Using Pre-trained Word Embeddings

In this notebook we will show some operations on pre-trained word embeddings to gain an intuition about them.

We will be using the pre-trained GloVe embeddings that can be found in the official website (https://nlp.stanford.edu/projects/glove/). In particular, we will use the file `glove.6B.300d.txt` contained in this zip file (https://nlp.stanford.edu/data/glove.6B.zip).

We will first load the GloVe embeddings using Gensim (https://radimrehurek.com/gensim/). Specifically, we will use `KeyedVectors` (https://radimrehurek.com/gensim/models/keyedvectors.html)'s `load_word2vec_format()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.load_w classmethod, which supports the original word2vec file format. However, there is a difference in the file formats used by GloVe and word2vec, which is a header used by word2vec to indicate the number of embeddings and dimensions stored in the file. The file that stores the GloVe embeddings doesn't have this header, so we will have to address that when loading the embeddings.

Loading the embeddings may take a little bit, so hang in there!

In [2]:
```
!pip install gensim
```

```
Collecting gensim
  Downloading gensim-4.2.0-cp37-cp37m-win_amd64.whl (24.0 MB)
     --------------------------------------- 24.0/24.0 MB 12.1 MB/s eta 0:00:00
Requirement already satisfied: scipy>=0.18.1 in c:\users\luism\.conda\envs\rstudio\lib\sit
e-packages (from gensim) (1.7.3)
Collecting smart-open>=1.8.1
  Downloading smart_open-7.0.5-py3-none-any.whl (61 kB)
     ---------------------------------------- 61.4/61.4 kB ? eta 0:00:00
Collecting Cython==0.29.28
  Downloading Cython-0.29.28-py2.py3-none-any.whl (983 kB)
     ------------------------------------- 983.8/983.8 kB 30.4 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17.0 in c:\users\luism\.conda\envs\rstudio\lib\sit
e-packages (from gensim) (1.21.6)
Requirement already satisfied: wrapt in c:\users\luism\.conda\envs\rstudio\lib\site-packag
es (from smart-open>=1.8.1->gensim) (1.16.0)
Installing collected packages: smart-open, Cython, gensim
Successfully installed Cython-0.29.28 gensim-4.2.0 smart-open-7.0.5
```

In [3]:
```
from gensim.models import KeyedVectors

fname = "glove.6B.300d.txt"
glove = KeyedVectors.load_word2vec_format(fname, no_header=True)
glove.vectors.shape
```

Out[3]:
```
(400000, 300)
```

Word similarity

One attribute of word embeddings that makes them useful is the ability to compare them using cosine similarity to find how similar they are. `KeyedVectors` (https://radimrehurek.com/gensim/models/keyedvectors.html) objects provide a method called `most_similar()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_s that we can use to find the closest words to a particular word of interest. By default, `most_similar()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_s returns the 10 most similar words, but this can be changed using the `topn` parameter.

Below we test this function using a few different words.

In [4]:
```
# common noun
glove.most_similar("cactus")
```

Out[4]:
```
[('cacti', 0.663456380367279),
 ('saguaro', 0.619585394859314),
 ('pear', 0.5233487486839294),
 ('cactuses', 0.5178282260894775),
 ('prickly', 0.5156316757202148),
 ('mesquite', 0.48448559641838074),
 ('opuntia', 0.45400843024253845),
 ('shrubs', 0.45362070202827454),
 ('peyote', 0.45344963669776917),
 ('succulents', 0.4512787461280823)]
```

In [5]:
```
# common noun
glove.most_similar("cake")
```

Out[5]:
```
[('cakes', 0.7506032586097717),
 ('chocolate', 0.6965583562850952),
 ('dessert', 0.6440261602401733),
 ('pie', 0.6087430119514465),
 ('cookies', 0.6082393527030945),
 ('frosting', 0.6017215251922607),
 ('bread', 0.5954801440238953),
 ('cookie', 0.5933820009231567),
 ('recipe', 0.5827102065086365),
 ('baked', 0.5819962620735168)]
```

In [6]:
```
# adjective
glove.most_similar("angry")
```

Out[6]:
```
[('enraged', 0.7087872624397278),
 ('furious', 0.7078357338905334),
 ('irate', 0.6938743591308594),
 ('outraged', 0.6705202460289001),
 ('frustrated', 0.6515548229217529),
 ('angered', 0.635320186614902),
 ('provoked', 0.5827428102493286),
 ('annoyed', 0.5818981528282166),
 ('incensed', 0.5751834511756897),
 ('indignant', 0.5704444646835327)]
```

In [7]:
```
# adverb
glove.most_similar("quickly")
```

Out[7]:
```
[('soon', 0.7661858797073364),
 ('rapidly', 0.7216639518737793),
 ('swiftly', 0.7197348475456238),
 ('eventually', 0.7043027281761169),
 ('finally', 0.6900883316993713),
 ('immediately', 0.684260904788971),
 ('then', 0.6697486042976379),
 ('slowly', 0.6645646095275879),
 ('gradually', 0.6401676535606384),
 ('when', 0.634766697883606)]
```

In [8]:
```
# preposition
glove.most_similar("between")
```

Out[8]:
```
[('sides', 0.5867609977722168),
 ('both', 0.5843431949615479),
 ('two', 0.5652361512184143),
 ('differences', 0.5140715837478638),
 ('which', 0.5120178461074829),
 ('conflict', 0.5115456581115723),
 ('relationship', 0.5022750496864319),
 ('and', 0.49842509627342224),
 ('in', 0.4970666468143463),
 ('relations', 0.49701136350631714)]
```

In [9]:
```
# determiner
glove.most_similar("the")
```

Out[9]:
```
[('of', 0.7057957053184509),
 ('which', 0.6992015242576599),
 ('this', 0.6747024655342102),
 ('part', 0.6727458238601685),
 ('same', 0.6592391133308411),
 ('its', 0.6446542143821716),
 ('first', 0.6398990750312805),
 ('in', 0.6361347436904907),
 ('one', 0.6245333552360535),
 ('that', 0.6176422834396362)]
```

Word analogies

Another characteristic of word embeddings is their ability to solve analogy problems. The same `most_similar()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_s method can be used for this task, by passing two lists of words: a `positive` list with the words that should be added and a `negative` list with the words that should be subtracted. Using these arguments, the famous example $\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$ can be executed as follows:

In [10]:
```python
# king - man + woman
glove.most_similar(positive=["king", "woman"], negative=["man"])
```

Out[10]:
```
[('queen', 0.6713277101516724),
 ('princess', 0.5432625412940979),
 ('throne', 0.5386105179786682),
 ('monarch', 0.5347574353218079),
 ('daughter', 0.49802514910697937),
 ('mother', 0.49564430117607117),
 ('elizabeth', 0.483265221118927),
 ('kingdom', 0.47747087478637695),
 ('prince', 0.4668240249156952),
 ('wife', 0.4647327959537506)]
```

Here are a few other interesting analogies:

In [11]:
```python
# car - drive + fly
glove.most_similar(positive=["car", "fly"], negative=["drive"])
```

Out[11]:
```
[('airplane', 0.5897148847579956),
 ('flying', 0.5675230026245117),
 ('plane', 0.53170245885849),
 ('flies', 0.5172374844551086),
 ('flown', 0.514790415763855),
 ('airplanes', 0.5091356635093689),
 ('flew', 0.5011662244796753),
 ('planes', 0.4970923364162445),
 ('aircraft', 0.4957723915576935),
 ('helicopter', 0.45859551429748535)]
```

In [12]:
```python
# berlin - germany + australia
glove.most_similar(positive=["berlin", "australia"], negative=["germany"])
```

Out[12]:
```
[('sydney', 0.6780861616134644),
 ('melbourne', 0.6499180197715759),
 ('australian', 0.5948832035064697),
 ('perth', 0.5828553438186646),
 ('canberra', 0.5610731840133667),
 ('brisbane', 0.55231112241745),
 ('zealand', 0.524011492729187),
 ('queensland', 0.5193883776664734),
 ('adelaide', 0.5027670860290527),
 ('london', 0.4644603729248047)]
```

In [13]:
```
# england - london + baghdad
glove.most_similar(positive=["england", "baghdad"], negative=["london"])
```

Out[13]:
```
[('iraq', 0.5320571660995483),
 ('fallujah', 0.48340919613838196),
 ('iraqi', 0.47287359833717346),
 ('mosul', 0.46466362476348877),
 ('iraqis', 0.43555372953414917),
 ('najaf', 0.43527641892433167),
 ('baqouba', 0.4206319749355316),
 ('basra', 0.4190516471862793),
 ('samarra', 0.41253671050071716),
 ('saddam', 0.4079156517982483)]
```

In [14]:
```
# japan - yen + peso
glove.most_similar(positive=["japan", "peso"], negative=["yen"])
```

Out[14]:
```
[('mexico', 0.5726831555366516),
 ('philippines', 0.5445370078086853),
 ('peru', 0.48382270336151123),
 ('venezuela', 0.48166725039482117),
 ('brazil', 0.4664310216903686),
 ('argentina', 0.45490506291389465),
 ('philippine', 0.44178417325019836),
 ('chile', 0.4396097958087921),
 ('colombia', 0.4386259913444519),
 ('thailand', 0.43396779894828796)]
```

In [15]:
```
# best - good + tall
glove.most_similar(positive=["best", "tall"], negative=["good"])
```

Out[15]:
```
[('tallest', 0.5077418684959412),
 ('taller', 0.47616493701934814),
 ('height', 0.46000057458877563),
 ('metres', 0.4584785997867584),
 ('cm', 0.4521271884441376),
 ('meters', 0.44067251682281494),
 ('towering', 0.42784246802330017),
 ('centimeters', 0.42345425486564636),
 ('inches', 0.41745859384536743),
 ('erect', 0.4087313711643219)]
```

Looking under the hood

Now that we are more familiar with the `most_similar()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_s method, it is time to implement its functionality ourselves. But first, we need to take a look at the different parts of the `KeyedVectors` (https://radimrehurek.com/gensim/models/keyedvectors.html) object that we will need. Obviously, we will need the vectors themselves. They are stored in the `vectors` attribute.

In [16]:
```
glove.vectors.shape
```

Out[16]:
```
(400000, 300)
```

As we can see above, `vectors` is a 2-dimensional matrix with 400,000 rows and 300 columns. Each row corresponds to a 300-dimensional word embedding. These embeddings are not normalized, but normalized embeddings can be obtained using the [`get_normed_vectors()` (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.get_no](https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.get_no) method.

In [17]:
```
normed_vectors = glove.get_normed_vectors()
normed_vectors.shape
```

Out[17]:
```
(400000, 300)
```

Now we need to map the words in the vocabulary to rows in the `vectors` matrix, and vice versa. The [`KeyedVectors` (https://radimrehurek.com/gensim/models/keyedvectors.html)](https://radimrehurek.com/gensim/models/keyedvectors.html) object has the attributes `index_to_key` and `key_to_index` which are a list of words and a dictionary of words to indices, respectively.

In [18]:
```
#glove.index_to_key
# Lista de palabras en el vocabulario (por índice)
vocabulario = glove.index_to_key
print(vocabulario[:10])  # Esto mostrará las primeras 10 palabras
```

```
['the', ',', '.', 'of', 'to', 'and', 'in', 'a', '"', "'s"]
```

In [19]:
```
#glove.key_to_index
# Diccionario que mapea palabras a sus índices
mapa_indices = glove.key_to_index
print(list(mapa_indices.items())[:10])  # Esto mostrará los primeros 10 pares palabra-índic
```

```
[('the', 0), (',', 1), ('.', 2), ('of', 3), ('to', 4), ('and', 5), ('in', 6), ('a', 7),
('"', 8), ("'s", 9)]
```

Word similarity from scratch

Now we have everything we need to implement a `most_similar_words()` function that takes a word, the vector matrix, the `index_to_key` list, and the `key_to_index` dictionary. This function will return the 10 most similar words to the provided word, along with their similarity scores.

In [20]:
```
import numpy as np

def most_similar_words(word, vectors, index_to_key, key_to_index, topn=10):
    # Retrieve word_id corresponding to the given word
    if word not in key_to_index:
        return f"'{word}' not found in vocabulary."
    word_id = key_to_index[word]

    # Retrieve embedding for the given word
    word_vec = vectors[word_id]

    # Calculate similarities to all words in our vocabulary (hint: use @)
    similarities = np.dot(vectors, word_vec) / (np.linalg.norm(vectors, axis=1) * np.linalg

    # Get word_ids in ascending order with respect to similarity score
    word_ids = np.argsort(similarities)

    # Reverse word_ids to get the most similar words
    word_ids = word_ids[::-1]

    # Get a boolean array with the element corresponding to word_id set to false
    word_ids = word_ids[word_ids != word_id]

    # Get the topn word_ids
    top_word_ids = word_ids[:topn]

    # Retrieve topn words with their corresponding similarity score
    top_words = [(index_to_key[i], similarities[i]) for i in top_word_ids]

    # Return results
    return top_words
```

Now let's try the same example that we used above: the most similar words to "cactus".

In [21]:
```
vectors = glove.get_normed_vectors()
index_to_key = glove.index_to_key
key_to_index = glove.key_to_index
most_similar_words("cactus", vectors, index_to_key, key_to_index)
```

Out[21]:
```
[('cacti', 0.6634565),
 ('saguaro', 0.6195856),
 ('pear', 0.5233486),
 ('cactuses', 0.5178283),
 ('prickly', 0.5156319),
 ('mesquite', 0.48448554),
 ('opuntia', 0.45400843),
 ('shrubs', 0.45362073),
 ('peyote', 0.4534496),
 ('succulents', 0.4512787)]
```

Analogies from scratch

The `most_similar_words()` function behaves as expected. Now let's implement a function to perform the analogy task. We will give it the very creative name `analogy`. This function will get two lists of words (one for positive words and one for negative words), just like the `most_similar()`

(https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_s

In [22]:
```
from numpy.linalg import norm

def analogy(positive, negative, vectors, index_to_key, key_to_index, topn=10):
    # Find ids for positive and negative words
    pos_ids = [key_to_index[word] for word in positive if word in key_to_index]
    neg_ids = [key_to_index[word] for word in negative if word in key_to_index]

    # Combine the ids
    given_word_ids = pos_ids + neg_ids

    # Get embeddings for positive and negative words
    pos_emb = np.sum([vectors[i] for i in pos_ids], axis=0)
    neg_emb = np.sum([vectors[i] for i in neg_ids], axis=0)

    # Get embedding for analogy
    emb = pos_emb - neg_emb

    # Normalize embedding
    emb = emb / norm(emb)

    # Calculate similarities to all words in our vocabulary
    similarities = np.dot(vectors, emb) / (norm(vectors, axis=1) * norm(emb))

    # Get word_ids in ascending order with respect to similarity score
    ids_ascending = np.argsort(similarities)

    # Reverse word_ids to get the most similar words
    ids_descending = ids_ascending[::-1]

    # Get a boolean array with the element corresponding to any of given_word_ids set to fa
    given_words_mask = ~np.isin(ids_descending, given_word_ids)

    # Obtain new array of indices that doesn't contain any of the given_word_ids
    ids_descending = ids_descending[given_words_mask]

    # Get topn word_ids
    top_ids = ids_descending[:topn]

    # Retrieve topn words with their corresponding similarity score
    top_words = [(index_to_key[i], similarities[i]) for i in top_ids]

    # Return results
    return top_words
```

Let's try this function with the $\vec{king} - \vec{man} + \vec{woman} \approx \vec{queen}$ example we discussed above.

In [23]:
```
positive = ["king", "woman"]
negative = ["man"]
vectors = glove.get_normed_vectors()
index_to_key = glove.index_to_key
key_to_index = glove.key_to_index
analogy(positive, negative, vectors, index_to_key, key_to_index)
```

Out[23]:
```
[('queen', 0.67132765),
 ('princess', 0.5432624),
 ('throne', 0.5386105),
 ('monarch', 0.53475755),
 ('daughter', 0.4980251),
 ('mother', 0.49564433),
 ('elizabeth', 0.4832652),
 ('kingdom', 0.47747096),
 ('prince', 0.46682417),
 ('wife', 0.46473274)]
```