we took the 5000 most relevant values and reduced its dimension up to a 2D representation in order to see different clusters that might represent different types of words.

Our first checks are the top and bottom clusters, they seem to be separated enough to have a distinction from the other ones. As we predicted they have a clear distinction, the top cluster is the 2D representation of Adverbs, and the bottom cluster represents many derivations of the verb "haver". Other clusters are different numbers, date related words, and other different little correlations between this clusters.

We used several cluster layouts and this was close to what the real words meant, the center clusters try to differentiate and they do a close job, but the ones that are further from the center tend to be more precise (except for the little clusters like the right-most one and other like that).

4 Future research

As a conclusion, we want to comment on a point that has not been emphasized throughout the report. This has to do with the two vector representations that can be used as a word vectors: the input and output vectors of the hidden layer. In our case, we have been using a first model created from the input vectors to test the most similar words of a given one, and a second model created from the output vectors to test the analogies between word pairs. This has been done arbitrarily without using any clear criteria. For future research related to this area, it should be interesting to study first: what grounded differences exist between the two word vectors, and second: how the aggregation of both vector representations (by adding or concatenating them) results on different outcomes regarding the quality of both similarity and analogy between words.

Appendix

A The *WordVector* class code

```
class WordVectors:
    def __init__(self, vectors, vocabulary):
        self.vectors = vectors
        self.vocabulary = vocabulary

    def get_vector(self, word):
        return self.vectors[self.vocabulary.get_index(word)]

    def cosine_distance(self, vect_a, vect_b):
        return np.dot(vect_a, vect_b) / (norm(vect_a)*norm(vect_b))

    def most_similar(self, word = '', word_vect = None, invalid_words = None, topn = 10):
        invalid_words = (word) if invalid_words is None else invalid_words
        word_vect = self.get_vector(word) if word_vect is None else word_vect
        cosine_dict = {}
        for item in self.vocabulary.token2idx:
            if item not in invalid_words:
                item_vect = self.get_vector(item)
                cosine_dict[item] = self.cosine_distance(word_vect, item_vect)
        cosine_dict = sorted(cosine_dict.items(), key = lambda x: -x[1])
        words_list = [(item[0], item[1]) for item in cosine_dict]
        return words_list[:topn]

    def analogy(self, x1, x2, y1, topn = 8, keep_all = False):
        x1_vect = self.get_vector(x1)
```

```
    x2_vect = self.get_vector(x2)
    y1_vect = self.get_vector(y1)
    dangling_word_vect = (x2_vect - x1_vect) + y1_vect
    remove_words = None if keep_all else (x1, x2, y1)
    return self.most_similar(word_vect = dangling_word_vect, invalid_words = remove_words, topn = t

def closest_words_scatterplot(self, word, topn = 10, analogy = False):
    X = np.empty((0,100))
    if analogy:
        word_labels = word
        similar_words = self.analogy(word[0],word[1],word[2])
        X = np.append(X, [np.array(self.get_vector(word[0]))], axis=0)
        X = np.append(X, [np.array(self.get_vector(word[1]))], axis=0)
        X = np.append(X, [np.array(self.get_vector(word[2]))], axis=0)
    else:
        word_labels = word
        similar_words = self.most_similar(word[0])
        X = np.append(X, [np.array(self.get_vector(word[0]))], axis=0)

    for item in similar_words:
        word_vect = self.get_vector(item[0])
        word_labels.append(item[0])
        X = np.append(X, [np.array(word_vect)], axis=0)

    X_embedded = PCA(n_components = 2).fit_transform(X)
    x_coords = X_embedded[:, 0]
    y_coords = X_embedded[:, 1]
    plt.scatter(x_coords, y_coords)

    for label, x, y in zip(word_labels, x_coords, y_coords):
        plt.annotate(label, xy=(x, y), xytext=(0, 0), textcoords='offset points')
    plt.xlim(x_coords.min()+0.00005, x_coords.max()+0.00005)
    plt.ylim(y_coords.min()+0.00005, y_coords.max()+0.00005)
    if analogy:
        plt.plot(x_coords[0:2], y_coords[0:2], '--')
        plt.plot(x_coords[2:4], y_coords[2:4], '--')
    else:
        plt.plot(x_coords[0:2], y_coords[0:2], '--')
    plt.show()

def clusters(self, C = 3):
    word_vectors = {}
    for word in words:
        # 'words' is a previous created tuple with the
        # 5000 most frequent words sorted decreasingly
        word_vectors[word] = self.get_vector(word)
    X = np.array(list(word_vectors.values()))

    X_embedded = TSNE(n_components = 2).fit_transform(X)
    x_coords = X_embedded[:, 0]
    y_coords = X_embedded[:, 1]

    kmeans = KMeans(n_clusters = 8).fit(X_embedded)
    centroids = kmeans.cluster_centers_
```

```
plt.scatter(x_coords, y_coords, c = kmeans.labels_.astype(float), s = 50, alpha = 0.5)
plt.scatter(centroids[:, 0], centroids[:, 1], c = 'red', s = 50)
closest, _ = pairwise_distances_argmin_min(centroids, X_embedded)

for index in closest:
    plt.scatter(x_coords[index], y_coords[index], c = 'black', s = 50)
    point = words[index]
    print(point)
```

References

[1] Google Code Archive. *word2vec*. 2013. URL: https://code.google.com/archive/p/word2vec/.

[2] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL].

[3] Xin Rong. *word2vec Parameter Learning Explained*. 2014. arXiv: 1411.2738 [cs.CL].

[4] Tomas Mikolov et al. *Advances in Pre-Training Distributed Word Representations*. 2017. arXiv: 1712.09405 [cs.CL].

[5] Towards Data Science. *A Beginner's Guide to Word Embedding with Gensim Word2Vec Model*. 2019. URL: https://towardsdatascience.com/a-beginners-guide-to-word-embedding-with-gensim-word2vec-model-5970fa56cc92.

[6] Gonzalo Ruiz de Villa. *Introducción a Word2vec (skip gram model)*. 2019. URL: https://medium.com/@gruizdevilla/introducciC3B3n-a-word2vec-skip-gram-model-4800f72c871f.

[7] GitHub user: zhlli1. *Genism-word2vec*. 2019. URL: https://github.com/zhlli1/Genism-word2vec.