Word Vectors
Training and Analysis

Spoken and Written Language Processing

Data Science and Engineering
Polytechnic University of Catalonia

Claudia Len Manero - 53869554
Alina Castell Blasco - 49188689S

March 2024

1 Improve CBOW model

This section was divided in 5 parts and we will analyze the results of each subsection individually to better understand the results.

We trained each modification in 4 epochs and with the same hyperparameters, except for the last one which search for the best hyperparameters.

1.1 Fixed scalar weight for words closer to the predicted central word

For this modification of the CBOW model we used *register_buffer*() function to add the unmodifiable *position_weight* parameter as shown in the following code snippet.

```
class CBOWa(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)

        # Doesn't store as parameters, so they can't be modified by the model
        self.register_buffer('position_weight', torch.tensor([1,2,3,3,2,1],
            dtype=torch.float32))

    # 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)
        # get defined fixed weights
        e = e*self.position_weight.view(1, -1, 1)
        # 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
```

This model resulted in a final training accuracy of 33.6% and a loss of 4.34.

Comparatively, the initial accuracy was a mere 4%, accompanied by a loss of 10.59, reflecting a substantial improvement of +29.6% in accuracy and a notable reduction of -6.25 in loss.

It's crucial to analyze the evolution of this improvement across various training epochs. The most remarkable progress occurred in the first epoch, witnessing a leap from 4% to 29% in accuracy and a reduction in loss from 10.59 to 5. This aligns with expectations, as accuracy typically exhibits this type of behavior, rapidly approaching optimal values. The second epoch also showed improvement, although it was more discrete. It was in the 3rd and 4th epochs that concerns were raised as the accuracy ultimately fell below the starting accuracy for each epoch. This indicates inefficiency during these stages, suggesting potential enhancements for the model to consistently achieve optimal accuracy instead of fluctuating around it.

Validation metrics of this model were:

Dataset	Accuracy	Loss
Wikipedia	31.3%	4.56
El Periodico	21.3%	5.61

As the wikipedia dataset was used for training it was expected that validation metrics were higher.

The first 10 predicted tokens by this model were:

1	l
2	haver
3	haver
4	hi
5	un
6	l
7	un
8	la
9	principis
10	s

1.2 Trained scalar weights for each position

For this section we used trainable weights for each position.

```
    class CBOWb(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)

        # Assignment of trainable weights
        self.position_weight = nn.Parameter(torch.tensor([1,2,3,3,2,1],
            dtype=torch.float32))

    # 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)
        # get defined weights
        e = e*self.position_weight.view(1, -1, 1)
        # 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
```

Starting with accuracy 3,9% and loss 10,54 and ending with accuracy 35,4% and 4,16 loss. A significant improve of +31.5% and -6.38. Not far from the results of the first section but giving a noticeable improvement.

During the first epoch as it happened in the first section, accuracy started low and loss high and rapidly evolved to better values.

Second and third epoch has a slight improvement but not remarkable and for the last epoch the same concerning behaviour described in the first section was observed. Starting at 35,5% and ending at 35,4%. It is not a considerable decrease but it still means the model could be improved.

Validation metrics of this model were:

Dataset	Accuracy	Loss
Wikipedia	33.3%	4.37
El Periodico	22.8%	5.37

Once again metrics look good as the wikipedia dataset accuracy is closer to the training one.

The first 10 predicted tokens by this model were:

1	l
2	s
3	començar
4	s
5	un
6	l
7	un
8	la
9	mitjans
10	de

1.3 Trained vector weight for each position element-wise multiplied by the corresponding position-dependent vector of weights.

In this third section we added a position-dependent factor to the weights:

```
    class CBOWc(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)
        # Definition of the position weight parameter
        self.position_weight =
            nn.Parameter(torch.tensor(torch.rand(1,6,100),
            dtype=torch.float32))
    # 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)
        # get defined weights
        e = e*self.position_weight
        # 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
```

For this version of the model, we obtained a final accuracy of 43.3% and training loss of 3.48 that clearly shows a notable improvement respect to the previous versions, which only reached 35% accuracy. In this case accuracy increased a +39.4% and loss decreased -5.95.

In the first epoch as all previous models there was a rapid increase in accuracy going from a starting 3.9% to a 37.4%, which is a 859% increase, much higher than the previous 695%.

During the second epoch accuracy only increased 1%, a subtle improvement as we also saw in other versions of the model. The difference was in the 3rd and 4th epoch where accuracy didn't decrease as it happened in the previous sections, instead it got stuck in the initial value, which isn't good either.

Validation metrics of this model were:

Dataset	Accuracy	Loss
Wikipedia	41.2%	3.67
El Periodico	31%	4.58

The first 10 predicted tokens by this model were:

1	l
2	s
3	haver
4	s
5	un
6	l
7	aquest
8	La
9	desenvolupament
10	fet

1.4 Hyperparameter optimization

To try to improve the last results we decided to analyze the models behaviour with different hyper-parameter values.

We trained the *embedding_dim* with 50, 100 and 200 using a parameter grid.

```
class CBOWd(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.tensor(torch.rand(1,6,embedding_dim),
            dtype=torch.float32))
    # 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)
        # get defined weights (exercise a)
        e = e*self.position_weight
        # 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

    from sklearn.model_selection import ParameterGrid

param_grid = {
        'embedding_dim': [50, 100, 200],  # Adjust as needed
    }

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
i=0
for parameters in ParameterGrid(param_grid):
    i+=1
    model_c = CBOWd(len(vocab), parameters['embedding_dim']).to(device)  #
        Reset model parameters for each configuration
    optimizer = torch.optim.Adam(model_c.parameters())
    criterion = nn.CrossEntropyLoss(reduction='sum')
    for epoch in range(params.epochs):
        acc, loss = train(model_c, 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_c, 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_c, criterion, valid_x, valid_y,
            parames.batch_size, device)
        valid_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid
            loss={loss:.2f} (El Peri dico)')
```

For *embedding_dim* = 50 accuracy ended at 40.5% and train loss was 3.78.

Validation metrics of this model were:

Dataset	Accuracy	Loss
Wikipedia	38.6%	3.95
El Periodico	28.6%	4.84

The first 5 predicted tokens by this model were:

1	l
2	s
3	haver
4	s
5	un

This doesn't suppose an improvement from the other model versions, so we still prefer the last one explained.

For *embedding_dim* = 100 results were better with a train accuracy of 43.4% and a train loss of 3.48.

Validation parameters:

Dataset	Accuracy	Loss
Wikipedia	41.3%	3.66
El Periodico	31%	4.59

This was the value used in the 3 first implementations of the model, so we already knew the results.

The first 5 tokens predicted by the model were:

Lastly, for *embedding_dim* = 200 there was an increase in accuracy that indicates an improvement of the model, making clear that the best value for *embedding_dim* is 200 instead of 100, which was used in the other implementations.

Train accuracy reached 46% which is a +2.7% improve from the 3rd version of the model we analyzed and loss 3.14, which is also lower. So we can state that this is the most efficient model for now.

Validation parameters were better too, giving:

For now this is the best model, but we also tried changing *batch_size* to see if this would cause an improvement and used 500, 1000, 2000.

```
    from sklearn.model_selection import ParameterGrid

param_grid = {
        'batch_size': [500, 1000, 2000],  # Adjust as needed
    }

train_accuracy = []
wiki_accuracy = []
valid_accuracy = []
i=0
for parameters in ParameterGrid(param_grid):
```

```
1    l
2    s
3  haver
4    s
5   un
```

Dataset	Accuracy	Loss
Wikipedia	43.2%	3.53
El Periodico	33.1%	4.71

```
    i+=1
    model_c = CBOWd(len(vocab), params.embedding_dim).to(device)  # Reset model parameters for each configuration
    optimizer = torch.optim.Adam(model_c.parameters())
    criterion = nn.CrossEntropyLoss(reduction='sum')
    for epoch in range(params.epochs):
        acc, loss = train(model_c, criterion, optimizer, data[0][0], data[0][1], parameters['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_c, criterion, data[1][0], data[1][1], parameters['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_c, criterion, valid_x, valid_y, parameters['batch_size'], device)
        valid_accuracy.append(acc)
        print(f'| epoch {epoch:03d} | valid accuracy={acc:.1f}%, valid loss={loss:.2f} (El Peri dico)')

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

    y_pred = validate(model-c, None, test_x, None, parameters['batch_size'], device)
    y_token = [vocab.idx2token[index] for index in y_pred]
    submission = pd.DataFrame({'id':valid_x_df['id'], 'token': y_token}, columns=['id', 'token'])
    print(submission.head())
    name = 'submission' + str(i) + '.csv'
    submission.to_csv(name, index=False)
```

For *batch_size* = 500 we obtained an accuracy of 42.9% and loss of 3.63 which is getting closer to the best results obtained but still not better.

Dataset	Accuracy	Loss
Wikipedia	40.8%	3.8
El Periodico	30.8%	4.71

For *batch_size* = 1000 we obtained an accuracy of 43.9% and loss of 3.33 which is getting closer to the best results obtained but still not better.

Dataset	Accuracy	Loss
Wikipedia	41.8%	3.57
El Periodico	31.6%	4.81

The first 5 tokens predicted by the model were:

1	l
2	s
3	haver
4	s
5	un

Finally, for *batch_size* = 2000 we got an accuracy of 44% and loss 3.35.

We would have wanted to try to optimize more hyperparameters as *num_embeddings* or the optimizer used.

1.5 Other models

Finally we implemented another method for word embedding creation.

```
    class TransformerWordEmbedding(nn.Module):
    def __init__(self, num_embeddings, embedding_dim, num_heads=8,
        hidden_dim=512, num_layers=6):
        super(TransformerWordEmbedding, self).__init__()

        self.embedding = nn.Embedding(num_embeddings, embedding_dim,
            padding_idx=0)
        self.transformer_encoder = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=embedding_dim,
                nhead=num_heads, dim_feedforward=hidden_dim),
            num_layers=num_layers
        )
        self.fc = nn.Linear(embedding_dim, num_embeddings)

    def forward(self, input):
        # input shape is (B, W)
        e = self.embedding(input)
        # e shape is (B, W, E)

        # Permute for transformer input format
        transformer_input = e.permute(1, 0, 2)

        # Transformer Encoder
        transformer_output = self.transformer_encoder(transformer_input)

        # Take the output from the last layer and transpose back
        transformer_output = transformer_output[-1].permute(1, 0, 2)

        # Global Average Pooling
        pooled = torch.mean(transformer_output, dim=1)

        z = self.fc(pooled)
        # z shape is (B, V)
        return z
```

This implementation, although more complex, didn't provide better results giving a train accuracy of 31.9% and train loss of 4.43.

Dataset	Accuracy	Loss
Wikipedia	31.5%	4.49
El Periodico	22.5%	5.4

The first 5 tokens predicted by the model were:

1	l
2	s
3	construcció
4	no
5	se

Results were 10% lower than our best resulting model, which means this method wasn't a better alternative to the CBOW model.

2 Analyse the generated word vectors

2.1 WordVector class implementation

```
# Find closest vectors and analogies using the cosine similarity measure.
class WordVectors:
    def __init__(self, vectors, vocabulary):
        self.vectors = vectors
        self.vocabulary = vocabulary
        self.vector_norms = np.linalg.norm(vectors, axis=1)

    def most_similar(self, word, topn=10):
        # Ids and vectors of input word
        word_ids = self.vocabulary.get_index(word)
        word_vector = self.vectors[word_ids]

        # Calculate cosine similariy between target word vector and the rest
            of word vectors
        similarities = np.dot(self.vectors,word_vector)/ (self.vector_norms
            * np.linalg.norm(word_vector))

        # Get the indices of the top n most similar words
        top_indices = np.argsort(similarities)[::-1][:topn]

        # Create a list of (word, similarity) tuples for the top n most similar words
        similar_words = [(self.vocabulary.get_token(i), similarities[i]) for i in top_indices if i != word_ids and i != self.vocabulary.get_index('<pad>')]

        return similar_words


    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
        # Ids and vectors of input words
        x1_id = self.vocabulary.get_index(x1)
        x2_id = self.vocabulary.get_index(x2)
        y1_id = self.vocabulary.get_index(y1)

        x1_vector = self.vectors[x1_id]
        x2_vector = self.vectors[x2_id]
        y1_vector = self.vectors[y1_id]

        # Apply formula
        analogy_vector = y1_vector + (x2_vector - x1_vector)

        # Calculate cosine similariy between target word vector and the rest of word vectors
        similarities = np.dot(self.vectors, analogy_vector) / (self.vector_norms * np.linalg.norm(analogy_vector))

        # Get the indices of words
        indices = np.argsort(similarities)[::-1]

        if keep_all:
            # Similar words contains all words
            similar_words = [(self.vocabulary.get_token(i), similarities[i]) for i in indices if i not in (x1_id, x2_id, y1_id)]
        else:
            # Similar words contains only the top n
            similar_words = [(self.vocabulary.get_token(i), similarities[i]) for i in indices if i not in (x1_id, x2_id, y1_id)][:topn]

        return similar_words
```

2.2 Intrinsic evaluation

After implementing the WordVector class with the *most_similar* and *analogy* methods using the cosine similarity measure, we are now able to perform an informal evaluation of both methods.

First, we perform evaluation for the *most_similar* method. After, many trials we have decided to analyse the three examples shown in the next table because of the difference in their outputs, they are ordered from good to bad example of closest words.

WordVector class most_similar method				
'violeta'	'dades'	'girasol'	'home'	'dona'
[('porpra', 0.76075983), ('morat', 0.75984585), ('marró', 0.74861825), ('púrpura', 0.7224956), ('verd', 0.7220345), ('lila', 0.7151704), ('rosa', 0.71447676), ('rosat', 0.70139754)]	[('informacions', 0.7611591), ('notícies', 0.69987947), ('conclusions', 0.6970468), ('mostres', 0.6964814), ('novetats', 0.6952016), ('notes', 0.68635607), ('signatures', 0.68160015), ('cites', 0.67419755)]	[('Bar', 0.5264481), ('Gallo', 0.5070498), ('Pous', 0.4835284), ('Alba', 0.48207828), ('Schmidt', 0.47576365), ('Scarlatti', 0.47018117), ('Ideas', 0.46398053), ('Doria', 0.4590072)]	[('noi', 0.6846899), ('individu', 0.68141776), ('animal', 0.65379584), ('esclau', 0.64577675), ('ocell', 0.6063776), ('esperit', 0.58602643), ('adolescent', 0.5859838), ('espectador', 0.5824114)]	[('noia', 0.6471737), ('nena', 0.6124585), ('mare', 0.5958722), ('dama', 0.587081), ('núvia', 0.5695971), ('metgessa', 0.5585767), ('nina', 0.5413984), ('serventa', 0.53372526)]

Table 1: Most Similar method

We have selected the term 'violeta' due to its dual interpretation, which can be perceived either as a flower or a color. Our aim was to assess whether the method would prioritize one interpretation over the other in the similarity function. The output reveals that all words similar to 'violeta' are colors, boasting the highest scores among the sample words. This indicates that the method has effectively captured words similar to the sample term.
Regarding the term 'dades' the first reason to choose this word is because of the wide range of definitions it has (eventhough all highly related). Secondly, its direct relevance to our field of study influenced our choice. It can be observed that the word 'informacions' has the highest score of all, which aligns with its status as a direct synonym. While other words are also considerably similar, none of them stands out as shown on their scores.
Moving on to the term 'girasol,' it serves as a clear example of inadequacy. Despite its thematic connection to the initial term, our observation quickly revealed the method's failure to function accurately with it. The identified similar words for 'girasol' are proper nouns bearing no relevance to the term, accompanied by notably lower scores compared to others.
Moving on to the term 'girasol' is serves as a clear bad example. Despite its topic relation to the initial term, we rapidly saw that the similarity method failed to work with it. The identified similar words to 'girasol' are proper nouns bearing no relevance to the term, along with notably lower scores compared to others.
Lastly, we have observed discrepancies between the similar words 'home' and 'dona' in most cases, in fact, the only word that align are 'noi' and 'noia', respectively. The similarity words may be gender biased because of the words 'mare' and 'serventa' that appear in the 'dona' list of similar words.

Secondly, we perform evaluation for the *analogy* method. We have chosen the words ['dos', '2', 'tres'] and ['rei', 'reina', 'doctor'], the analogies of which we expected to be '3' and 'doctora', respectively. Indeed, the obtained results confirmed our expectations, validating the successful implementation of the method.

2.3 Word analogies and word clustering visualisation

Now that the *most_similar* and *analogy* methods have been examined, we can illustrate their relationships through PCA and clustering. Initiating with the visualization of word analogies, we present the PCA plot:

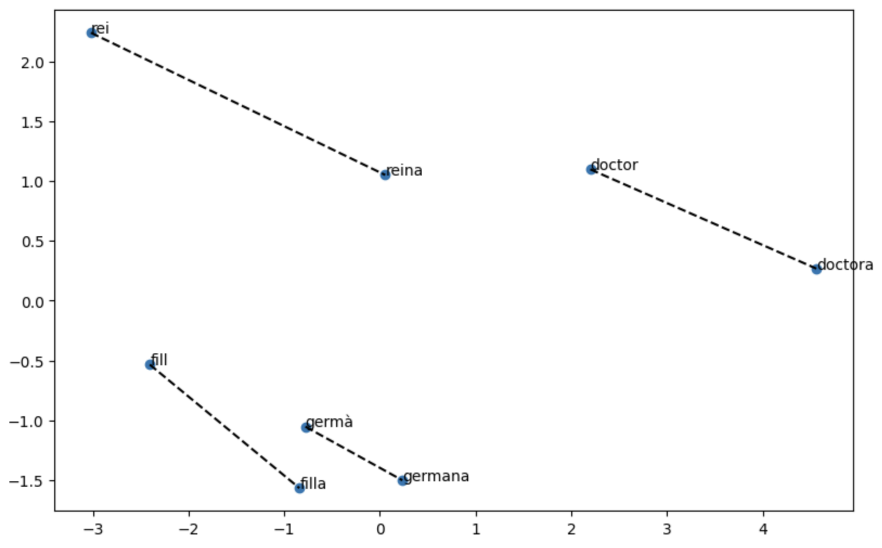

Figure 1: Analogy visualisation with PCA components

From this figure, it is evident that words sharing a similar theme are positioned at similar angles within the plot, such as ['rei','reina'] and ['doctor','doctora'], as well as ['fill','filla'] and ['germà', 'germana'].

Moving on to the clustering visualisation we have studied the topics of colours, relatives and insects, along with, antonyms, shown in the following plots:

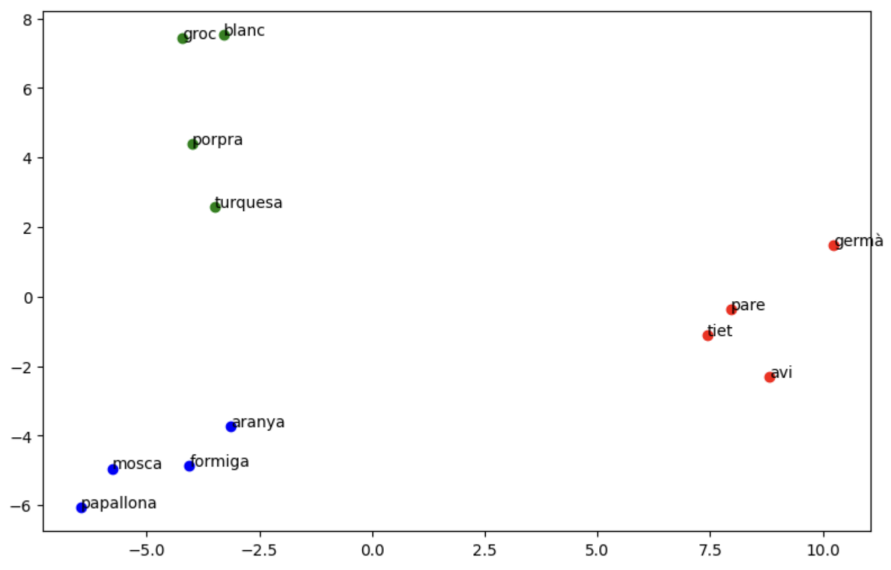

Figure 2: *most_similar* cluster colours, relatives and insects

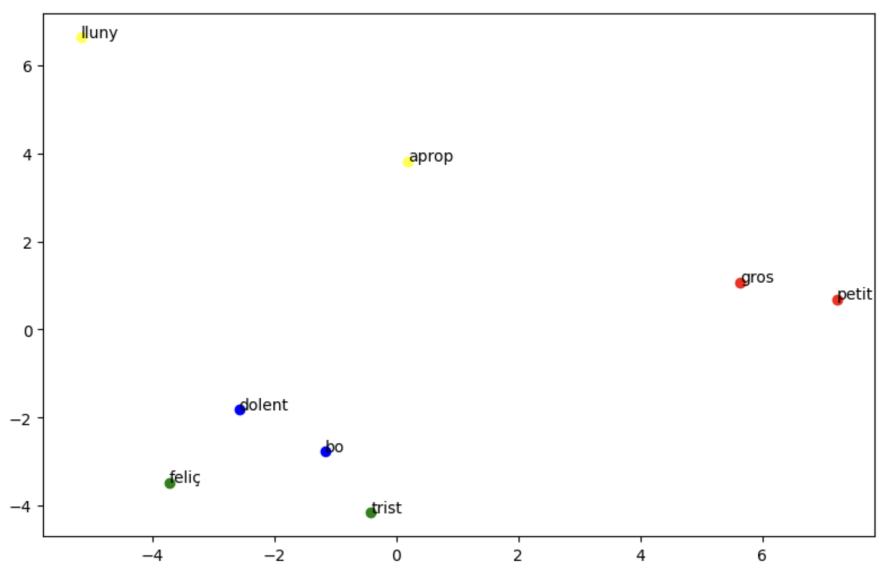

Figure 3: *most_similar* cluster antonyms

The topic groups show clear divisions, with insects positioned in the lower-left quadrant, colours situated in the upper-left, and relatives positioned in the middle-right. Notably, the subgroup comprising the rarest colors, 'porpra' and 'turquesa', stands out distinctively, while the remaining colors are visibly clustered together.

Conversely, the antonym groups appear to have greater difficulty in maintaining significant distances between them. 'Gros' and 'petit' are the most distinguishable, located close to each other yet separated from the others. 'Lluny' and 'aprop' demonstrate a considerable distance from one another while maintaining separation from the rest. Meanwhile, 'bo', 'dolent', 'feliç', and 'trist' exhibit some degree of association, while keeping at a distance from the other two groups.

2.4 Prediction accuracy

We are now comparing our best submission, namely the one that has achieved the highest accuracy, with the one that was given. The given values are:

- valid accuracy=24.0% valid loss=5.12 (wikipedia)
- valid accuracy=15.5%, valid loss=6.10 (El Periódico)

Our best values are obtained using a size of embedding 200 and the CBOW model:

- valid accuracy=43.2% valid loss=3.53 (wikipedia)
- valid accuracy=33.1%, valid loss=4.71 (El Periódico)

Which indeed show a better result than using the CBOW model with a size of the embeddings of 100.