Actividad_Embeddings_Preentrenados

October 18, 2024

Realizado por: Rodolfo Jesús Cruz Rebollar

Matrícula: A01368326

Grupo: 101

1 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!

[1]:
```python
from gensim.models import KeyedVectors

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

[1]:
```
(400000, 300)
```

1.1 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.

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

[2]:
```
[('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)]
```

[3]:
```python
# common noun
glove.most_similar("cake")
```

[3]:
```
[('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)]
```

[4]:
```python
# adjective
glove.most_similar("angry")
```

[4]:
```
[('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)]
```

[5]:
```python
# adverb
glove.most_similar("quickly")
```

[5]:
```
[('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)]
```

[6]:
```
# preposition
glove.most_similar("between")
```

[6]:
```
[('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)]
```

[7]:
```
# determiner
glove.most_similar("the")
```

[7]:
```
[('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)]
```

1.2 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:

[8]:
```
# king - man + woman
glove.most_similar(positive=["king", "woman"], negative=["man"])
```

[8]:
```
[('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:

[9]:
```
# car - drive + fly
glove.most_similar(positive=["car", "fly"], negative=["drive"])
```

[9]:
```
[('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)]
```

[10]:
```
# berlin - germany + australia
glove.most_similar(positive=["berlin", "australia"], negative=["germany"])
```

[10]:
```
[('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)]
```

[11]:
```
# england - london + baghdad
glove.most_similar(positive=["england", "baghdad"], negative=["london"])
```

[11]:
```
[('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)]
```

[12]:
```
# japan - yen + peso
glove.most_similar(positive=["japan", "peso"], negative=["yen"])
```

[12]:
```
[('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)]
```

[13]:
```
# best - good + tall
glove.most_similar(positive=["best", "tall"], negative=["good"])
```

[13]:
```
[('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)]
```

1.3 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.

[14]:
```
glove.vectors.shape
```

[14]:
```
(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.

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

[15]:
```
(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.

[16]:
```python
# Mostrar los primeros 15 elementos de la lista para no abarcar muchas
# páginas del documento final

glove.index_to_key[:15]
```

[16]:
```
['the',
 ',',
 '.',
 'of',
 'to',
 'and',
 'in',
 'a',
 '"',
 "'s",
 'for',
 '-',
 'that',
 'on',
 'is']
```

[17]:
```python
# Mostrar sólo los primeros 15 elementos del diccionario para no abarcar
# muchas páginas del documento final

list(glove.key_to_index.items())[:15]
```

[17]:
```
[('the', 0),
 (',', 1),
 ('.', 2),
 ('of', 3),
 ('to', 4),
 ('and', 5),
 ('in', 6),
 ('a', 7),
 ('"', 8),
 ("'s", 9),
 ('for', 10),
 ('-', 11),
 ('that', 12),
```

```
 ('on', 13),
 ('is', 14)]
```

1.4 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.

[18]:
```python
import numpy as np

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

    # retrieve word_id corresponding to given word

    id_palabra = key_to_index[word]

    # En caso de que la palabra sea nula, desplegar un mensaje al usuario de␣
 ↪que no existe la palabra

    if id_palabra == None:

        print(f'{word} does not exist in the vocabulary...')

    else:

        # retrieve embedding for given word

        w_emb = vectors[id_palabra]

        # calculate similarities to all words in out vocabulary (hint: use @)

        sim_palabras = vectors @ w_emb

        # get word_ids in ascending order with respect to similarity score

        Id_palabras_ord = np.argsort(sim_palabras)

        # reverse word_ids (esto se hace para obtener los ids de las palabras,␣
 ↪pero esta vez en orden descendente)

        id_palabras_desc = Id_palabras_ord[::-1]

        # get boolean array with element corresponding to word_id set to false

        bool_values_palabras = np.ones(len(index_to_key), dtype = bool)
```

```
        # Ahora se asigna el valor False al Id de la palabra en cuestión

        bool_values_palabras[id_palabra] = False

        # obtain new array of indices that doesn't contain word_id

        # Actualizar el array de ids en orden descendente para que ya no tenga␣
↪al word id en cuestión

        id_palabras_desc =␣
↪id_palabras_desc[bool_values_palabras[id_palabras_desc]]

        # (otherwise the most similar word to the argument would be the␣
↪argument itself)

        '''Si no se encuentran palabras semejantes a word, entonces la palabra␣
↪con mayor similitud a word
        será ella misma'''

        if len(id_palabras_desc) == 0:

            # Asignar la palabra word misma a la lista de palabras más␣
↪semejantes a ella

            topN_ids_palabras = word

        else:

            # get topn word_ids

            topN_ids_palabras = id_palabras_desc[range(0, topn)]

        # retrieve topn words with their corresponding similarity score

        puntuaciones_topN_palabras = list((index_to_key[k], sim_palabras[k])␣
↪for k in topN_ids_palabras)

        # return results

        return puntuaciones_topN_palabras
```

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

[19]:
```
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)
```

[19]: [('cacti', 0.66345644),
 ('saguaro', 0.6195854),
 ('pear', 0.5233487),
 ('cactuses', 0.5178282),
 ('prickly', 0.51563185),
 ('mesquite', 0.4844855),
 ('opuntia', 0.45400843),
 ('shrubs', 0.45362067),
 ('peyote', 0.4534496),
 ('succulents', 0.45127875)]

1.5 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.

[20]:
```
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 = list(key_to_index[pal_positiva] for pal_positiva in positive)

    neg_ids = list(key_to_index[pal_negativa] for pal_negativa in negative)

    # Calcular el total de palabras tanto positivas como negativas

    given_word_ids = pos_ids + neg_ids

    # get embeddings for positive and negative words

    pos_emb = sum(vectors[j] for j in pos_ids)

    neg_emb = sum(vectors[h] for h in neg_ids)

    # get embedding for analogy

    emb = pos_emb - neg_emb

    # normalize embedding

    emb /= norm(emb)

    # calculate similarities to all words in out vocabulary
```

```
    similarities = vectors @ emb

    # get word_ids in ascending order with respect to similarity score

    ids_ascending = np.argsort(similarities)

    # reverse word_ids para obtener los ids de palabras en orden descendente

    ids_descending = ids_ascending[::-1]

    # get boolean array with element corresponding to any of given_word_ids set␣
↪to false
    ### Hint: You can use np.isni

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

    # 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[range(0, topn)]

    # retrieve topn words with their corresponding similarity score

    top_words = list((index_to_key[f], similarities[f]) for f 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.

[21]:
```
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)
```

[21]:
```
[('queen', 0.67132777),
 ('princess', 0.5432625),
 ('throne', 0.5386105),
 ('monarch', 0.53475755),
 ('daughter', 0.49802518),
```

```
('mother', 0.49564433),
('elizabeth', 0.48326525),
('kingdom', 0.47747087),
('prince', 0.466824),
('wife', 0.4647328)]
```