Manipulating word embeddings

In this week's assignment, you are going to use a pre-trained word embedding for finding word analogies and equivalence. This exercise can be used as an Intrinsic Evaluation for the word embedding performance. In this notebook, you will apply linear algebra operations using NumPy to find analogies between words manually. This will help you to prepare for this week's assignment.

```python
import pandas as pd # Library for Dataframes
import numpy as np # Library for math functions
import pickle # Python object serialization library. Not secure

word_embeddings = pickle.load( open( "word_embeddings_subset.p", "rb" ) )
len(word_embeddings) # there should be 243 words that will be used in this assign
```

Out[1]: 243

Now that the model is loaded, we can take a look at the word representations. First, note that the *word*embeddings_ is a dictionary. Each word is the key to the entry, and the value is its corresponding vector presentation. Remember that square brackets allow access to any entry if the key exists.

In [2]:
```python
countryVector = word_embeddings['country'] # Get the vector representation for tl
print(type(countryVector)) # Print the type of the vector. Note it is a numpy ar|
print(countryVector) # Print the values of the vector.
```

```
<class 'numpy.ndarray'>
[-0.08007812  0.13378906  0.14355469  0.09472656 -0.04736328 -0.02355957
 -0.00854492 -0.18652344  0.04589844 -0.08154297 -0.03442383 -0.11621094
  0.21777344 -0.10351562 -0.06689453  0.15332031 -0.19335938  0.26367188
 -0.13671875 -0.05566406  0.07470703 -0.00070953  0.09375    -0.14453125
  0.04296875 -0.01916504 -0.22558594 -0.12695312 -0.0168457   0.05224609
  0.0625     -0.1484375  -0.01965332  0.17578125  0.10644531 -0.04760742
 -0.10253906 -0.28515625  0.10351562  0.20800781 -0.07617188 -0.04345703
  0.08642578  0.08740234  0.11767578  0.20996094 -0.07275391  0.1640625
 -0.01135254  0.0025177   0.05810547 -0.03222656  0.06884766  0.046875
  0.10107422  0.02148438 -0.16210938  0.07128906 -0.16210938  0.05981445
  0.05102539 -0.05566406  0.06787109 -0.03759766  0.04345703 -0.03173828
 -0.03417969 -0.01116943  0.06201172 -0.08007812 -0.14941406  0.11914062
  0.02575684  0.00302124  0.04711914 -0.17773438  0.04101562  0.05541992
  0.00598145  0.03027344 -0.07666016 -0.109375    0.02832031 -0.10498047
  0.0100708  -0.03149414 -0.22363281 -0.03125    -0.01147461  0.17285156
  0.08056641 -0.10888672 -0.09570312 -0.21777344 -0.07910156 -0.10009766
  0.06396484 -0.11962891  0.18652344 -0.02062988 -0.02172852  0.29296875
 -0.00793457  0.0324707  -0.15136719  0.00227356 -0.03540039 -0.13378906
  0.0546875  -0.03271484 -0.01855469 -0.10302734 -0.13378906  0.11425781
  0.16699219  0.01361084 -0.02722168 -0.2109375   0.07177734  0.08691406
 -0.09960938  0.01422119 -0.18261719  0.00741577  0.01965332  0.00738525
 -0.03271484 -0.15234375 -0.26367188 -0.14746094  0.03320312 -0.03344727
 -0.01000977  0.01855469  0.00183868 -0.10498047  0.09667969  0.07910156
  0.11181641  0.13085938 -0.08740234 -0.1328125   0.05004883  0.19824219
  0.0612793   0.16210938  0.06933594  0.01281738  0.01550293  0.01531982
  0.11474609  0.02758789  0.13769531 -0.08349609  0.01123047 -0.20507812
 -0.12988281 -0.16699219  0.20410156 -0.03588867 -0.10888672  0.0534668
  0.15820312 -0.20410156  0.14648438 -0.11572266  0.01855469 -0.13574219
  0.24121094  0.12304688 -0.14550781  0.17578125  0.11816406 -0.30859375
  0.10888672 -0.22363281  0.19335938 -0.15722656 -0.07666016 -0.09082031
 -0.19628906 -0.23144531 -0.09130859 -0.14160156  0.06347656  0.03344727
 -0.03369141  0.06591797  0.06201172  0.3046875   0.16796875 -0.11035156
 -0.03833008 -0.02563477 -0.09765625  0.04467773 -0.0534668   0.11621094
 -0.15039062 -0.16308594 -0.15527344  0.04638672  0.11572266 -0.06640625
 -0.04516602  0.02331543 -0.08105469 -0.0255127  -0.07714844  0.0016861
  0.15820312  0.00994873 -0.06445312  0.15722656 -0.03112793  0.10644531
 -0.140625    0.23535156 -0.11279297  0.16015625  0.00061798 -0.1484375
  0.02307129 -0.109375    0.05444336 -0.14160156  0.11621094  0.03710938
  0.14746094 -0.04199219 -0.01391602 -0.03881836  0.02783203  0.10205078
  0.07470703  0.20898438 -0.04223633 -0.04150391 -0.00588989 -0.14941406
 -0.04296875 -0.10107422 -0.06176758  0.09472656  0.22265625 -0.02307129
  0.04858398 -0.15527344 -0.02282715 -0.04174805  0.16699219 -0.09423828
  0.14453125  0.11132812  0.04223633 -0.16699219  0.10253906  0.16796875
  0.12597656 -0.11865234 -0.0213623  -0.08056641  0.24316406  0.15527344
  0.16503906  0.00854492 -0.12255859  0.08691406 -0.11914062 -0.02941895
  0.08349609 -0.03100586  0.13964844 -0.05151367  0.00765991 -0.04443359
 -0.04980469 -0.03222656 -0.00952148 -0.10888672 -0.10302734 -0.15722656
  0.19335938  0.04858398  0.015625   -0.08105469 -0.11621094 -0.01989746
  0.05737305  0.06103516 -0.14550781  0.06738281 -0.24414062 -0.07714844
  0.04760742 -0.07519531 -0.14941406 -0.04418945  0.09716797  0.06738281]
```

It is important to note that we store each vector as a NumPy array. It allows us to use the linear algebra operations on it.

The vectors have a size of 300, while the vocabulary size of Google News is around 3 million words!

```
In [3]:  #Get the vector for a given word:
         def vec(w):
             return word_embeddings[w]
```

Operating on word embeddings

Remember that understanding the data is one of the most critical steps in Data Science. Word embeddings are the result of machine learning processes and will be part of the input for further processes. These word embedding needs to be validated or at least understood because the performance of the derived model will strongly depend on its quality.

Word embeddings are multidimensional arrays, usually with hundreds of attributes that pose a challenge for its interpretation.

In this notebook, we will visually inspect the word embedding of some words using a pair of attributes. Raw attributes are not the best option for the creation of such charts but will allow us to illustrate the mechanical part in Python.

In the next cell, we make a beautiful plot for the word embeddings of some words. Even if plotting the dots gives an idea of the words, the arrow representations help to visualize the vector's alignment as well.

```
In [4]: import matplotlib.pyplot as plt # Import matplotlib

        words = ['oil', 'gas', 'happy', 'sad', 'city', 'town', 'village', 'country', 'co

        bag2d = np.array([vec(word) for word in words]) # Convert each word to its vecto

        fig, ax = plt.subplots(figsize = (10, 10)) # Create custom size image

        col1 = 3 # Select the column for the x axis
        col2 = 2 # Select the column for the y axis

        # Print an arrow for each word
        for word in bag2d:
            ax.arrow(0, 0, word[col1], word[col2], head_width=0.005, head_length=0.005,


        ax.scatter(bag2d[:, col1], bag2d[:, col2]); # Plot a dot for each word

        # Add the word label over each dot in the scatter plot
        for i in range(0, len(words)):
            ax.annotate(words[i], (bag2d[i, col1], bag2d[i, col2]))


        plt.show()
```

`<Figure size 1000x1000 with 1 Axes>`

Note that similar words like 'village' and 'town' or 'petroleum', 'oil', and 'gas' tend to point in the same direction. Also, note that 'sad' and 'happy' looks close to each other; however, the vectors point in opposite directions.

In this chart, one can figure out the angles and distances between the words. Some words are close in both kinds of distance metrics.

Word distance

Now plot the words 'sad', 'happy', 'town', and 'village'. In this same chart, display the vector from 'village' to 'town' and the vector from 'sad' to 'happy'. Let us use NumPy for these linear algebra operations.

```
In [5]: words = ['sad', 'happy', 'town', 'village']

        bag2d = np.array([vec(word) for word in words]) # Convert each word to its vector

        fig, ax = plt.subplots(figsize = (10, 10)) # Create custom size image

        col1 = 3 # Select the column for the x axe
        col2 = 2 # Select the column for the y axe

        # Print an arrow for each word
        for word in bag2d:
            ax.arrow(0, 0, word[col1], word[col2], head_width=0.0005, head_length=0.0005

        # print the vector difference between village and town
        village = vec('village')
        town = vec('town')
        diff = town - village
        ax.arrow(village[col1], village[col2], diff[col1], diff[col2], fc='b', ec='b', w

        # print the vector difference between village and town
        sad = vec('sad')
        happy = vec('happy')
        diff = happy - sad
        ax.arrow(sad[col1], sad[col2], diff[col1], diff[col2], fc='b', ec='b', width = 1


        ax.scatter(bag2d[:, col1], bag2d[:, col2]); # Plot a dot for each word

        # Add the word label over each dot in the scatter plot
        for i in range(0, len(words)):
            ax.annotate(words[i], (bag2d[i, col1], bag2d[i, col2]))


        plt.show()
```
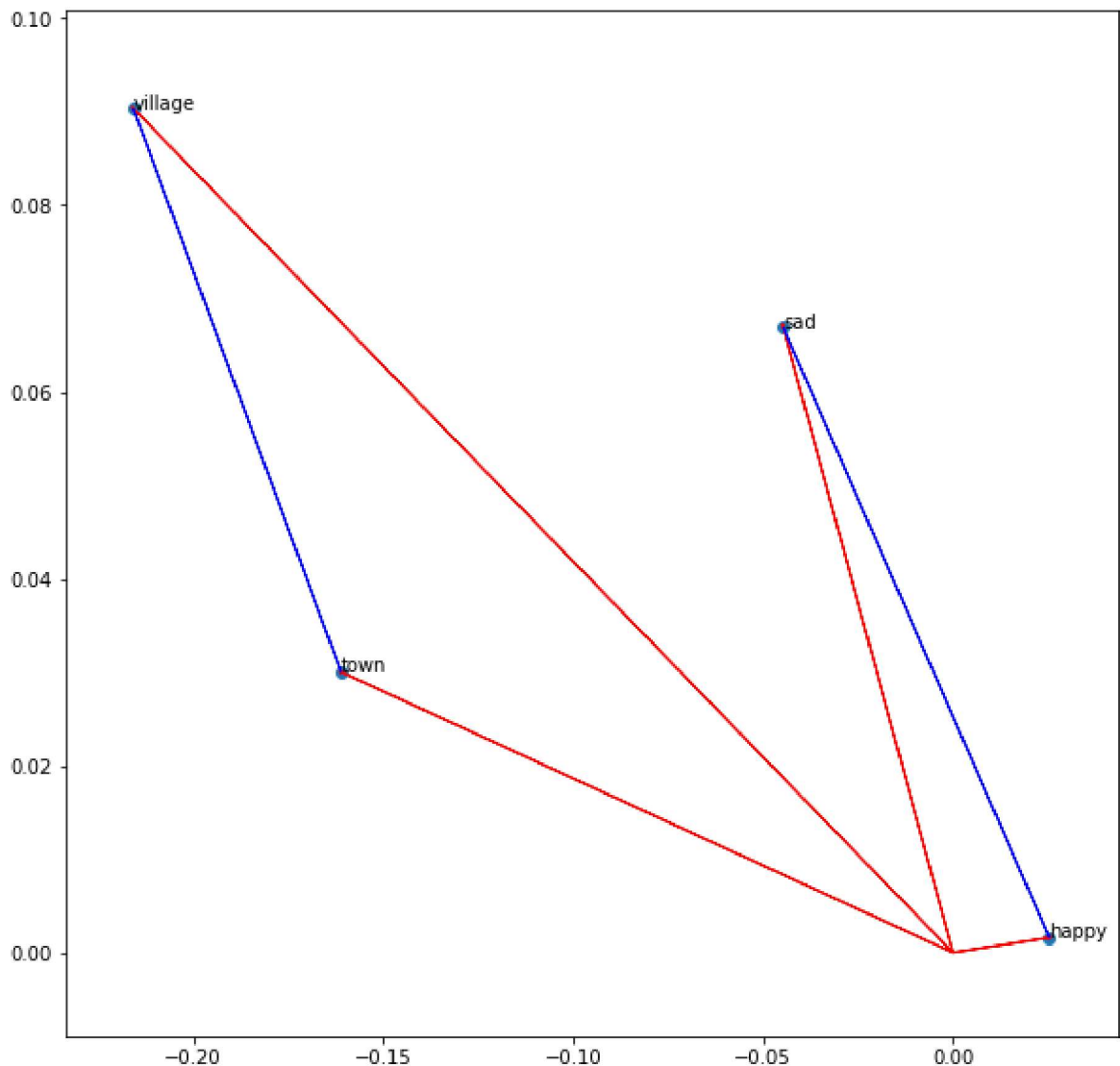
Linear algebra on word embeddings

In the lectures, we saw the analogies between words using algebra on word embeddings. Let us see how to do it in Python with Numpy.

To start, get the **norm** of a word in the word embedding.

```
In [6]:  print(np.linalg.norm(vec('town'))) # Print the norm of the word town
         print(np.linalg.norm(vec('sad'))) # Print the norm of the word sad
```

```
2.3858097
2.9004838
```

Predicting capitals

Now, applying vector difference and addition, one can create a vector representation for a new word. For example, we can say that the vector difference between 'France' and 'Paris' represents the concept of Capital.

One can move from the city of Madrid in the direction of the concept of Capital, and obtain something close to the corresponding country to which Madrid is the Capital.

```
In [7]:  capital = vec('France') - vec('Paris')
         country = vec('Madrid') + capital

         print(country[0:5]) # Print the first 5 values of the vector
```

```
[-0.02905273 -0.2475586   0.53952026  0.20581055 -0.14862823]
```

We can observe that the vector 'country' that we expected to be the same as the vector for Spain is not exactly it.

```
In [8]:  diff = country - vec('Spain')
         print(diff[0:10])
```

```
[-0.06054688 -0.06494141  0.37643433  0.08129883 -0.13007355 -0.00952148
 -0.03417969 -0.00708008  0.09790039 -0.01867676]
```

So, we have to look for the closest words in the embedding that matches the candidate country. If the word embedding works as expected, the most similar word must be 'Spain'. Let us define a function that helps us to do it. We will store our word embedding as a DataFrame, which facilitate the lookup operations based on the numerical vectors.

```
In [9]:  # Create a dataframe out of the dictionary embedding. This facilitate the algebr
         keys = word_embeddings.keys()
         data = []
         for key in keys:
             data.append(word_embeddings[key])

         embedding = pd.DataFrame(data=data, index=keys)
         # Define a function to find the closest word to a vector:
         def find_closest_word(v, k = 1):
             # Calculate the vector difference from each word to the input vector
             diff = embedding.values - v
             # Get the norm of each difference vector.
             # It means the squared euclidean distance from each word to the input vector
             delta = np.sum(diff * diff, axis=1)
             # Find the index of the minimun distance in the array
             i = np.argmin(delta)
             # Return the row name for this item
             return embedding.iloc[i].name
```

```
In [10]:   # Print some rows of the embedding as a Dataframe
           embedding.head(10)
```

Out[10]:

	0	1	2	3	4	5	6	7
country	-0.080078	0.133789	0.143555	0.094727	-0.047363	-0.023560	-0.008545	-0.186523	0.
city	-0.010071	0.057373	0.183594	-0.040039	-0.029785	-0.079102	0.071777	0.013306	-0.
China	-0.073242	0.135742	0.108887	0.083008	-0.127930	-0.227539	0.151367	-0.045654	-0.
Iraq	0.191406	0.125000	-0.065430	0.060059	-0.285156	-0.102539	0.117188	-0.351562	-0.
oil	-0.139648	0.062256	-0.279297	0.063965	0.044434	-0.154297	-0.184570	-0.498047	0.
town	0.123535	0.159180	0.030029	-0.161133	0.015625	0.111816	0.039795	-0.196289	-0.
Canada	-0.136719	-0.154297	0.269531	0.273438	0.086914	-0.076172	-0.018677	0.006256	0.
London	-0.267578	0.092773	-0.238281	0.115234	-0.006836	0.221680	-0.251953	-0.055420	0.
England	-0.198242	0.115234	0.062500	-0.058350	0.226562	0.045898	-0.062256	-0.202148	0.
Australia	0.048828	-0.194336	-0.041504	0.084473	-0.114258	-0.208008	-0.164062	-0.269531	0.

10 rows × 300 columns
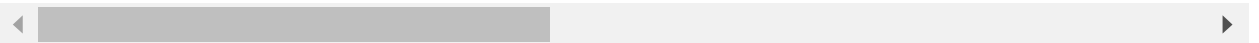
Now let us find the name that corresponds to our numerical country:

```
In [11]:   find_closest_word(country)
```

Out[11]:   'Spain'

Predicting other Countries

```
In [12]:   find_closest_word(vec('Italy') - vec('Rome') + vec('Madrid'))
```

Out[12]:   'Spain'

```
In [13]:   print(find_closest_word(vec('Berlin') + capital))
           print(find_closest_word(vec('Beijing') + capital))
```

```
Germany
China
```

However, it does not always work.

```
In [14]:   print(find_closest_word(vec('Lisbon') + capital))
```

```
Lisbon
```

Represent a sentence as a vector

A whole sentence can be represented as a vector by summing all the word vectors that conform to the sentence. Let us see.

```
In [15]:  doc = "Spain petroleum city king"
          vdoc = [vec(x) for x in doc.split(" ")]
          doc2vec = np.sum(vdoc, axis = 0)
          doc2vec
```

```
Out[15]:  array([ 2.87475586e-02,   1.03759766e-01,   1.32629395e-01,   3.33007812e-01,
                 -2.61230469e-02,  -5.95703125e-01,  -1.25976562e-01,  -1.01306152e+00,
                 -2.18544006e-01,   6.60705566e-01,  -2.58300781e-01,  -2.09960938e-02,
                 -7.71484375e-02,  -3.07128906e-01,  -5.94726562e-01,   2.00561523e-01,
                 -1.04980469e-02,  -1.10748291e-01,   4.82177734e-02,   6.38977051e-01,
                  2.36083984e-01,  -2.69775391e-01,   3.90625000e-02,   4.16503906e-01,
                  2.83416748e-01,  -7.25097656e-02,  -3.12988281e-01,   1.05712891e-01,
                  3.22265625e-02,   2.38403320e-01,   3.88183594e-01,  -7.51953125e-02,
                 -1.26281738e-01,   6.60644531e-01,  -7.89794922e-01,  -7.04345703e-02,
                 -1.14379883e-01,  -4.78515625e-02,   4.76318359e-01,   5.31127930e-01,
                  8.10546875e-02,  -1.17553711e-01,   1.02050781e+00,   5.59814453e-01,
                 -1.17187500e-01,   1.21826172e-01,  -5.51574707e-01,   1.44531250e-01,
                 -7.66113281e-01,   5.36102295e-01,  -2.80029297e-01,   3.85986328e-01,
                 -2.39135742e-01,  -2.86865234e-02,  -5.10498047e-01,   2.59658813e-01,
                 -7.52929688e-01,   4.32128906e-02,  -7.17773438e-02,  -1.26708984e-01,
                  4.40673828e-02,   5.12939453e-01,  -5.15808105e-01,   1.20117188e-01,
                 -5.52978516e-02,  -3.92089844e-01,  -3.15917969e-01,   1.57226562e-01,
                 -3.19702148e-01,   1.75170898e-01,  -3.81835938e-01,  -2.07031250e-01,
                 -4.72717285e-02,  -2.79296875e-01,  -3.29040527e-01,  -1.69067383e-01,
                  1.61132812e-02,   1.71569824e-01,   5.73730469e-02,  -2.44140625e-03,
                  8.34960938e-02,  -1.58203125e-01,  -3.10119629e-01,   5.28564453e-02,
                  8.60595703e-02,   5.12695312e-02,  -7.22900391e-01,   4.97924805e-01,
                 -5.85937500e-03,   4.49951172e-01,   3.82446289e-01,  -2.80029297e-01,
                 -3.28125000e-01,  -6.27441406e-02,  -4.81933594e-01,   1.93176270e-02,
                 -1.69326782e-01,  -4.28649902e-01,   5.39062500e-01,  -1.28417969e-01,
                 -8.83789062e-02,   5.13916016e-01,   9.13085938e-02,  -1.60156250e-01,
                  6.86035156e-02,  -9.74121094e-02,  -3.70712280e-01,  -3.27270508e-01,
                  1.77978516e-01,  -4.65332031e-01,   1.70410156e-01,   9.08203125e-02,
                  2.76857376e-01,  -1.69677734e-01,   3.27728271e-01,  -3.12500000e-02,
                 -2.20809937e-01,  -3.46679688e-01,   4.67407227e-01,   5.31860352e-01,
                 -1.30615234e-01,  -2.36816406e-02,  -6.56250000e-01,  -5.79589844e-01,
                 -2.05810547e-01,  -3.03222656e-01,   1.94259644e-01,  -7.28515625e-01,
                 -4.92522240e-01,  -5.37109375e-01,  -3.47656250e-01,   1.08642578e-01,
                 -1.41601562e-01,  -2.07031250e-01,   2.52441406e-01,  -7.78808594e-02,
                 -5.02441406e-01,   1.53808594e-02,   8.64257812e-02,   2.59765625e-01,
                  6.64062500e-02,  -7.12890625e-01,  -1.45751953e-01,   7.56835938e-03,
                  4.87792969e-01,   1.39160156e-01,   1.15722656e-01,   1.28662109e-01,
                 -4.75585938e-01,   2.21191406e-01,   3.25317383e-01,   1.06323242e-01,
                 -6.11083984e-01,  -3.59619141e-01,   6.54296875e-02,  -2.41699219e-01,
                 -6.29882812e-02,  -1.62109375e-01,   4.26269531e-01,  -4.38354492e-01,
                  1.93725586e-01,   4.89562988e-01,   5.31494141e-01,  -7.29370117e-02,
                  1.77246094e-01,   9.39941406e-02,   2.92236328e-01,  -2.74047852e-01,
                  2.63366699e-02,   4.36035156e-01,  -3.76953125e-01,   3.10546875e-01,
                  4.87304688e-01,  -2.43041992e-01,   1.21612549e-02,  -3.80371094e-01,
                  3.80493164e-01,  -6.22436523e-01,  -3.98071289e-01,   1.24206543e-01,
                 -8.20312500e-01,  -2.72583008e-01,  -6.21582031e-01,  -4.87060547e-01,
                  3.06671143e-01,  -2.61230469e-01,   5.12451172e-01,   5.55694580e-01,
                  5.66894531e-01,   7.33886719e-01,  -1.75781250e-01,   4.13574219e-01,
                 -2.54272461e-01,   1.32507324e-01,  -4.78515625e-01,   4.63256836e-01,
                 -6.21948242e-02,  -1.80664062e-01,  -5.46386719e-01,  -6.31103516e-01,
                 -1.47949219e-01,  -3.15185547e-01,  -7.12890625e-02,  -7.67578125e-01,
```

```
       3.92272949e-01, -1.97753906e-01,  2.23144531e-01, -5.07324219e-01,
       8.39843750e-02, -4.98657227e-02,  1.01074219e-01,  2.07885742e-01,
      -2.77343750e-01,  1.03027344e-01, -1.38671875e-01,  2.87353516e-01,
      -4.81895447e-01, -1.66748047e-01, -1.47277832e-01,  3.61633301e-01,
       6.38504028e-02, -6.69189453e-01,  1.95312500e-03, -7.34375000e-01,
      -1.28158569e-01,  9.76562500e-04, -7.08007812e-02,  3.72558594e-01,
       8.31176758e-01,  5.94482422e-01,  5.37109375e-02, -3.00140381e-01,
      -4.53857422e-01,  1.11511230e-01, -1.32812500e-01,  1.25732422e-01,
       3.39843750e-01, -2.48352051e-01, -1.62353516e-02, -2.84667969e-01,
       4.70703125e-01, -4.48242188e-01,  8.50753784e-02,  2.69042969e-01,
       3.98254395e-03, -3.53759766e-01, -3.90625000e-02, -3.22753906e-01,
      -6.90917969e-02, -4.13818359e-02,  1.35314941e-01, -8.50396156e-02,
       1.28417969e-01,  6.15966797e-01,  3.55957031e-01, -6.05468750e-02,
      -2.25463867e-01, -2.62207031e-01, -2.72949219e-01, -5.16113281e-01,
       1.59179688e-01,  2.74902344e-01, -7.61718750e-02, -3.41796875e-03,
       4.37500000e-01,  2.98583984e-01, -4.40795898e-01, -3.43261719e-01,
       1.73583984e-01,  3.32092285e-01, -2.12646484e-01,  5.76171875e-01,
       2.06787109e-01, -7.91015625e-02,  5.79695702e-02, -1.01806641e-01,
      -7.06787109e-01, -3.40576172e-02, -4.11865234e-01,  9.82666016e-02,
      -1.70410156e-01, -4.18212891e-01,  8.39233398e-01, -1.15722656e-01,
       1.28173828e-01, -2.07763672e-01, -4.08203125e-01, -1.77612305e-01,
       1.01196289e-01,  4.24072266e-01, -5.26428223e-02, -5.58593750e-01,
       1.12304688e-02, -1.12060547e-01, -9.42382812e-02,  2.35595703e-02,
      -3.92578125e-01, -7.12890625e-02,  5.69824219e-01,  9.81445312e-02],
     dtype=float32)
```

In [16]: `find_closest_word(doc2vec)`

Out[16]: `'petroleum'`

Congratulations! You have finished the introduction to word embeddings manipulation!