LAB 1

WORD VECTORS: TRAINING AND ANALYSIS

POE

DATA SCIENCE AND ENGINEERING
UNIVERSITAT POLITÈCNICA DE CATALUNYA

Adrián Cerezuela Hernández - 48222010A

Ramon Ventura Navarro - 21785256R

March 2024

1 CBOW Model Improvement

Word vectors play a vital role in NLP and find widespread application in tasks such as categorizing text and information retrieval. How good these word vectors are affects the effectiveness of these tasks.

In this first part, we will work with the CBOW model, which is really good at making high-quality word vectors. Our job is to make, study, and check word vectors using this CBOW model, using a set of data from Wikipedia.

During training, the model will take 4 batches of 1000 training examples. Each example has a main word and the words around it (within seven words, three on each side). The model uses these context words to predict the main word, and then compare it to the actual embedding of the target word using a cross-entropy function. By the conclusion of the training, the model will have acquired a collection of word embeddings suitable for representing words in a condensed vector space.

We are given a standard CBOW model which sums all the context word vectors with the same weight. It will be our baseline model. The metrics obtained from its training, and therefore our metric baseline, are the following:

	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	22.5%	5.47	25.3%	5.04	25.9%	4.94	26.2%	4.89
Validation (Wikipedia)	22.7%	5.33	23.4%	5.21	23.8%	5.15	24.0%	5.12
Validation (El Periódico)	14.8%	6.24	15.3%	6.16	15.4%	6.13	15.5%	6.10

1.1 Fixed Scalar Weight

In this first section we are asked to implement and evaluate a weighted sum of the context words with a fixed scalar weight, which is $[1, 2, 3, 3, 2, 1]$, to give more weight to the words that are closer to the predicted central word. We are also given the python command to do so, which will be added to the $\_\_init\_\_()$ method with $self.register\_buffer()$ 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)
        self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],dtype=torch.f
loat32))

    # 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), multiplied by the defined weights
        u = torch.matmul(self.position_weight,e)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

With that configuration, we train the model for the different datasets described in the assignment and check its accuracy and loss at the end of each epoch for each of them. The results are shown in the following table:

	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	29.1%	5.00	32.5%	4.52	33.2%	4.41	33.6%	4.34
Validation (Wikipedia)	29.9%	4.81	30.8%	4.66	31.1%	4.60	31.4%	4.56
Validation (El Periódico)	20.1%	5.79	20.8%	5.68	21.2%	5.63	20.8%	5.61

In the very first training epoch, it can be observed that the model offers better metrics than the standard CBOW proposed as a baseline. However, with a 33.6% accuracy and 20-30% validation, we cannot consider it a good model. Additionally, we can see that the model's performance varies depending on the validation set, achieving better results for the *Wikipedia* dataset, which means it fits these data better than those from *El Periódico*. In the following section we will try to implement a more optimal solution.

1.2 Trained Scalar Weight

Once tested its performance using fixed scalar weights, in this second section we are asked to repeat the process with a trained scalar weight for each position instead. We will use $[1, 2, 3, 3, 2, 1]$ as the initial weights that we will train. As we have previously done, the CBOW class will be modified in order to achieve it as it can be seen 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.lin = nn.Linear(embedding_dim, num_embeddings, bias=False)
        # self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],dtype=torch.f
loat32))
        self.position_weight = nn.Parameter(torch.tensor([1,2,3,3,2,1], dtype=torch.floa
t32))

    # 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), multiplied by the defined weights
        u = torch.matmul(self.position_weight,e)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

As we have done before, the model is trained for different datasets. The following table is used in order to display the results:

	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	31.0%	4.75	34.4%	4.30	35.0%	4.21	35.4%	4.16
Validation (Wikipedia)	31.6%	4.58	32.7%	4.46	33.1%	4.40	33.3%	4.37
Validation (El Periódico)	22.1%	5.50	22.6%	5.41	22.9%	5.38	22.8%	5.37

The first thing we should note is that the model's performance has slightly improved compared to the previous one, but it still remains poor. The behavior across epochs also remains consistent, performing better for the *Wikipedia* validation set than for the *El Periódico* set. The next step in seeking an optimal solution to the problem would be to train vector weights.

1.3 Trained Vector Weight

Finally, taking it one step further, in this section we are asked to implement and evaluate a weighted sum of the context words with a trained vector weight for each position. Each word vector is element-wise multiplied by the corresponding position-dependent weight and then added with the rest of the weighted word vectors.

```
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)
        # self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],dtype=torch.float32))
        # self.position_weight = nn.Parameter(torch.tensor([1,2,3,3,2,1], dtype=torch.float32))
        # Tensor with all the elements following a normal dist. with 0 mean and sd=1.5
        self.position_weight = nn.Parameter(torch.randn(6,embedding_dim)*1.5+0)

        # Tensor full of ones
        # self.position_weight = nn.Parameter(torch.ones(6,embedding_dim))


    # 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), multiplied by the defined weights
        w = self.position_weight.unsqueeze(0)
        e = e * w
        u = e.sum(dim=1)
        # u shape is (B, E)
        z = self.lin(u)
        # z shape is (B, V)
        return z
```

Through this implementation in the CBOW class, our goal can be achieved. It is worth noting that we have considered two initializations for testing purposes, one in which the tensor consists of all ones, and another in which all elements of the tensor follow a normal distribution with a mean of 0 and a standard deviation of 1.5. These values have been arbitrarily chosen. It is also worth mentioning that the way to obtain u is different from previous implementations. Since the weight tensor must have 3 dimensions so we can multiply it by the tensor e with shape (B, W, E), we have found it more convenient to explicitly perform the operation by unsqueezing it to have shape (1, 6, 1). In the following tables, the performance of the model can be compared with both initializations. The first one corresponds to the full-one tensor while the second one to the tensor initialized with torch.randn:

	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	37.3%	4.16	42.0%	3.63	42.9%	3.53	43.3%	3.48
Validation (Wikipedia)	39.4%	3.89	40.7%	3.75	41.1%	3.69	41.2%	3.67
Validation (El Periódico)	29.3%	4.78	30.3%	4.65	30.8%	4.61	31.1%	4.58

	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	37.6%	4.16	42.2%	3.63	43.0%	3.53	43.4%	3.48
Validation (Wikipedia)	39.6%	3.89	40.8%	3.74	41.1%	3.69	41.3%	3.67
Validation (El Periódico)	29.4%	4.77	30.6%	4.65	31.1%	4.61	31.4%	4.58

We can see how the metrics are practically identical. Due to the minimal difference, we will stick with this last configuration as the best version of the model, since in either case, the performance improves significantly compared to any of the previous sections.

1.4 Hyperparameter Optimization

With our word vectors constructed, it would be interesting to study the performance of the model as a function of one of its parameters. In this part of the study the following parameters will be included: embedding size, optimizer and number of epochs. The remaining parameters for which this process could be carried out have not been considered due to the rapid depletion of GPU time during testing, so the conclusions will not be entirely representative for the model, but for this set of parameters.

1.4.1 Embeddinng size

In order to find the optimal embedding size for our model, we have performed hyperparameter tuning, considering the following values for the parameter: *[50,100,150,200]*. The code fragments with modifications are the ones shown below. Following them, you will find a table collecting the logs for each value of the hyperparameter.

```
params = SimpleNamespace(
    embedding_dim = [50,100,150,200],
    batch_size = 1000,
    epochs = 4,
    preprocessed = f'{DATASET_ROOT}/{DATASET_PREFIX}',
    working = f'{WORKING_ROOT}/{DATASET_PREFIX}',
    modelname = f'{WORKING_ROOT}/{DATASET_VERSION}.pt',
    train = True
)
```

```
models_emb = []
for emb in params.embedding_dim:
    model = CBOW(len(vocab), emb)
    print(model)
    models_emb.append(model.to(device))
    for name, param in model.named_parameters():
        print(f'{name:20} {param.numel()} {list(param.shape)}')
    print(f'TOTAL                {sum(p.numel() for p in model.parameters())}')
```

```
for model in models_emb:
    optimizer = torch.optim.Adam(model.parameters())

    train_accuracy = []
    wiki_accuracy = []
    valid_accuracy = []
    for epoch in range(params.epochs):
        acc, loss = train(model, criterion, optimizer, data[0][0], data[0][1], params.batch_size,
device, log=True)
        train_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | train accuracy={acc:.1f}%, train loss={loss:.2f}')
        acc, loss = validate(model, criterion, data[1][0], data[1][1], params.batch_size, device)
        wiki_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f}
(wikipedia)')
        acc, loss = validate(model, criterion, valid_x, valid_y, params.batch_size, device)
        valid_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Perió
dico)')

    # Save model
    torch.save(model.state_dict(), params.modelname)
```

embedding_dim = 50	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	35.0%	4.42	39.6%	3.90	40.3%	3.81	40.7%	3.77
Validation (Wikipedia)	36.7%	4.16	37.9%	4.02	38.5%	3.96	38.5%	3.94
Validation (El Periódico)	27.9%	4.99	28.6%	4.89	28.9%	4.84	29.1%	4.82

embedding_dim = 100	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	37.6%	4.16	42.2%	3.63	43.0%	3.53	43.4%	3.48
Validation (Wikipedia)	39.6%	3.89	40.8%	3.74	41.1%	3.69	41.3%	3.67
Validation (El Periódico)	29.4%	4.77	30.6%	4.65	31.1%	4.61	31.4%	4.58

embedding_dim = 150	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	38.7%	4.04	43.3%	3.50	44.3%	3.39	44.7%	3.34
Validation (Wikipedia)	40.8%	3.76	41.8%	3.62	42.3%	3.57	42.4%	3.54
Validation (El Periódico)	30.3%	4.68	31.2%	4.54	31.5%	4.50	31.7%	4.48

embedding_dim = 200	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	39.5%	3.96	44.0%	3.42	45.0%	3.30	45.5%	3.25
Validation (Wikipedia)	41.4%	3.69	42.5%	3.56	42.8%	3.51	43.0%	3.48
Validation (El Periódico)	30.8%	4.62	32.0%	4.50	32.2%	4.46	32.5%	4.43

In the results, it can be observed that the model's accuracy improves as the embedding dimension increases. However, it can be seen that the improvement is more significant for the training set than for either of the two validation sets. One can expect that increasing the embedding dimensions will widen the gap between the training and validation accuracies for either of the two datasets.

1.4.2 Number of epochs

With the aim of looking for the best number of epochs for our model, the same performance is repeated. In this case, we will be trying the following values for the hyperparameter: *[2,4,6,8]*.

```
params = SimpleNamespace(
    embedding_dim = 100,
    batch_size = 1000,
    epochs = [2,4,6,8],
    preprocessed = f'{DATASET_ROOT}/{DATASET_PREFIX}',
    working = f'{WORKING_ROOT}/{DATASET_PREFIX}',
    modelname = f'{WORKING_ROOT}/{DATASET_VERSION}.pt',
    train = True
)

optimizer = torch.optim.Adam(model.parameters())

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
for epoch_num in params.epochs:
    print("Test with number of epochs: ", epoch_num)
    for epoch in range(epoch_num):
        acc, loss = train(model, criterion, optimizer, data[0][0], data[0][1], params.batch_size,
device, log=True)
        train_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | train accuracy={acc:.1f}%, train loss={loss:.2f}')
        acc, loss = validate(model, criterion, data[1][0], data[1][1], params.batch_size, device)
        wiki_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f}
(wikipedia)')
        acc, loss = validate(model, criterion, valid_x, valid_y, params.batch_size, device)
        valid_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Perió
dico)')

    # Save model
    torch.save(model.state_dict(), params.modelname)
```

		Training	Validation (Wiki)	Validation (EP)			Training	Validation(Wiki)	Validation (EP)
Epoch 000	Accuracy	37.5%	39.4%	29.5%	Epoch 004	Accuracy	43.6%	41.5%	31.3%
	Loss	4.17	3.88	4.76		Loss	3.45	3.65	4.57
Epoch 001	Accuracy	42.1%	40.7%	30.4%	Epoch 005	Accuracy	43.8%	41.6%	31.5%
	Loss	3.62	3.74	4.65		Loss	3.44	3.65	4.56
Epoch 002	Accuracy	43.0%	41.1%	31.0%	Epoch 006	Accuracy	43.9%	41.5%	31.4%
	Loss	3.52	3.69	4.60		Loss	3.42	3.63	4.56
Epoch 003	Accuracy	43.4%	41.4%	31.3%	Epoch 007	Accuracy	44.0%	41.7%	31.5%
	Loss	3.48	3.67	4.57		Loss	3.42	3.63	4.55

In the results, it can be observed that, following the same pattern as with the previous parameter, the training error decreases slightly faster than the validation error. Therefore, it can be expected that increasing the number of epochs will make this even more noticeable. However, it can be seen that increasing the number of epochs provides a slight improvement in the metrics, although this is more pronounced in the early epochs of training.

1.4.3 Optimizer

Continuing with our hyperparameter tuning section, the next one to perform with will be which optimizer is used for our CBOW model. In this case, the code will be executed with an *Adam* optimizer, with a *SGD* and lastly with a *RMSprop*. Below you can find the code with the modifications for the two alternatives, as the results with the *Adam* optimizer have been previously shown.

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.001)

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
for epoch in range(params.epochs):
    acc, loss = train(model, criterion, optimizer, data[0][0], data[0][1], params.batch_size, devi
ce, log=True)
    train_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | train accuracy={acc:.1f}%, train loss={loss:.2f}')
    acc, loss = validate(model, criterion, data[1][0], data[1][1], params.batch_size, device)
    wiki_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (wikipedia)')
    acc, loss = validate(model, criterion, valid_x, valid_y, params.batch_size, device)
    valid_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Periódico
)')

# Save model
torch.save(model.state_dict(), params.modelname)
```

Optimizer = SDG	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	24.7%	5.40	29.9%	4.75	31.9%	4.53	33.2%	4.38
Validation (Wikipedia)	25.0%	5.15	28.2%	4.84	30.1%	4.67	31.3%	4.55
Validation (El Periódico)	17.9%	5.94	20.4%	5.65	21.9%	5.47	22.8%	5.38

We can observe poorer performance with this type of optimizer, even significantly worse when compared to the model optimized using *Adam* as the baseline. Therefore, we can discard the *SGD* optimizer as a good alternative for training our CBOW model. Below are the results using the *RMSprop* one.

Optimizer = RMSprop	Epoch 000		Epoch 001		Epoch 002		Epoch 003	
	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss	Accuracy	Loss
Training	38.6%	4.45	39.9%	4.31	40.0%	4.30	40.0%	4.30
Validation (Wikipedia)	37.4%	4.47	37.9%	4.45	37.6%	4.49	37.7%	4.45
Validation (El Periódico)	28.5%	5.32	29.5%	5.28	29.1%	5.33	28.9%	5.29

```
optimizer = torch.optim.RMSprop(model.parameters())

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
for epoch in range(params.epochs):
    acc, loss = train(model, criterion, optimizer, data[0][0], data[0][1], params.batch_size, devi
ce, log=True)
    train_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | train accuracy={acc:.1f}%, train loss={loss:.2f}')
    acc, loss = validate(model, criterion, data[1][0], data[1][1], params.batch_size, device)
    wiki_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (wikipedia)')
    acc, loss = validate(model, criterion, valid_x, valid_y, params.batch_size, device)
    valid_accuracy.append(acc)
    print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Periódico
)')

# Save model
torch.save(model.state_dict(), params.modelname)
```

Finally, we see that the *RMSprop* optimizer also does not surpass the metrics obtained with *Adam*, so we conclude that the latter is the one that best operates our model and, therefore, should be chosen.

In conclusion, after this hyperparameter tuning process, all results suggest that the training error decreases faster than the validation error when increasing the value of any hyperparameter used, but they improve considerably when adjusted. Of all the metrics obtained, the configuration that provides the highest accuracy is with the model as it was previously trained, but modifying the embedding dimension to 200, and perhaps if this were higher, we would achieve higher accuracy.

Additionally, it is concluded, after completing the process of this section, that learning the weight during training (whether it is a scalar or a vector) improves the performance of CBOW. Moreover, using a vector weight leads to better performance than using a scalar weight.

2 Word Vectors Evaluation

2.1 WordVector Class

In this second part, we will study how well the trained embeddings perform when used with real examples. In order to do so, we have used the best CBOW version developed in the previous task, which is the model trained with trained vector weights, to implement the *most_similar* and *analogy* methods.

Firstly, we have developed the *most_similar* method, which takes a word as input and returns the list of words that are most similar to it. The similarity between words is calculated based on the cosine similarity between the input word's vector representation and the vectors of other words in the embedding space. It first computes the similarity scores between the input word and all other words in the vocabulary, and then, it sorts them by their scores and returns the *topn* similar words.

The code is found below.

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

    def most_similar(self, word, topn=10):
        word_idx = self.vocabulary.get_index(word)
        word_vec = self.vectors[word_idx]

        # Compute norms for cosine similarity scores
        embedding_norms = np.linalg.norm(self.vectors, axis=1)
        word_vec_norm = np.linalg.norm(word_vec)

        # Add a small epsilon value to avoid division by zero and compute cosine similarity scores
        epsilon = 1e-10
        similarity_scores = np.dot(self.vectors, word_vec) / ((embedding_norms + epsilon) * (word_vec_norm + epsilon))

        # Exclude the word itself and the <pad> word from the list of similar words
        similarity_scores[word_idx] = similarity_scores[self.vocabulary.get_index('<pad>')] = -np.inf

        # Get indices of top similar words
        similar_indices = np.argpartition(similarity_scores, -topn)[-topn:]

        # Get similar words and their cosine similarities
        similar_words = [(self.vocabulary.get_token(idx), similarity_scores[idx]) for idx in similar_indices]

        # Sort by similarity score in descending order
        similar_words.sort(key=lambda x: x[1], reverse=True)

        return similar_words
```

Secondly, in order to implement the *analogy* method, we will need the help of an auxiliary function, *find_most_similar_words*. This last function helps us compute the most similar words for more than a single input word.

Finally, let's understand how *analogy* is constructed. This method enables the exploration of analogical relationships between words. Given a set of 3 input words, it constructs an analogy vector to represent the relationship between x_1, x_2, and y_1. The estimation of the analogous word of y_1 based on the relation between x_1 and x_2 is computed with the following operation:

$$analogy = y_1 + x_2 - x_1$$

The code can be found below:

```python
def find_most_similar_words(self, vector, topn):
    similarities = []
    for i in range(len(self.vocabulary)):
        vec = np.array(self.vectors[i])
        cosine = np.dot(vector, vec) / (np.sqrt(np.dot(vector, vector)) * np.sqrt(np.dot(vec, vec)))
        similarities.append((self.vocabulary.get_token(i), cosine))
    return sorted(similarities, key=lambda x: -x[1])[:topn]

def analogy(self, x1, x2, y1, topn=5, keep_all=False):
    # Compute the analogy vector
    x1_vec = np.array(self.vectors[self.vocabulary.get_index(x1)])
    x2_vec = np.array(self.vectors[self.vocabulary.get_index(x2)])
    y1_vec = np.array(self.vectors[self.vocabulary.get_index(y1)])
    analogy_vec = y1_vec + x2_vec - x1_vec

    # Return all words if keep_all is True, otherwise return topn most similar words
    if keep_all:
        return self.find_most_similar_words(analogy_vec, len(self.vocabulary))
    else:
        return self.find_most_similar_words(analogy_vec, topn)
```

2.2 Intrinsic Evaluation

In this next section we will study how good or bad these methods performs with several examples. We will try using synonyms, antonyms and also words that have multiple meanings and bias to see how the most similar words and analogies perform.

Firstly, in the case of polysemic words, we have seen that overall only a meaning is considered. For example, for the word *clau* there appear no words related to the concept of house key, as it's only being inferred as something *essencial, fonamental, concret, bàsic, crucial, ...* . Similarly, for the word *copa*, we see that it only refers to the trophy given in certain competitions (*lliga, competició, supercopa, cursa, marató, ...*) while it ignores the meaning of cup or glass.

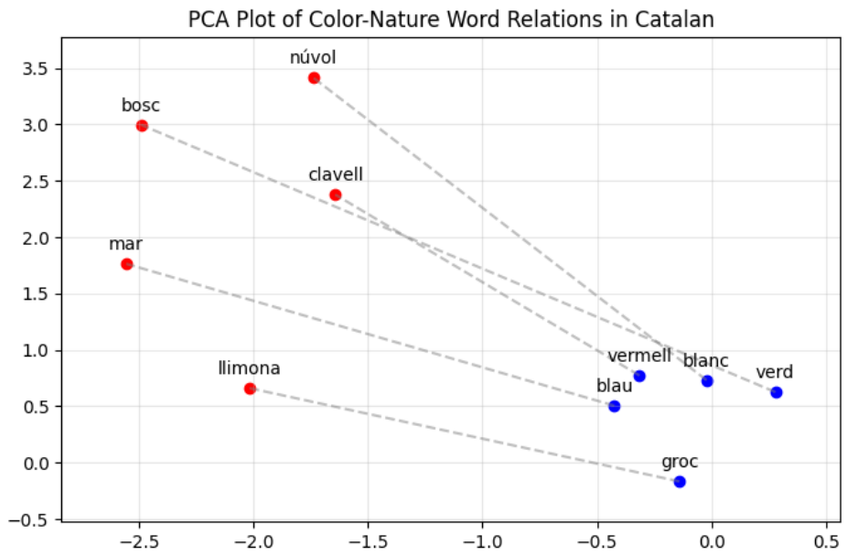
Moreover, we have checked for words that could induce gender discrimination. We have been surprised by the finding of the most similar words to *noi* and *noia*. We see the method returns words like *nen, nado, rodamón, bebè, dimoni, ...* for the first, while it returns words like *nena, bruixa, prostituta, criatura, fada, ...* for the second, being witch and prostitute the second and third words with the highest score. These scores are surprisingly high and suggest the introduction of a gender discriminatory bias.

As for the analogies, we see that in some cases they perform really well, as for example for the given set *futbol, futbolista, tennis*, it returns *tennista*. Or for *ballar, ball, cantar* it returns *cant* as the top 5 word, which is quite good. However, it's not able to find the relation between antonyms and performs poorly in this sense. We can see this in the following example *sol, dia, lluna*, which does not return the expected word *nit* but non-related words such as *comentar, setmana, temporada, ...*, which make no sense in this specific context.

Finally, we conclude that while both methods generally perform well in providing synonyms for input words, our evaluation reveals limitations. The model struggles a lot with antonyms, as it's not able to return any single good example, words with multiple meanings, and also with certain given examples, where we see the introduction of discriminatory biases.

2.3 Visualization and word clustering properties

In order to properly visualize how the model automatically organizes concepts and learns implicitly the relationships between them, we have plotted a two-dimensional PCA projection of several word embeddings, focusing specifically on color-nature word relations. On the following plot we can see the following five colors: *groc, blau, vermell, blanc, verd* and their respective nature-related pairs: *llimona, mar, clavell, núvol, bosc*.



From this plot we clearly observe two distinct clusters formed by words representing related concepts: colors and nature-related elements, which appear grouped together, respectively. Moreover, the cluster of colors appears more tightly grouped compared to the cluster of natural elements, primarily due to colors being relatively well-defined and have a narrower range of meanings, leading to a more cohesive cluster in the PCA plot. On the other hand, natural elements cover a wider range of ideas, such as landscapes and flora, leading to a more spread-out cluster because these concepts can vary in their connections and clarity.

Furthermore, colors appear closer to each other within their cluster than to their respective pairs, indicating that the two clusters themselves are somewhat distant. This suggests that a color is more closely related to any other color within its own cluster than to its paired natural element. In other words, colors exhibit stronger intra-cluster relationships compared to inter-cluster relationships with their corresponding natural elements.

Despite that, the model still demonstrates an effective ability to recognize the words and their relationships. This implies that, although there may be some separation between the concepts of colors and natural elements in the semantic space, the model is still capable of discerning and capturing the inherent associations between words within each cluster. Hence, despite the apparent separation, the model's performance in recognizing and representing the semantic relationships between words remains robust.