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 [2]:
```
from gensim.models import KeyedVectors

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

Out[2]:
```
(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 [3]:
```
# common noun
glove.most_similar("cactus")
```

Out[3]:
```
[('cacti', 0.663456380367279),
 ('saguaro', 0.6195855140686035),
 ('pear', 0.5233485698699951),
 ('cactuses', 0.5178281664848328),
 ('prickly', 0.515631914138794),
 ('mesquite', 0.48448556661605835),
 ('opuntia', 0.4540084898471832),
 ('shrubs', 0.45362064242362976),
 ('peyote', 0.45344963669776917),
 ('succulents', 0.4512787461280823)]
```

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

Out[4]:
```
[('cakes', 0.7506030201911926),
 ('chocolate', 0.6965583562850952),
 ('dessert', 0.6440261006355286),
 ('pie', 0.608742892742157),
 ('cookies', 0.6082394123077393),
 ('frosting', 0.601721465587616),
 ('bread', 0.5954801440238953),
 ('cookie', 0.593381941318512),
 ('recipe', 0.5827102661132812),
 ('baked', 0.5819962620735168)]
```

In [5]:
```
# adjective
glove.most_similar("angry")
```

Out[5]:
```
[('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', 0.5704443454742432)]
```

In [6]:
```
# adverb
glove.most_similar("quickly")
```

Out[6]:
```
[('soon', 0.766185998916626),
 ('rapidly', 0.7216640114784241),
 ('swiftly', 0.7197349667549133),
 ('eventually', 0.7043026685714722),
 ('finally', 0.6900882124900818),
 ('immediately', 0.6842609643936157),
 ('then', 0.6697486042976379),
 ('slowly', 0.6645645499229431),
 ('gradually', 0.6401675939559937),
 ('when', 0.6347666382789612)]
```

In [7]:
```
# preposition
glove.most_similar("between")
```

Out[7]:
```
[('sides', 0.5867610573768616),
 ('both', 0.5843431949615479),
 ('two', 0.5652360916137695),
 ('differences', 0.514071524143219),
 ('which', 0.5120179057121277),
 ('conflict', 0.5115456581115723),
 ('relationship', 0.5022751092910767),
 ('and', 0.498425155878067),
 ('in', 0.4970666766166687),
 ('relations', 0.4970114529132843)]
```

In [8]:
```
# determiner
glove.most_similar("the")
```

Out[8]:
```
[('of', 0.7057957649230957),
 ('which', 0.6992015838623047),
 ('this', 0.6747026443481445),
 ('part', 0.6727458238601685),
 ('same', 0.6592389345169067),
 ('its', 0.6446539759635925),
 ('first', 0.6398990750312805),
 ('in', 0.6361348032951355),
 ('one', 0.6245334148406982),
 ('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 [9]:
```
# king - man + woman
glove.most_similar(positive=["king", "woman"], negative=["man"])
```

Out[9]:
```
[('queen', 0.6713277101516724),
 ('princess', 0.5432624816894531),
 ('throne', 0.5386103987693787),
 ('monarch', 0.5347574949264526),
 ('daughter', 0.49802514910697937),
 ('mother', 0.49564430117607117),
 ('elizabeth', 0.4832652509212494),
 ('kingdom', 0.47747090458869934),
 ('prince', 0.4668239951133728),
 ('wife', 0.46473270654678345)]
```

Here are a few other interesting analogies:

In [10]:
```
# car - drive + fly
glove.most_similar(positive=["car", "fly"], negative=["drive"])
```

Out[10]:
```
[('airplane', 0.5897148251533508),
 ('flying', 0.5675230026245117),
 ('plane', 0.5317023992538452),
 ('flies', 0.5172374248504639),
 ('flown', 0.514790415763855),
 ('airplanes', 0.5091356635093689),
 ('flew', 0.5011662244796753),
 ('planes', 0.4970923364162445),
 ('aircraft', 0.4957723915576935),
 ('helicopter', 0.45859551429748535)]
```

In [11]:
```python
# berlin - germany + australia
glove.most_similar(positive=["berlin", "australia"], negative=["germany"])
```

Out[11]:
```
[('sydney', 0.6780862212181091),
 ('melbourne', 0.6499180793762207),
 ('australian', 0.594883143901825),
 ('perth', 0.5828553438186646),
 ('canberra', 0.5610732436180115),
 ('brisbane', 0.5523110628128052),
 ('zealand', 0.5240115523338318),
 ('queensland', 0.5193883180618286),
 ('adelaide', 0.5027671456336975),
 ('london', 0.4644604027271271)]
```

In [12]:
```python
# england - london + baghdad
glove.most_similar(positive=["england", "baghdad"], negative=["london"])
```

Out[12]:
```
[('iraq', 0.5320571660995483),
 ('fallujah', 0.4834090769290924),
 ('iraqi', 0.47287362813949585),
 ('mosul', 0.464663565158844),
 ('iraqis', 0.43555372953414917),
 ('najaf', 0.4352763295173645),
 ('baqouba', 0.42063194513320923),
 ('basra', 0.41905173659324646),
 ('samarra', 0.4125366508960724),
 ('saddam', 0.40791556239128113)]
```

In [13]:
```python
# japan - yen + peso
glove.most_similar(positive=["japan", "peso"], negative=["yen"])
```

Out[13]:
```
[('mexico', 0.5726832151412964),
 ('philippines', 0.5445368885993958),
 ('peru', 0.48382261395454407),
 ('venezuela', 0.4816672205924988),
 ('brazil', 0.4664309620857239),
 ('argentina', 0.45490506291389465),
 ('philippine', 0.4417841136455536),
 ('chile', 0.43960973620414734),
 ('colombia', 0.4386259913444519),
 ('thailand', 0.43396785855293274)]
```

In [14]:
```python
# best - good + tall
glove.most_similar(positive=["best", "tall"], negative=["good"])
```

Out[14]:
```
[('tallest', 0.5077419281005859),
 ('taller', 0.47616496682167053),
 ('height', 0.46000057458877563),
 ('metres', 0.4584786891937256),
 ('cm', 0.45212721824645996),
 ('meters', 0.44067248702049255),
 ('towering', 0.42784252762794495),
 ('centimeters', 0.4234543442726135),
 ('inches', 0.41745859384536743),
 ('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 [15]:
```
glove.vectors.shape
```

Out[15]:
```
(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 [16]:
```
normed_vectors = glove.get_normed_vectors()
normed_vectors.shape
```

Out[16]:
```
(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.

In [17]:
```
#glove.index_to_key
vocab_list = glove.index_to_key
```

In [18]:
```
#glove.key_to_index
vocab_dict = glove.key_to_index
word = 'king'
index = vocab_dict[word]
vector = glove.vectors[index]
print(f"Vector for '{word}':\n", vector)
word_at_index_0 = vocab_list[0]
print(f"Word at index 0: {word_at_index_0}")
```

```
Vector for 'king':
 [ 0.0033901 -0.34614    0.28144    0.48382    0.59469    0.012965
  0.53982    0.48233    0.21463   -1.0249    -0.34788   -0.79001
 -0.15084    0.61374    0.042811   0.19323    0.25462    0.32528
  0.05698    0.063253  -0.49439    0.47337   -0.16761    0.045594
  0.30451   -0.35416   -0.34583   -0.20118    0.25511    0.091111
  0.014651  -0.017541  -0.23854    0.48215   -0.9145    -0.36235
  0.34736    0.028639  -0.027065  -0.036481  -0.067391  -0.23452
 -0.13772    0.33951    0.13415   -0.1342     0.47856   -0.1842
  0.10705   -0.45834   -0.36085   -0.22595    0.32881   -0.13643
  0.23128    0.34269    0.42344    0.47057    0.479      0.074639
  0.3344     0.10714   -0.13289    0.58734    0.38616   -0.52238
 -0.22028   -0.072322   0.32269    0.44226   -0.037382   0.18324
  0.058082   0.26938    0.36202    0.13983    0.016815  -0.34426
  0.4827     0.2108     0.75618   -0.13092   -0.025741   0.43391
  0.33893   -0.16438    0.26817    0.68774    0.311     -0.2509
  0.0027749 -0.39809   -0.43399    0.049531  -0.42686   -0.094679
  0.56925    0.28742   -0.015721  -0.059162   0.1912    -0.59814
  0.65486   -0.31363    0.16881    0.10862    0.075316   0.34093
 -0.14706    0.8359     0.39697    0.52358   -0.0096367 -0.14406
  0.37783   -0.596     -0.063192  -0.85297   -0.3098    -1.0587
 -1.025      0.4508    -0.73324   -1.2461    -0.028488   0.20299
  0.00259    0.31995    0.35744    0.28533    0.228      0.50956
 -0.35942    0.32683    0.046264  -0.86896   -0.2707    -0.15454
 -0.32152    0.31121    0.44134    0.85189    0.21065   -0.13741
 -0.15359   -0.059722   0.027375   0.23724   -0.39197   -0.66065
  0.23587    0.032384  -0.64043    0.55004    0.29597    0.14989
  0.46079   -0.26561   -0.1607    -0.36328    1.0782     0.31375
  0.1149     0.20248    0.032748   0.41082   -0.082536   0.36606
  0.18771    0.75415    0.079648   0.24181   -0.60319   -0.37296
 -0.047767   0.45008   -0.21135    0.022251  -0.084325   0.18644
 -0.14682    0.56571   -0.30995    0.17423   -0.41122   -0.84772
 -0.71114    0.69895   -0.13008   -0.34195   -0.30501   -0.12646
  0.29957   -0.43488    0.31935    0.2817    -0.20631   -0.48877
  0.34477    0.03907    1.6198    -0.6352    -0.0037675 -0.41271
  0.30704   -0.50486    0.036385  -0.046386  -0.12004    0.010029
 -0.49116    0.041486   0.002979  -0.57694   -0.42088   -0.063218
  0.0034244 -0.25093   -0.39689   -0.36984    0.32689    0.01385
  0.23634   -0.055199  -0.58453    0.13211    0.50943    0.25198
 -0.0088309 -0.21273   -0.48423    0.5234    -0.32832   -0.013821
  0.15812    0.46696    0.036822  -0.090878   0.18854    0.20794
 -0.42682    0.59705    0.53109    0.19185   -0.16392    0.064956
 -0.36009   -0.59882   -0.28134    0.1017     0.02601    0.44298
 -0.31922   -0.22432    0.7828     0.041307   0.1742     0.27777
  0.43792   -0.84324    0.27012   -0.21547    0.52408   -0.19426
 -0.21878   -0.20713    0.092994  -0.15804    0.28716   -0.11911
 -0.20688   -0.36482    0.68548   -0.10394   -0.49974   -0.47038
 -1.2953    -0.46236    0.44467    0.13337    0.88762   -0.26494
  0.080676  -0.20625   -0.51232    0.31112    0.062035   0.30302
 -0.33344   -0.20924   -0.17348   -0.43434   -0.45743   -0.077803
 -0.33248   -0.078633   0.82182    0.082088  -0.68795    0.30266  ]
Word at index 0: the
```

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 [19]:
```
import numpy as np

def most_similar_words(word, vectors, index_to_key, key_to_index, topn=10):
    if word not in key_to_index:
        raise ValueError(f"'{word}' not found in vocabulary.")

    word_id = key_to_index[word]

    word_vector = vectors[word_id]

    word_norm = np.linalg.norm(word_vector)
    similarities = (vectors @ word_vector) / (np.linalg.norm(vectors, axis=1

    similar_word_ids = np.argsort(similarities)[::-1]

    mask = similar_word_ids != word_id
    similar_word_ids = similar_word_ids[mask]

    top_word_ids = similar_word_ids[:topn]
    top_words = [(index_to_key[i], similarities[i]) for i in top_word_ids]

    return top_words
```

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

In [20]:
```
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[20]:
```
[('cacti', 0.66345644),
 ('saguaro', 0.61958545),
 ('pear', 0.5233487),
 ('cactuses', 0.5178283),
 ('prickly', 0.5156319),
 ('mesquite', 0.4844855),
 ('opuntia', 0.45400846),
 ('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 [21]:
```
from numpy.linalg import norm

def analogy(positive, negative, vectors, index_to_key, key_to_index, topn=10
    pos_ids = [key_to_index[word] for word in positive if word in key_to_inc
    neg_ids = [key_to_index[word] for word in negative if word in key_to_inc
    given_word_ids = pos_ids + neg_ids

    if not pos_ids or not neg_ids:
        raise ValueError("One or more words are not in the vocabulary.")

    pos_emb = np.sum(vectors[pos_ids], axis=0)
    neg_emb = np.sum(vectors[neg_ids], axis=0)

    emb = pos_emb - neg_emb

    emb = emb / norm(emb)

    similarities = vectors @ emb / (np.linalg.norm(vectors, axis=1) * norm(e

    ids_ascending = np.argsort(similarities)
    ids_descending = ids_ascending[::-1]

    given_words_mask = np.isin(ids_descending, given_word_ids, invert=True)
    ids_descending = ids_descending[given_words_mask]

    top_ids = ids_descending[:topn]

    top_words = [(index_to_key[i], similarities[i]) for i in top_ids]

    return top_words
```

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

In [22]:
```
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[22]:
```
[('queen', 0.67132777),
 ('princess', 0.54326254),
 ('throne', 0.5386106),
 ('monarch', 0.5347576),
 ('daughter', 0.4980252),
 ('mother', 0.49564433),
 ('elizabeth', 0.48326525),
 ('kingdom', 0.47747087),
 ('prince', 0.466824),
 ('wife', 0.4647328)]
```

In [23]:
```
!jupyter nbconvert --to html 'embeddings.ipynb'
```

```
[NbConvertApp] Converting notebook embeddings.ipynb to html
/Users/marcelo/Library/Python/3.10/lib/python/site-packages/nbformat/__init_
_.py:93: MissingIDFieldWarning: Code cell is missing an id field, this will
become a hard error in future nbformat versions. You may want to use `normal
ize()` on your notebooks before validations (available since nbformat 5.1.
4). Previous versions of nbformat are fixing this issue transparently, and w
ill stop doing so in the future.
  validate(nb)
[NbConvertApp] Writing 325012 bytes to embeddings.html
```