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. In particular, we will use the file `glove.6B.300d.txt` contained in this zip file.

We will first load the GloVe embeddings using Gensim. Specifically, we will use `KeyedVectors` 's `load_word2vec_format()` 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 [4]:
```
#pip install gensim
```

In [5]:
```
from gensim.models import KeyedVectors

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

Out[5]: (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` objects provide a method called `most_similar()` that we can use to find the closest words to a particular word of interest. By default, `most_similar()` 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 [6]:
```
# common noun
glove.most_similar("cactus")
```

Out[6]:
```
[('cacti', 0.6634564399719238),
 ('saguaro', 0.619585394859314),
 ('pear', 0.5233486890792847),
 ('cactuses', 0.5178281664848328),
 ('prickly', 0.5156318545341492),
 ('mesquite', 0.4844855070114136),
 ('opuntia', 0.4540084898471832),
 ('shrubs', 0.45362064242362976),
 ('peyote', 0.45344963669776917),
 ('succulents', 0.4512787461280823)]
```

In [7]:
```
# common noun
glove.most_similar("cake")
```

Out[7]:
```
[('cakes', 0.7506030201911926),
 ('chocolate', 0.6965583562850952),
 ('dessert', 0.6440261006355286),
 ('pie', 0.6087430119514465),
 ('cookies', 0.6082394123077393),
 ('frosting', 0.6017215251922607),
 ('bread', 0.5954802632331848),
 ('cookie', 0.5933820009231567),
 ('recipe', 0.5827102065086365),
 ('baked', 0.5819962620735168)]
```

In [8]:
```
# adjective
glove.most_similar("angry")
```

Out[8]:
```
[('enraged', 0.7087873816490173),
 ('furious', 0.7078357934951782),
 ('irate', 0.6938743591308594),
 ('outraged', 0.6705204248428345),
 ('frustrated', 0.6515549421310425),
 ('angered', 0.635320246219635),
 ('provoked', 0.5827428102493286),
 ('annoyed', 0.581898033618927),
 ('incensed', 0.5751833319664001),
 ('indignant', 0.5704444646835327)]
```

In [9]:
```
# adverb
glove.most_similar("quickly")
```

Out[9]:
```
[('soon', 0.7661860585212708),
 ('rapidly', 0.7216639518737793),
 ('swiftly', 0.7197349667549133),
 ('eventually', 0.7043026685714722),
 ('finally', 0.6900882124900818),
 ('immediately', 0.6842609643936157),
 ('then', 0.6697486042976379),
 ('slowly', 0.6645646095275879),
 ('gradually', 0.6401676535606384),
 ('when', 0.6347666382789612)]
```

In [10]:
```
# preposition
glove.most_similar("between")
```

Out[10]:
```
[('sides', 0.5867610573768616),
 ('both', 0.5843431949615479),
 ('two', 0.5652360916137695),
 ('differences', 0.5140716433525085),
 ('which', 0.5120178461074829),
 ('conflict', 0.511545717716217),
 ('relationship', 0.5022751092910767),
 ('and', 0.498425155878067),
 ('in', 0.4970666766166687),
 ('relations', 0.49701136350631714)]
```

In [11]:
```python
# determiner
glove.most_similar("the")
```

Out[11]:
```
[('of', 0.7057957053184509),
 ('which', 0.6992015242576599),
 ('this', 0.6747025847434998),
 ('part', 0.6727458834648132),
 ('same', 0.6592389941215515),
 ('its', 0.6446540355682373),
 ('first', 0.6398991346359253),
 ('in', 0.6361348032951355),
 ('one', 0.6245333552360535),
 ('that', 0.6176422834396362)]
```

Word analogies

Another characteristic of word embeddings is their ability to solve analogy problems. The same `most_similar()` 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 [12]:
```python
# king - man + woman
glove.most_similar(positive=["king", "woman"], negative=["man"])
```

Out[12]:
```
[('queen', 0.6713277101516724),
 ('princess', 0.5432624220848083),
 ('throne', 0.5386105179786682),
 ('monarch', 0.5347574949264526),
 ('daughter', 0.4980250597000122),
 ('mother', 0.4956442713737488),
 ('elizabeth', 0.4832652509212494),
 ('kingdom', 0.47747090458869934),
 ('prince', 0.4668240249156952),
 ('wife', 0.46473270654678345)]
```

Here are a few other interesting analogies:

In [13]:
```python
# car - drive + fly
glove.most_similar(positive=["car", "fly"], negative=["drive"])
```

Out[13]:
```
[('airplane', 0.5897148251533508),
 ('flying', 0.5675230026245117),
 ('plane', 0.5317023396492004),
 ('flies', 0.5172374248504639),
 ('flown', 0.514790415763855),
 ('airplanes', 0.5091356635093689),
 ('flew', 0.5011662244796753),
 ('planes', 0.4970923364162445),
 ('aircraft', 0.4957723915576935),
 ('helicopter', 0.45859551429748535)]
```

In [14]:
```
# berlin - germany + australia
glove.most_similar(positive=["berlin", "australia"], negative=["germany"])
```

Out[14]:
```
[('sydney', 0.6780862212181091),
 ('melbourne', 0.6499180793762207),
 ('australian', 0.594883143901825),
 ('perth', 0.5828552842140198),
 ('canberra', 0.5610732436180115),
 ('brisbane', 0.55231112241745),
 ('zealand', 0.5240115523338318),
 ('queensland', 0.5193883180618286),
 ('adelaide', 0.5027671456336975),
 ('london', 0.4644604027271271)]
```

In [15]:
```
# england - london + baghdad
glove.most_similar(positive=["england", "baghdad"], negative=["london"])
```

Out[15]:
```
[('iraq', 0.5320571660995483),
 ('fallujah', 0.4834090769290924),
 ('iraqi', 0.47287362813949585),
 ('mosul', 0.464663565158844),
 ('iraqis', 0.43555372953414917),
 ('najaf', 0.4352763295173645),
 ('baqouba', 0.42063191533088684),
 ('basra', 0.4190516471862793),
 ('samarra', 0.4125366508960724),
 ('saddam', 0.40791556239128113)]
```

In [16]:
```
# japan - yen + peso
glove.most_similar(positive=["japan", "peso"], negative=["yen"])
```

Out[16]:
```
[('mexico', 0.5726831555366516),
 ('philippines', 0.5445368885993958),
 ('peru', 0.48382261395454407),
 ('venezuela', 0.4816672205924988),
 ('brazil', 0.46643102169036865),
 ('argentina', 0.45490509271621704),
 ('philippine', 0.4417841136455536),
 ('chile', 0.43960973620414734),
 ('colombia', 0.4386259913444519),
 ('thailand', 0.43396785855293274)]
```

In [27]:
```
# best - good + tall
glove.most_similar(positive=["best", "tall"], negative=["good"])
```

Out[27]:
```
[('tallest', 0.5077418684959412),
 ('taller', 0.47616493701934814),
 ('height', 0.46000057458877563),
 ('metres', 0.4584785997867584),
 ('cm', 0.45212721824645996),
 ('meters', 0.44067248702049255),
 ('towering', 0.42784252762794495),
 ('centimeters', 0.4234543442726135),
 ('inches', 0.4174586832523346),
 ('erect', 0.4087314009666443)]
```

Looking under the hood

Now that we are more familiar with the `most_similar()` method, it is time to implement its functionality ourselves. But first, we need to take a look at the different parts of the `KeyedVectors` object that we will need. Obviously, we will need the vectors themselves. They are stored in the `vectors` attribute.

In [28]:
```
glove.vectors.shape
```

Out[28]:
```
(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()` method.

In [29]:
```
normed_vectors = glove.get_normed_vectors()
normed_vectors.shape
```

Out[29]:
```
(400000, 300)
```

Now we need to map the words in the vocabulary to rows in the `vectors` matrix, and vice versa. The `KeyedVectors` 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.

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 [34]:
```
import numpy as np

def most_similar_words(word, vectors, index_to_key, key_to_index, topn=10):
```

```
    word_idx = key_to_index[word]
    word_vec = vectors[word_idx]

    # Normalizar el vector de la palabra
    word_vec /= np.linalg.norm(word_vec)

    # Calcular las similitudes coseno
    similarities = vectors @ word_vec

    # Excluir la propia palabra
    similarities[word_idx] = -np.inf

    # Obtener los índices de las palabras más similares
    top_indices = np.argpartition(-similarities, range(topn))[:topn]
    top_indices = top_indices[np.argsort(-similarities[top_indices])]

    # Obtener las palabras y sus similitudes
    top_words = [(index_to_key[idx], similarities[idx]) for idx in top_indic

    return top_words
```

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

In [35]:
```
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[35]:
```
[('cacti', 0.6634565),
 ('saguaro', 0.6195854),
 ('pear', 0.5233487),
 ('cactuses', 0.5178282),
 ('prickly', 0.51563185),
 ('mesquite', 0.48448554),
 ('opuntia', 0.45400843),
 ('shrubs', 0.45362067),
 ('peyote', 0.4534496),
 ('succulents', 0.45127875)]
```

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()` method we discussed above.

In [36]:
```
from numpy.linalg import norm
import numpy as np

def analogy(positive, negative, vectors, index_to_key, key_to_index, topn=10
    # Obtener los índices de las palabras positivas y negativas
    pos_indices = [key_to_index[word] for word in positive]
    neg_indices = [key_to_index[word] for word in negative]
```

```
    given_indices = set(pos_indices + neg_indices)

    # Calcular el vector de consulta
    pos_vector = vectors[pos_indices].sum(axis=0)
    neg_vector = vectors[neg_indices].sum(axis=0)
    query_vector = pos_vector - neg_vector
    query_vector /= np.linalg.norm(query_vector)

    # Calcular similitudes
    similarities = vectors @ query_vector

    # Excluir las palabras proporcionadas
    similarities[list(given_indices)] = -np.inf

    # Obtener los índices de las palabras más similares
    top_indices = np.argpartition(-similarities, range(topn))[:topn]
    top_indices = top_indices[np.argsort(-similarities[top_indices])]

    # Obtener las palabras y sus similitudes
    top_words = [(index_to_key[idx], similarities[idx]) for idx in top_indic

    return top_words
```

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

In [37]:
```
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[37]:
```
[('queen', 0.67132765),
 ('princess', 0.5432624),
 ('throne', 0.53861046),
 ('monarch', 0.5347575),
 ('daughter', 0.4980251),
 ('mother', 0.49564427),
 ('elizabeth', 0.48326522),
 ('kingdom', 0.47747084),
 ('prince', 0.466824),
 ('wife', 0.46473268)]
```