soc128d_notebook_9_training_word_embeddings

July 28, 2021

Sociology 128D: Mining Culture Through Text Data: Introduction to Social Data Science

1 Notebook 9: Training Word Embeddings using `gensim`

In this notebook, we'll explore implementing the popular word2vec algorithm for training word embeddings.

[36]:
```
import os
import pandas as pd
import time

from collections import Counter
from gensim.models.callbacks import CallbackAny2Vec
from gensim.models.word2vec import Word2Vec
```

1.1 Setting up the Data

You can get the data from Canvas (Files -> Data)

[4]:
```
f = "rjobs_2020_preprocessed.json"
```

[5]:
```
df = pd.read_json(f)
```

[6]:
```
df.head()
```

[6]:
```
       id  author_id  score  num_comments  \
64761   0     946515      1             4
1867    1     116092      1             1
11033   2     385865      1             2
670     3     969296      1             1
9633    4     317635      1             4

                                          preprocessed                 date  \
64761      need job m m need job need money place apply  2020-11-22 20:37:06
1867   make post low quality post try ask help get re… 2020-01-11 23:50:18
11033  recruiter reach position experience hello toda… 2020-02-20 00:24:54
670    sudden mental block lose drive work work year … 2020-01-05 19:47:36
```

```
9633   critique revise resume research academia data … 2020-02-13 21:15:27

       dayofyear  hour  dayofmonth  month  dayofweek  week  day_name  \
64761        327    20          22     11          6    47    Sunday
1867          11    23          11      1          5     2  Saturday
11033         51     0          20      2          3     8  Thursday
670            5    19           5      1          6     1    Sunday
9633          44    21          13      2          3     7  Thursday

      month_name
64761   November
1867     January
11033   February
670      January
9633    February
```

[7]:
```
df.shape
```

[7]:
```
(49872, 14)
```

Since we are training a word embedding model, we will only use the text. Any analyses we do will be based on relationships among words (well, their embeddings), so we won't use document-level metadata like the date or number of replies.

The line of code below does three things: 1. `.apply(str.split)` splits the preprocessed text on whitespace 2. `.tolist()` converts the column to a list 3. Finally, we assign the result–a list of lists–to the variable text

[8]:
```
text = df["preprocessed"].apply(str.split).tolist()
```

[9]:
```
print(text[-1])
```

```
['need', 'advice', 'emailing', 'reconnecte', 'recruiter', 'covid', 'hi',
'touch', 'recruiter', 'month', 'ago', 'send', 'application', 'interested',
'say', 'open', 'position', 'right', 'email', 'assume', 'opening', 'covid',
'come', 'know', 'probably', 'new', 'people', 'want', 'email', 'forget', 'order',
'unreliable', 'agree', 'connect', 'write', 'unaware', 'situation', 'include',
'email', 'interested', 'position', 'hope', 'chance', 'crisis', 'come', 'end',
'experience', 'waste', 'time', 'need', 'text', 'worth', 'respond', 'appreciate',
'advice']
```

[10]:
```
len(text)
```

[10]:
```
49872
```

The model will learn representations of words (as vectors) based on how the words are used. It will learn less about rare words. Below we can see the impact on the vocabulary size of excluding words that don't appear a minimum of five or 25 times.

[11]:
```
len(set(filter(lambda x: x[1] >= 5, Counter([word for post in text for word in␣
 ↪post]).items())))
```

[11]: 13022

[12]:
```
len(set(filter(lambda x: x[1] >= 25, Counter([word for post in text for word in␣
 ↪post]).items())))
```

[12]: 5875

[13]:
```
all_words = [word for post in text for word in post]
min_of_five = [word for word, count in Counter(all_words).items() if count >= 5]
min_of_twentyfive = [word for word, count in Counter(all_words).items() if␣
 ↪count >= 25]
```

If we exclude words that do not appear a minimum of 25 times, we lose words like "cheesecake" that actually seem quite relevant to the example below. Whether or not we keep a word like "cheesecake" matters for three reasons. First, if we exclude it, we do not get a word vector for it, and we cannot use it for any analyses. Second, representations for words like "factory" will be affect because they directly co-occur. Finally, if we remove words, we also bring the remaining words closer together.

[15]:
```
samp = df.loc[5239].preprocessed

samp_min_five = " ".join([word for word in samp.split() if word in min_of_five])
samp_min_twentyfive = " ".join([word for word in samp.split() if word in␣
 ↪min_of_twentyfive])

print(len(samp_min_five.split()), samp_min_five, "\n")
print(len(samp_min_twentyfive.split()), samp_min_twentyfive)
```

```
58 lol freshman college want job apply like alot place place want hire
cheesecake factory apply like week call hour ago mini interview think good talk
fast catch say interview wonder work work cheesecake factory tell usually happen
schedule interview email day ask repeat say lol interview idk look young idk
reconsider consider cheesecake factory fine dining establishment lol

55 lol freshman college want job apply like alot place place want hire factory
apply like week call hour ago mini interview think good talk fast catch say
interview wonder work work factory tell usually happen schedule interview email
day ask repeat say lol interview idk look young idk reconsider consider factory
fine dining establishment lol
```

We can speed things up by changing the number of workers!

[16]:
```
os.cpu_count()
```

[16]: 16

The class below is adapted from https://stackoverflow.com/a/58515344. It allows us to print the loss when we train the model for more than one epoch.

[17]:
```
class callback(CallbackAny2Vec):
    """
    Callback to print loss after each epoch.
    from https://stackoverflow.com/a/58515344
    """

    def __init__(self):
        self.epoch = 0
        self.loss_to_be_subed = 0

    def on_epoch_end(self, model):
        loss = model.get_latest_training_loss()
        loss_now = loss - self.loss_to_be_subed
        self.loss_to_be_subed = loss
        print(f"Loss after epoch {self.epoch}: {loss_now:,}")
        self.epoch += 1
```

1.2 Training a Basic Model

You can see the details of gensim's implementation of word2vec here. The sg argument let's use the skip-gram algorithm, and negative let's use specify the number of negative samples.

[31]:
```
basic_model = Word2Vec(text, vector_size=300, window = 7, sg = 1, negative = 5,␣
↪workers = os.cpu_count()-1, min_count = 5)
```

[32]:
```
type(basic_model)
```

[32]:
```
gensim.models.word2vec.Word2Vec
```

[33]:
```
basic_model = basic_model.wv
```

[ ]:
```
type(basic_model)
```

[34]:
```
basic_model.most_similar("employment")
```

[34]:
```
[('employement', 0.590010941028595),
 ('backcheck', 0.557177722454071),
 ('unexplained', 0.5360859632492065),
 ('asurint', 0.53061842918396),
 ('contingency', 0.5091899633407593),
 ('involuntary', 0.4967845678329468),
 ('binding', 0.4930131435394287),
 ('probational', 0.49218234419822693),
 ('fudge', 0.48438072204589844),
 ('stipulate', 0.48102515935897827)]
```

[35]:
```
basic_model.most_similar("job")
```

[35]:
```
[('widen', 0.6490814089775085),
 ('carmax', 0.6279255151748657),
 ('headway', 0.6188454627990723),
 ('nineteen', 0.6154090762138367),
 ('peruse', 0.6147359013557434),
 ('fourteen', 0.6141305565834045),
 ('interstate', 0.6117398142814636),
 ('albertsons', 0.6105733513832092),
 ('relentlessly', 0.6091293692588806),
 ('scarcity', 0.6089302897453308)]
```

1.3 Training a Better Model

Now let's compare that model to a better model. We're going to train a model several epochs, meaning the model will have several chances to update the word embeddings and improve them.

[27]:
```
start_time = time.time()

model = Word2Vec(text, vector_size = 300, window = 7, sg = 1, negative = 5,␣
 ↪workers = os.cpu_count()-1, min_count = 5,
                 epochs = 100, callbacks=[callback()], compute_loss = True)

minutes = (time.time() - start_time)/60
print(f"Training completed in {minutes:.1f} minutes.")
```

```
Loss after epoch 0: 3,494,915.25
Loss after epoch 1: 3,471,232.25
Loss after epoch 2: 3,095,479.5
Loss after epoch 3: 2,418,138.0
Loss after epoch 4: 2,728,945.0
Loss after epoch 5: 2,547,640.0
Loss after epoch 6: 2,620,300.0
Loss after epoch 7: 2,620,006.0
Loss after epoch 8: 2,626,252.0
Loss after epoch 9: 2,230,428.0
Loss after epoch 10: 2,504,620.0
Loss after epoch 11: 2,529,150.0
Loss after epoch 12: 1,652,202.0
Loss after epoch 13: 1,554,964.0
Loss after epoch 14: 1,561,304.0
Loss after epoch 15: 1,562,160.0
Loss after epoch 16: 1,315,100.0
Loss after epoch 17: 1,490,456.0
Loss after epoch 18: 1,311,044.0
Loss after epoch 19: 1,295,232.0
Loss after epoch 20: 1,488,840.0
Loss after epoch 21: 1,424,360.0
Loss after epoch 22: 1,365,596.0
```

```
Loss after epoch 23: 1,465,004.0
Loss after epoch 24: 1,272,900.0
Loss after epoch 25: 1,288,124.0
Loss after epoch 26: 1,469,328.0
Loss after epoch 27: 1,401,712.0
Loss after epoch 28: 1,500,004.0
Loss after epoch 29: 1,328,048.0
Loss after epoch 30: 1,262,048.0
Loss after epoch 31: 1,450,748.0
Loss after epoch 32: 1,496,568.0
Loss after epoch 33: 1,374,428.0
Loss after epoch 34: 1,422,720.0
Loss after epoch 35: 1,308,912.0
Loss after epoch 36: 198,140.0
Loss after epoch 37: 48,360.0
Loss after epoch 38: 41,856.0
Loss after epoch 39: 47,664.0
Loss after epoch 40: 42,392.0
Loss after epoch 41: 45,040.0
Loss after epoch 42: 48,288.0
Loss after epoch 43: 43,184.0
Loss after epoch 44: 47,848.0
Loss after epoch 45: 42,536.0
Loss after epoch 46: 43,800.0
Loss after epoch 47: 48,480.0
Loss after epoch 48: 43,152.0
Loss after epoch 49: 38,960.0
Loss after epoch 50: 38,048.0
Loss after epoch 51: 40,552.0
Loss after epoch 52: 40,568.0
Loss after epoch 53: 42,256.0
Loss after epoch 54: 40,056.0
Loss after epoch 55: 41,984.0
Loss after epoch 56: 43,728.0
Loss after epoch 57: 42,384.0
Loss after epoch 58: 43,168.0
Loss after epoch 59: 41,960.0
Loss after epoch 60: 44,584.0
Loss after epoch 61: 42,072.0
Loss after epoch 62: 37,856.0
Loss after epoch 63: 41,240.0
Loss after epoch 64: 41,976.0
Loss after epoch 65: 40,208.0
Loss after epoch 66: 35,544.0
Loss after epoch 67: 40,872.0
Loss after epoch 68: 33,488.0
Loss after epoch 69: 34,136.0
Loss after epoch 70: 33,224.0
```

```
Loss after epoch 71: 39,904.0
Loss after epoch 72: 41,024.0
Loss after epoch 73: 36,856.0
Loss after epoch 74: 38,824.0
Loss after epoch 75: 35,824.0
Loss after epoch 76: 32,824.0
Loss after epoch 77: 32,480.0
Loss after epoch 78: 41,072.0
Loss after epoch 79: 38,528.0
Loss after epoch 80: 38,888.0
Loss after epoch 81: 37,544.0
Loss after epoch 82: 32,848.0
Loss after epoch 83: 36,968.0
Loss after epoch 84: 38,008.0
Loss after epoch 85: 38,512.0
Loss after epoch 86: 35,176.0
Loss after epoch 87: 32,960.0
Loss after epoch 88: 32,496.0
Loss after epoch 89: 32,744.0
Loss after epoch 90: 30,904.0
Loss after epoch 91: 35,456.0
Loss after epoch 92: 34,272.0
Loss after epoch 93: 33,624.0
Loss after epoch 94: 30,976.0
Loss after epoch 95: 33,784.0
Loss after epoch 96: 28,736.0
Loss after epoch 97: 30,160.0
Loss after epoch 98: 27,440.0
Loss after epoch 99: 31,664.0
Training completed in 11.3 minutes.
```

[28]:
```
model.wv.most_similar("employment")
```

[28]:
```
[('employer', 0.41440436244010925),
 ('employ', 0.3883063793182373),
 ('asurint', 0.385675847530365),
 ('backcheck', 0.38199561834335327),
 ('indemnity', 0.3556252121925354),
 ('probational', 0.3533172607421875),
 ('job', 0.3499438166618347),
 ('verification', 0.3492674231529236),
 ('employement', 0.3435443639755249),
 ('gap', 0.3432706296443939)]
```

[29]:
```
model.wv.most_similar("job")
```

```
[29]: [('position', 0.5627887845039368),
       ('look', 0.4782736599445343),
       ('find', 0.46779850125312805),
       ('apply', 0.4550776779651642),
       ('month', 0.45464614033699036),
       ('work', 0.45331627130508423),
       ('search', 0.4172516465187073),
       ('start', 0.414230078458786),
       ('get', 0.4109322428703308),
       ('recently', 0.4074569046497345)]
```