Chapter 13
Workshop#4 Semantic Analysis and Word Vectors Using spaCy (Hour 7–8)

13.1 Introduction

In Chaps. 5 and 6, we studied the basic concepts and theories related to meaning representation and semantic analysis. This workshop will explore how to use spaCy technology to perform semantic analysis starting from a revisit on word vectors concept, implement and pre-train them followed by the study of similarity method and other advanced semantic analysis.

13.2 What Are Word Vectors?

Word vectors (Albrecht et al. 2020; Bird et al. 2009; Hardeniya et al. 2016; Kedia and Rasu 2020; NLTK 2022) are practical tools in NLP.

A word vector is a dense representation of a word. Word vectors are important for semantic similarity applications like similarity calculations between words, phrases, sentences, and documents, e.g. they provide information about synonymity, semantic analogies at word level.

Word vectors are produced by algorithms to reflect similar words appear in similar contexts. This paradigm captures target word meaning by collecting information from surrounding words which is called distributional semantics.

They are accompanied by associative semantic similarity methods including word vector computations such as distance, analogy calculations, and visualization to solve NLP problems.

R. S. T. Lee, *Natural Language Processing*

In this workshop, we are going to cover the following main topics (Altinok 2021; Arumugam and Shanmugamani 2018; Perkins 2014; spaCy 2022; Srinivasa-Desikan 2018; Vasiliev 2020):

- Understanding word vectors.
- Using spaCy's pre-trained vectors.
- Advanced semantic similarity methods.

13.3 Understanding Word Vectors

Word vectors, or word2vec are important quantity units in statistical methods to represent text in statistical NLP algorithms. There are several ways of text vectorization to provide words semantic representation.

13.3.1 Example: A Simple Word Vector

Let us look at a basic way to assign words vectors:

- Assign an index value to each word in vocabulary and encode this value into a sparse vector.
- Consider *tennis* as vocabulary and assign an index to each word according to vocabulary order as in Fig. 13.1:

Vocabulary word vector will be 0, except for word corresponding index value position as in Fig. 13.2:

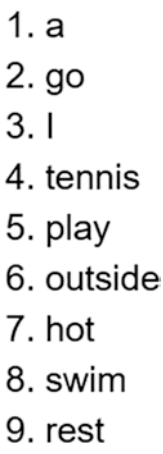


Fig. 13.1 A basic word vector example consists of 9 words

word									
a	1	0	0	0	0	0	0	0	0
go	0	1	0	0	0	0	0	0	0
I	0	0	1	0	0	0	0	0	0
tennis	0	0	0	1	0	0	0	0	0
play	0	0	0	0	1	0	0	0	0
outside	0	0	0	0	0	1	0	0	0
hot	0	0	0	0	0	0	1	0	0
swim	0	0	0	0	0	0	0	1	0
today	0	0	0	0	0	0	0	0	1

Fig. 13.2 Word vectors corresponding index value consists of 9 words

word									
I	0	0	1	0	0	0	0	0	0
play	0	0	0	0	1	0	0	0	0
tennis	0	0	0	1	0	0	0	0	0
today	0	0	0	0	0	0	0	0	1

Fig. 13.3 Word vector matrix for *I play tennis today*

Since each row corresponds to one word, a sentence represents a matrix, e.g. *I play tennis today* is represented by matrix as in Fig. 13.3:

Vectors length is equal to word numbers in vocabulary as shown above. Each dimension is apportioned to one word explicitly. When applying this encoding vectorization to text, each word is replaced by its vector and the sentence is transformed into a (N, V) matrix, where N is words number in sentence and V is vocabulary size.

This text representation is easy to compute, debug, and interpret. It looks good so far but there are potential problems:

- Vectors are sparse. Each vector contains many 0 s but has one 1. If words have similar meanings, they can group to share dimensions, this vector will deplete space. Also, numerical algorithms do not accept high dimension and sparse vectors in general.
- A sizeable vocabulary is comparable to high dimensions vectors that are impractical for memory storage and computation.

- Similar words do not assign with similar vectors resulting in unmeaningful vectors, e.g. *cheese, topping, salami,* and *pizza* have related meanings but have unrelated vectors. These vectors depend on corresponding word's index and assign randomly in vocabulary, indicating that one-hot encoded vectors are incapable to capture semantic relationships and against word vectors' purpose to answer preceding list concerns.

13.4 A Taste of Word Vectors

A word vector is a fixed-size, dense, and real-valued vector. It is a learnt representation of text where semantic similar words correspond to similar vectors and a solution to preceding problems.

```
the 0.418 0.24968 -0.41242 0.1217 0.34527 -0.044457 -0.49688 -0.17862 -0.00066023 -0.6566
0.27843 -0.14767 -0.55677 0.14658 -0.0095095 0.011658 0.10204 -0.12792 -0.8443 -0.12181
-0.016801 -0.33279 -0.1552 -0.23131 -0.19181 -1.8823 -0.76746 0.099051 -0.42125 -0.19526
4.0071 -0.18594 -0.52287 -0.31681 0.00059213 0.0074449 0.17778 -0.15897 0.012041
-0.054223 -0.29871 -0.15749 -0.34758 -0.045637 -0.44251 0.18785 0.0027849 -0.18411
-0.11514 -0.78581
```

This is a 50-dimensional vector for word *the,* these dimensions have floating points:

1. What do dimensions represent?
2. These individual dimensions do not have inherent meanings typically but instead they represent vector space locations, and the distance between these vectors indicates the similarity of corresponding words' meanings
3. Hence, a word's meaning is distributed across dimensions
4. This type of word's meaning representation is called distributional semantics

Use word vector visualizer for TensorFlow from (TensorFlow 2022) Google which offers word vectors for 10,000 words. Each vector is 200-dimensional and projected into three dimensions for visualization. Let us look at the representation of *tennis* as in Fig. 13.4:

tennis is semantically grouped with other sports, i.e. hockey, basketball, chess, etc. Words in proximity are calculated by their cosine distances as shown in Fig. 13.5

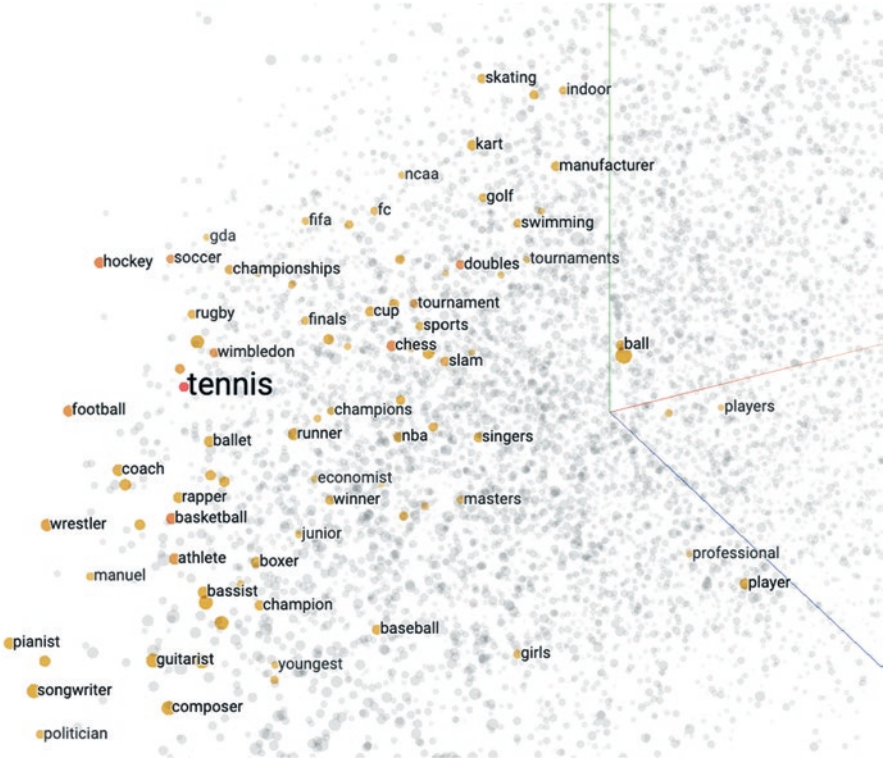


Fig. 13.4 Vector representation of *tennis* and semantic similar words

Word vectors are trained on a large corpus such as Wikipedia which included to learn proper nouns representations, e.g. *Alice* is a proper noun represented by vector as in Fig. 13.6:

It showed that all vocabulary input words are in lower cases to avoid multiple representations of the same word. *Alice* and *Bob* are person names to be listed. In addition, *lewis* and *carroll* have relevance to *Alice* because *of the famous literature Alice's Adventures in Wonderland* written by *Lewis Carroll*. Further, it also showed syntactic category of all neighboring words are nouns but not verbs.

Word vectors can capture synonyms, antonyms, and semantic categories such as animals, places, plants, names, and abstract concepts.

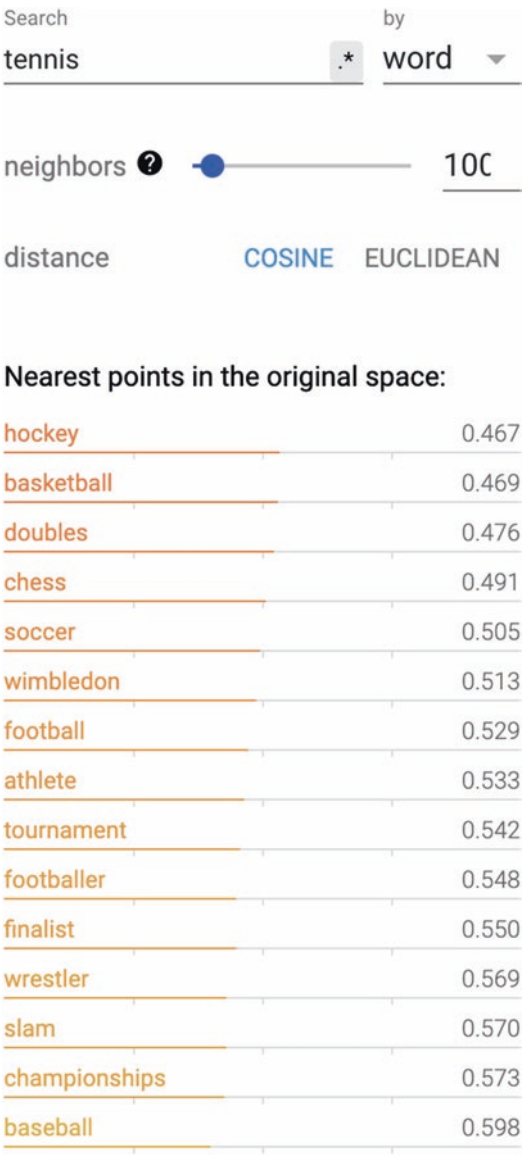


Fig. 13.5 *tennis* proximity words in three-dimensional space

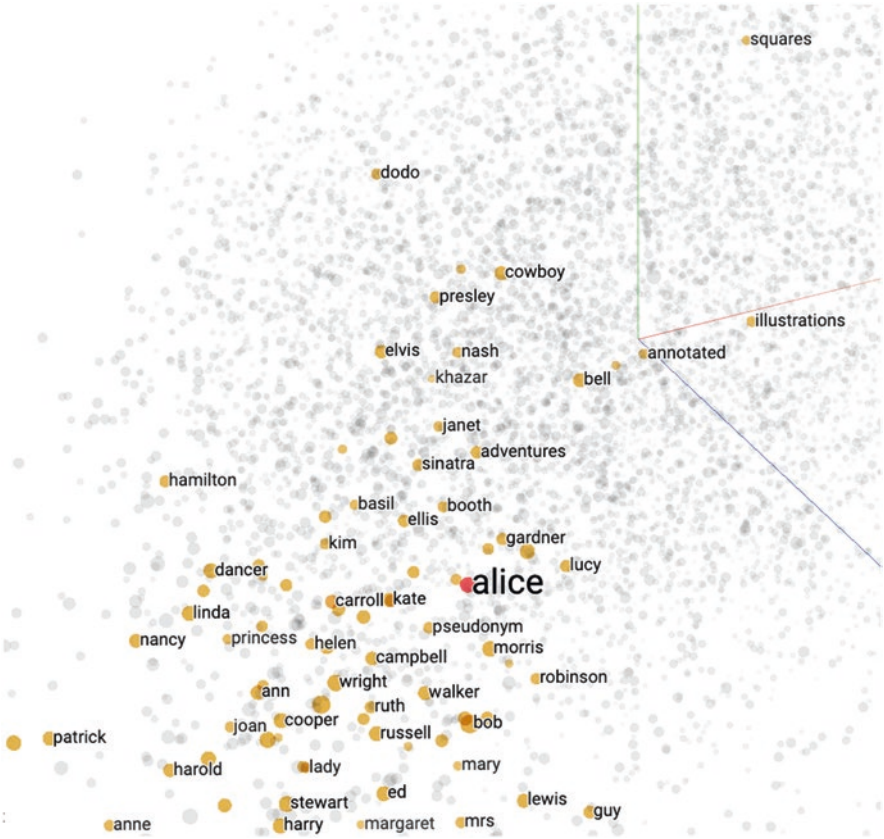


Fig. 13.6 Vector representation of *alice*

13.5 Analogies and Vector Operations

Word vectors capture semantics, support vector addition, subtraction, and analogies. A word analogy is a semantic relationship between a pair of words. There are many relationship types such as synonymity, anonymity, and whole-part relation. Some example pairs are (*King—man*, *Queen—woman*), (*airplane—air*, *ship—sea*), (*fish—sea*, *bird—air*), (*branch—tree*, *arm—human*), (*forward—backward*, *absent—present*) etc.

For example, gender mapping represents *Queen* and *King* as *Queen—Woman + Man = King*. If *woman* is subtracted by *Queen* and add *Man* instead to obtain *King*. Then, this analogy interprets queen is attributed to king as woman is attributed to man. Embeddings can generate analogies such as gender, tense, and capital city as shown in Fig. 13.7:

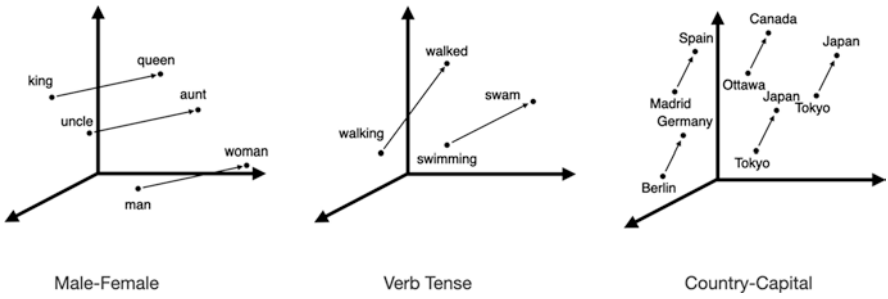


Fig. 13.7 Analogies created by word vectors

13.6 How to Create Word Vectors?

There are many ways to produce and pre-trained word vectors:

1. word2vec is a name of statistical algorithm created by Google to produce word vectors. Word vectors are trained with a neural network architecture to process windows of words and predicts each word vector depending on surrounding words. These pre-trained word vectors can be downloaded from Synthetic (2022).
2. Glove vectors are invented by Stanford NLP group. This method depends on singular value decomposition used in word co-occurrences matrix. The pre-trained vectors are available at nlp.stanford.edu (Stanford 2022).
3. fastText (FastText 2022) was created by Facebook Research like word2vec. word2vec predicts words based on their surrounding context, while fastText predicts subwords i.e. character N-grams. For example, the word *chair* generates the following subwords:

```
ch, ha, ai, ir, cha, hai, air
```

13.7 spaCy Pre-trained Word Vectors

Word vectors are part of many spaCy language models. For instance, en_core_web_md model ships with 300-dimensional vectors for 20,000 words, while en_core_web_lg model ships with 300-dimensional vectors with a 685,000 words vocabulary.

Typically, small models (names end with sm) do not include any word vectors but context-sensitive tensors. Semantic similarity calculations can perform but results will not be accurate as word vector computations.

A word's vector is via token.vector method. Let us look at this method using code query word vector for *banana*:

In[1]
```
# Import spaCy and load the en_core_web_md model
import spacy
nlp = spacy.load("en_core_web_md")

# Create a sample utterance (utt1)
utt1 = nlp("I ate a banana.")
```

In[2]
```
import en_core_web_md
nlp = en_core_web_md.load()
```

Use the following script to show Word Vector for *banana*:

In[3]
```
utt1[3].vector
```

Out[3]
```
array([ 2.0228e-01, -7.6618e-02,  3.7032e-01,  3.2845e-02, -4.1957e-01,
        7.2069e-02, -3.7476e-01,  5.7460e-02, -1.2401e-02,  5.2949e-01,
       -5.2380e-01, -1.9771e-01, -3.4147e-01,  5.3317e-01, -2.5331e-02,
        1.7380e-01,  1.6772e-01,  8.3984e-01,  5.5107e-02,  1.0547e-01,
        3.7872e-01,  2.4275e-01,  1.4745e-02,  5.5951e-01,  1.2521e-01,
       -6.7596e-01,  3.5842e-01, -4.0028e-02,  9.5949e-02, -5.0690e-01,
       -8.5318e-02,  1.7980e-01,  3.3867e-01,  1.3230e-01,  3.1021e-01,
        2.1878e-01,  1.6853e-01,  1.9874e-01, -5.7385e-01, -1.0649e-01,
        2.6669e-01,  1.2838e-01, -1.2803e-01, -1.3284e-01,  1.2657e-01,
        8.6723e-01,  9.6721e-02,  4.8306e-01,  2.1271e-01, -5.4990e-02,
       -8.2425e-02,  2.2408e-01,  2.3975e-01, -6.2260e-02,  6.2194e-01,
       -5.9900e-01,  4.3201e-01,  2.8143e-01,  3.3842e-02, -4.8815e-01,
       -2.1359e-01,  2.7401e-01,  2.4095e-01,  4.5950e-01, -1.8605e-01,
       -1.0497e+00, -9.7305e-02, -1.8908e-01, -7.0929e-01,  4.0195e-01,
       -1.8768e-01,  5.1687e-01,  1.2520e-01,  8.4150e-01,  1.2097e-01,
        8.8239e-02, -2.9196e-02,  1.2151e-03,  5.6825e-02, -2.7421e-01,
        ...
        1.7553e-01,  2.3049e-01,  2.8323e-01,  1.3882e-01,  3.1218e-03,
        1.7057e-01,  3.6685e-01,  2.5247e-03, -6.4009e-01, -2.9765e-01,
        7.8943e-01,  3.3168e-01, -1.1966e+00, -4.7156e-02,  5.3175e-01],
      dtype=float32)
```

In this example, *token.vector* returns a NumPy ndarray.
Use the following command to call NumPy methods for result.

In[4]
```
type(utt1[3].vector)
```

Out[4]
```
numpy.ndarray
```

In[5]
```
utt1[3].vector.shape
```
Out[5]
```
(300,)
```

Query Python type of word vector in this code segment. Then, invoke shape() method of NumPy array on the vector.

Doc and Span objects also have vectors. A sentence vector or a span is the average of words' vectors. Run following code and view results:

In[6]
```
# Create second utterance (utt2)
utt2 = nlp("I like a banana,")
utt2.vector
utt2[1:3].vector
```
Out[6]
```
array([-7.01859966e-02,  3.99469994e-02, -2.89449990e-01,  1.44250005e-01,
        3.08454990e-01, -3.68050039e-02,  5.54560013e-02, -3.84959996e-01,
       -1.67619996e-02,  2.18615007e+00, -2.49925002e-01, -6.11364990e-02,
        1.29144996e-01, -4.81765009e-02, -2.66460001e-01, -6.00790009e-02,
        5.89450002e-02,  1.18259001e+00, -2.54125506e-01, -7.12940022e-02,
       -4.14670035e-02, -2.59229988e-01, -2.93590009e-01,  1.39931455e-01,
       -3.80499959e-02, -2.37364992e-02, -3.13300081e-03, -1.39875501e-01,
        3.56814981e-01, -1.68979496e-01,  1.09136999e-01,  2.14434996e-01,
       -1.16950013e-02, -1.45829991e-02,  3.03380013e-01, -2.41475001e-01,
        1.53072998e-01, -6.51900023e-02, -2.27335006e-01, -3.70322496e-01,
        8.24549943e-02,  4.37065005e-01, -8.23500007e-03, -1.57661155e-01,
        1.33170500e-01, -2.15614997e-02, -3.75005007e-01, -1.87555000e-01,
       -9.86540020e-02,  1.52085498e-01, -3.38850021e-01,  4.04414982e-01,
        2.83199996e-02,  6.13904968e-02,  8.31245035e-02,  1.58006296e-01,
       -6.38950020e-02, -4.89500165e-03,  1.17890000e-01, -1.70920506e-01,
       -2.75707003e-02,  2.53172010e-01, -1.00804999e-01,  1.26459002e-01,
        2.05974996e-01, -3.44179988e-01, -1.34009004e-01,  1.12193003e-01,
        ....
       -1.32447511e-01, -1.34463996e-01,  2.20880002e-01,  1.96868494e-01,
        8.77650082e-02,  5.79900034e-02, -1.97049975e-02, -1.03452995e-01,
       -6.22849986e-02, -2.97744989e-01,  2.63655007e-01, -3.52320015e-01,
        1.48049995e-01, -1.52256504e-01, -3.67734991e-02,  9.91016999e-02,
        4.19600010e-02, -1.00619800e-01, -1.05862498e-01,  1.82905495e-01,
       -1.96168005e-01, -2.69535005e-01,  1.12740003e-01,  3.21924984e-01],
      dtype=float32)
```

Only words in model's vocabulary have vectors, words are not in vocabulary are called out-of-vocabulary (OOV) words. token.is_oov and token.has_vector are two methods to query whether a token is in the model's vocabulary and has a word vector:

In[7]
```
# Create the utterance 3
utt3 = nlp("You went there afskfsd.")
```

In[8]
```
for token in utt3:
    print( "Token is: ",token, "OOV: ", token.is_oov, "Token has vector:" , token.has_vector)
```

Out[8]
```
Token is:  You OOV:  False Token has vector: True
Token is:  went OOV:  False Token has vector: True
Token is:  there OOV:  False Token has vector: True
Token is:  afskfsd OOV:  True Token has vector: False
Token is:  . OOV:  False Token has vector: True
```

This is basically how to use spaCy's pretrained word vectors. Next, discover how to invoke spaCy's semantic similarity method on Doc, Span, and Token objects.

13.8 Similarity Method in Semantic Analysis

Every container type object has a similarity method that allows to calculate semantic similarity of other container objects by comparing word vectors in spaCy. Semantic similarity between two container objects is different container types. For instance, a Token object to a Doc object and a Doc object to a Span object.

The following example computes two Span objects similarity:

In[9]
```
# Create utt4 and utt5 and measure the similarity
utt4 = nlp("I visited England.")
utt5 = nlp("I went to London.")
utt4[1:3].similarity(utt5[1:4])
```

Out[9]
```
0.6539691090583801
```

Compare two Token objects, London and England:

In[10]
```
utt4[2]
```
Out[10]
```
England
```

In[11]
```
utt4[2].similarity(utt5[3])
```
Out[11]
```
0.7389127612113953
```

The sentence's similarity is computed by calling similarity() on Doc objects:

In[12]
```
utt4.similarity(utt5)
```
Out[12]
```
0.8771558796234277
```

1. The preceding code segment calculates semantic similarity between two sentences *I visited England* and *I went to London*
2. Similarity score is high enough to consider both sentences are similar (similarity degree ranges from 0 to 1, 0 represents unrelated and 1 represents identical)

similarity() method returns 1 compare an object to itself unsurprisingly:

In[13]
```
utt4.similarity(utt4)
```
Out[13]
```
1.0
```

Judge the distance with numbers is complex but review vectors on paper can understand how vocabulary word groups are formed.

Code snippet below visualizes a vocabulary of two graphical semantic classes. The first word class is for animals and the second class is for food.

In[14]
```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import numpy as np
import spacy

nlp = spacy.load( "en_core_web_md" )

vocab = nlp( "cat dog tiger elephant bird monkey lion cheetah burger pizza
food cheese wine salad noodles macaroni fruit vegetable" )

words = [word.text for word in vocab]
```

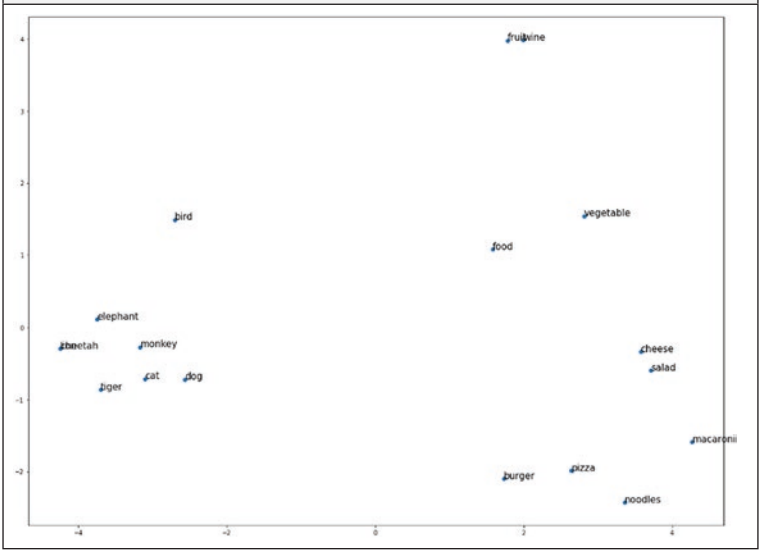
Create Word Vector vecs:

In[15]
```
vecs = np.vstack([word.vector for word in vocab if word.has_vector])
```

Use PCA (Principal Component Analysis) similarity analysis and plot similarity results with plt class.

In[16]
```
pca = PCA(n_components=2)
vecs_transformed = pca.fit_transform(vecs)
plt.figure(figsize=(20,15))
plt.scatter(vecs_transformed[:,0], vecs_transformed[:,1])
for word, coord in zip(words, vecs_transformed):
    x,y = coord
    plt.text(x,y,word, size=15)
plt.show()
```

Out[16]



1. Import matplotlib library to create a graph.
2. Next two imports are for vectors calculation.
3. Import spaCy and create a nlp object.
4. Create a Doc object from vocabulary.
5. Stack word vectors vertically by calling np.vstack.
6. Project vectors into a two-dimensional space for visualization since they are 300-dimensional. Extract two principal components via principal component analysis (PCA) for projection.
7. Create a scatter plot for rest of the code to deal with matplotlib function calls.

It showed that spaCy word vectors can visualize two semantic classes that are grouped. The distance between animals is reduced and uniformly distributed, while food class formed groups within the group.

Workshop 4.1 Word Vector Analysis on The Adventures of Sherlock Holmes
In this workshop, we have just learnt how to use spaCy to produce word vector to compare the similarity of two text objects/document. Try to use *The Adventures of Sherlock Holmes* (Doyle 2019; Gutenberg 2022) to select two "presentative" texts from this detective story:
1. Read Adventures_Holmes.txt text file.
2. Save contents into a string object "holmes_doc".
3. Plot Semantic Graphs for these two texts.
4. Perform Similarity text for these two documents. See what are found.

13.9 Advanced Semantic Similarity Methods with spaCy

It has learnt that spaCy's similarity method can calculate semantic similarity to obtain scores but there are advanced semantic similarity methods to calculate words, phrases, and sentences similarity.

13.9.1 Understanding Semantic Similarity

It is necessary to identify example characteristics when collecting data or text data (any sort of data), i.e. calculate two text similarity scores. Semantic similarity is a metric to define the distance between texts based on semantics texts.

A mathematics is basically a distance function. Every metric induces a topology on vector space. Word vectors are vectors that can be used to calculate the distance between them as a similarity score.

There are two commonly used distance functions: (1) Euclidian distance and (2) cosine distance.

13.9.2 Euclidian Distance

Euclidian distance counts on vector magnitude and disregards orientation. If a vector is drawn from an origin, let us call it a *dog* vector to another point, call a *cat* vector and subtract one vector from and other, the distance represents the magnitude of vectors is shown in Fig. 13.8.

If two or more semantically similar words (*canine, terrier*) to *dog* and make it a text of three words, i.e. *dog canine terrier*. Obviously, the *dog* vector will now grow in magnitude, possibly in the same direction. This time, the distance will be much

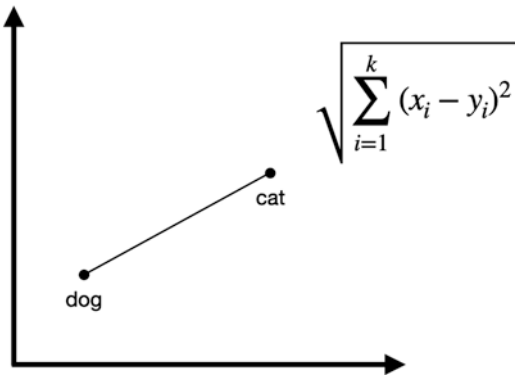


Fig. 13.8 Euclidian distance between two vectors: *dog* and *cat*

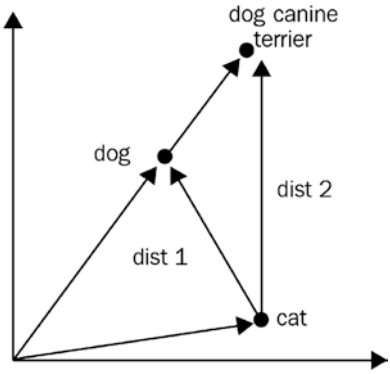


Fig. 13.9 Distance between *dog* and *cat*, as well as the distance between *dog canine terrier* and *cat*

bigger due to geometry, although the semantics of first piece of text (now *dog canine terrier*) remain the same.

This is the main drawback of using Euclidian distance for semantic similarity as the orientation of two vectors in the space is not considered. Figure 13.9 illustrates the distance between *dog* and *cat*, and the distance between *dog canine terrier* and *cat*.

How can we fix this problem? There is another way of calculating similarity called cosine similarity to address this problem.

13.9.3 Cosine Distance and Cosine Similarity

Contrary to Euclidian distance, cosine distance is more concerned with the orientation of two vectors in the space. The cosine similarity of two vectors is basically the cosine of angle created by these two vectors. Figure 13.10 shows the angle between *dog* and *cat* vectors:

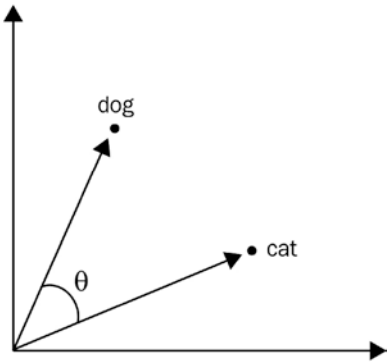


Fig. 13.10 The angle between *dog* and *cat* vectors. Here, the semantic similarity is calculated by cos(θ)

The maximum similarity score that is allowed by cosine similarity is 1. This is obtained when the angle between two vectors is 0 degree (hence, the vectors coincide). The similarity between two vectors is 0 when the angle between them is 90 degrees.

Cosine similarity provides scalability when vectors grow in magnitude. If one of the input vectors is expanded as in Fig. 13.10, the angle between them remains the same and so as the cosine similarity score.

Note that here is to calculate semantic similarity score and not distance. The highest possible value is 1 when vectors coincide, while the lowest score is 0 when two vectors are perpendicular. The cosine distance is $1 - \cos(\theta)$ which is a distance function.

spaCy uses cosine similarity to calculate semantic similarity. Hence, calling the similarity method helps to perform cosine similarity calculations.

So far, we have learnt to calculate similarity scores, but still have not discovered words meaning. Obviously, not all words in a sentence have the same impact on the semantics of sentence. The similarity method will only calculate the semantic similarity score, but the right keywords are required for calculation results comparison.

Consider the following text snippet:

```
Blue whales are the biggest mammals in the world. They're observed in California
coast during spring.
```

If is interested in finding the biggest mammals on the planet, the phrases *biggest mammals* and *in the world* will be keywords. By comparing these phrases with the search phrases *largest mammals* and *on the planet* should give a high similarity score. But if is interested in finding out about places in the world, *California* will be keyword. *California* is semantically like word *geography* and more suitably, the entity type is a geographical noun.

Since we have learnt how to calculate similarity score, the next section will learn about where to look for the meaning. It will cover a case study on text categorization before improving task results via key phrase extraction with similarity score calculations.

13.9.4 Categorizing Text with Semantic Similarity

Determining two sentences' semantic similarity can categorize texts into predefined categories or spot only the relevant texts. This case study will filter users' comments in an e-commerce website related to the word *perfume*. Suppose to evaluate the following comments:

```
I purchased a science fiction book last week.
I loved everything related to this fragrance: light, floral and feminine ···
I purchased a bottle of wine.
```

Here, it is noted that only the second sentence is related. This is because it contains the word *fragrance* and adjectives describing scents. To understand which sentences are related, can try several comparison strategies.

To start, compare *perfume* to each sentence. Recall that spaCy generates a word vector for a sentence by averaging the word vector of its tokens. The following code snippet compares preceding sentences to *perfume* search key:

In[17]
```
utt6 = nlp( "I purchased a science fiction book last week. I loved everything related to this fragrance: light, floral and feminine... I purchased a bottle of wine.  " )
key = nlp( "perfume" )
for utt in utt6.sents:
    print(utt.similarity(key))
```

Out[17]
```
0.2481654331382154
0.5075297559861377
0.4215429463030136
0.0
```

The following steps are performed:

Create a Doc object with three preceding sentences. For each sentence, calculate similarity score with *perfume* and print the score by invoking similarity() method on the sentence. The degree of similarity between *perfume* and the first sentence is minute, indicating that this sentence is irrelevant to the search key. The second sentence looks relevant which means that semantic similarity is correctly identified.

How about the third sentence? The script identified that the third sentence is relevant somehow, most probably because it includes the word *bottle* and perfumes are sold in bottles. The word *bottle* appears in similar contexts with the word *perfume*. For this reason, the similarity score of this sentence and search key is not small enough; also, the scores of second and third sentences are not distant enough to make the second sentence significant.

In practice, long texts such as web documents can be dealt with but averaging over them diminish the importance of keywords.

Let us look at how to identify key phrases in a sentence to improve performance.

13.9.5 Extracting Key Phrases

Semantic categorization is more effectively to extract important word phrases and compare them to the search key. Instead of comparing the key to different parts of speech, we can compare the key to noun phrases. Noun phrases are subjects, direct objects, and indirect objects of sentences that convey high percentages of sentences semantics.

For example, in sentence *Blue whales live in California*, focuses will likely be on *blue whales, whales, California,* or *whales in California*.

Similarly, in the preceding sentence about *perfume*, the focused is to pick out *fragrance* the noun. Different semantic tasks may need other context words such as verbs to decide what the sentence is about, but for semantic similarity, noun phrases convey significant weights.

What is a noun phrase? A noun phrase (NP) is a group of words that consist of a noun and its modifiers. Modifiers are usually pronouns, adjectives, and determiners. The following phrases are noun phrases:

```
A dog
My dog
My beautiful dog
A beautiful dog
A beautiful and happy dog
My happy and cute dog
```

spaCy extracts noun phases by parsing the output of dependency parser. It can identify noun phrases of a sentence by using doc.noun_chunks method:

```
In[18]  utt7 = nlp( "My beautiful and cute dog jumped over the fence" )

In[19]  utt7.noun_chunks
Out[19] <generator at 0x1932f2de900>

In[20]  list(utt7.noun_chunks)
Out[20] [My beautiful and cute dog, the fence]
```

Let us modify the preceding code snippet. Instead of comparing the search key perfume to the entire sentence, this time will only compare it with sentence's noun chunks:

In[21]:
```
for utt in utt7.sents:
    nchunks = [nchunk.text for nchunk in utt.noun_chunks]
    nchunk_utt = nlp(" ".join(nchunks))
    print(nchunk_utt.similarity(key))
```

Out[21]:
```
0.27409999728254997
```

The following is performed for the preceding code:
1. Iterate over sentences
2. Extract noun chunks and store them in a Python list for each sentence
3. Join noun chunks in the list into a Python string and convert it into a Doc object
4. Compare this Doc object of noun chunks to search key *perfume* to determine semantic similarity scores

If these scores are compared with previous scores, it is noted that that the first sentence remains irrelevant, so its score decreased marginally but the second sentence's score increased significantly. Also, the second and third sentences scores are distant from each other to reflect that second sentence is the most related sentence.

13.9.6 Extracting and Comparing Named Entities

In some cases, it can focus to extract proper nouns instead of every noun. Hence, it is required to extract named entities. Let us compare the following paragraphs:

```
"Google Search, often referred as Google, is the most popular search engine nowadays. It
answers a huge volume of queries every day."
"Microsoft Bing is another popular search engine. Microsoft is known by its star product
Microsoft Windows, a popular operating system sold over the world."
"The Dead Sea is the lowest lake in the world, located in the Jordan Valley of Israel. It
is also the saltiest lake in the world."
```

The codes should be able to recognize that first two paragraphs are about large technology companies and their products, whereas the third paragraph is about a geographic location.

Comparing all noun phrases in these sentences may not be helpful because many of them such as volume are irrelevant to categorization. The topics of these paragraphs are determined by phrases within them, that is, *Google Search, Google, Microsoft Bing, Microsoft, Windows, Dead Sea, Jordan Valley,* and *Israel*. spaCy can identify these entities:

In[22]
```
utt8 = nlp( "Google Search, often referred as Google, is the most popular search engine nowadays. It answers a huge volume of queries every day." )
utt9 = nlp( "Microsoft Bing is another popular search engine. Microsoft is known by its star product Microsoft Windows, a popular operating system sold over the world." )
utt10 = nlp( "The Dead Sea is the lowest lake in the world, located in the Jordan Valley of Israel. It is also the saltiest lake in the world." )
```

In[23]
```
utt8.ents
```
Out[23]
```
(Google Search, Google, every day)
```

In[24]
```
utt9.ents
```
Out[24]
```
(Microsoft Bing, Microsoft, Microsoft Windows)
```

In[25]
```
utt10.ents
```
Out[25]
```
(the Jordan Valley, Israel)
```

Since words are extracted for comparison, let's calculate similarity scores:

In[26]
```
ents1 = [ent.text for ent in utt8.ents]
ents2 = [ent.text for ent in utt9.ents]
ents3 = [ent.text for ent in utt10.ents]
ents1 = nlp(" ".join(ents1))
ents2 = nlp(" ".join(ents2))
ents3 = nlp(" ".join(ents3))
```

In[27]
```
ents1.similarity(ents2)
```
Out[27]
```
0.5394545341415748
```

In[28]
```
ents1.similarity(ents3)
```
Out[28]
```
0.48605042335384385
```
In[29]
```
ents2.similarity(ents3)
```
Out[29]
```
0.39674953175052086
```

These figures revealed that the highest level of similarity exists between first and second paragraphs, which are both about large tech companies. The third paragraph is unlike other paragraphs. How can this calculation be obtained by using word vectors only? It is probably because words *Google* and *Microsoft* often appear together in news and other social media text corpora, hence producing similar word vectors

This is the conclusion of advanced semantic similarity methods section with different ways to combine word vectors with linguistic features such as key phrases and named entities.

Workshop 4.2 Further Semantic Analysis on The Adventures of Sherlock Holmes

It has learnt to further improve semantic Analysis results on document similarity comparison by extracting (1) key phrases; (2) and comparing names entities. Try to use these techniques on *The Adventures of Sherlock Holmes*:

1. Extract three "representative texts" from this novel
2. Perform key phrases extraction to improve the similarity rate as compared with Workshop 4.1 results
3. Extract and compare name entities to identify significant name entities from this literature to further improve semantic analysis performance
4. Remember to plot semantic diagram to show how these entities and keywords are related
5. Discuss and explain what are found

References

Albrecht, J., Ramachandran, S. and Winkler, C. (2020) Blueprints for Text Analytics Using Python: Machine Learning-Based Solutions for Common Real World (NLP) Applications. O'Reilly Media.

Altinok, D. (2021) Mastering spaCy: An end-to-end practical guide to implementing NLP applications using the Python ecosystem. Packt Publishing.

Arumugam, R., & Shanmugamani, R. (2018). Hands-on natural language processing with python. Packt Publishing.

Bird, S., Klein, E., and Loper, E. (2009). Natural language processing with python. O'Reilly.

Doyle, A. C. (2019) The Adventures of Sherlock Holmes (AmazonClassics Edition). AmazonClassics.

FastText (2022) FastText official site. https://fasttext.cc/. Accessed 22 June 2022.

Gutenberg (2022) Project Gutenberg official site. https://www.gutenberg.org/ Accessed 16 June 2022.

Hardeniya, N., Perkins, J. and Chopra, D. (2016) Natural Language Processing: Python and NLTK. Packt Publishing.

Kedia, A. and Rasu, M. (2020) Hands-On Python Natural Language Processing: Explore tools and techniques to analyze and process text with a view to building real-world NLP applications. Packt Publishing.

NLTK (2022) NLTK official site. https://www.nltk.org/. Accessed 16 June 2022.

Perkins, J. (2014). Python 3 text processing with NLTK 3 cookbook. Packt Publishing Ltd.

SpaCy (2022) spaCy official site. https://spacy.io/. Accessed 16 June 2022.

Srinivasa-Desikan, B. (2018). Natural language processing and computational linguistics: A practical guide to text analysis with python, gensim, SpaCy, and keras. Packt Publishing Limited.
Stanford (2022) NLP.stanford.edu Glove official site. https://nlp.stanford.edu/projects/glove/. Accessed 22 June 2022.
Synthetic (2022) Synthetic Intelligent Network site on Word2Vec Model. https://developer.syn.co.in/tutorial/bot/oscova/pretrained-vectors.html#word2vec-and-glove-models. Accessed 22 June 2022.
TensorFlow (2022) TensorFlow official site. https://projector.tensorflow.org/. Accessed 22 June 2022.
Vasiliev, Y. (2020) Natural Language Processing with Python and spaCy: A Practical Introduction. No Starch Press.