

Data Science Lab

October 24, 2019

0.1 Global Land Temperature

0.1.1 Exercise 1.1

To load the dataset, the csv module can be used. This time, the dataset file contains an header in the first line. We will read it beforehand to get the names of the columns.

```
[1]: import csv

col_names = []
dataset = [[], [], [], [], [], [], []]
with open('GLT_filtered.csv') as fp:
    reader = csv.reader(fp)
    col_names = next(reader)
    for cols in reader:
        if len(cols) == len(col_names): # every column must be present
            for i in range(len(dataset)):
                dataset[i].append(cols[i])
```

Let's check if the parsing was successful by analyzing a few lines.

```
[2]: for i in range(4):
        for j in range(len(dataset)):
            print(f'{dataset[j][i]} ', end='')
        print('')
```

```
1849-01-01 26.704 1.435 Abidjan Côte D'Ivoire 5.63N 3.23W
1849-02-01 27.434 1.3619999999999999 Abidjan Côte D'Ivoire 5.63N 3.23W
1849-03-01 Abidjan Côte D'Ivoire 5.63N 3.23W
1849-04-01 26.14 1.3869999999999998 Abidjan Côte D'Ivoire 5.63N 3.23W
```

While the total number of rows is:

```
[3]: rows_c = len(dataset[1])
      rows_c
```

```
[3]: 219575
```

Note: the reader object splits each row and parses it into a list of strings. When a value is missing, the item of the list is casted coherently to an empty string: ''.

As you can see, many of the columns are nominal, while there are two numerical continuous attributes: AverageTemperature and AverageTemperatureUncertainty.

0.1.2 Exercise 1.2

The columns **AverageTemperature** and **AverageTemperatureUncertainty** contain missing values. By simply counting them, we obtain:

```
[4]: def count_missing(data):  
      return sum([1 for d in data if d == ''])  
  
count = len(dataset[1])  
print(f'AverageTemperature, missing values out of the whole dataset:␣  
      ↳{100*count_missing(dataset[1])/count:.1f}%')  
print(f'AverageTemperatureUncertainty, missing values out of the whole dataset:␣  
      ↳{100*count_missing(dataset[2])/count:.1f}%')
```

AverageTemperature, missing values out of the whole dataset: 12.6%

AverageTemperatureUncertainty, missing values out of the whole dataset: 12.6%

It can easily be seen that the rows in the dataset are sorted by City and then by Date. Therefore, filling the missing values in columns requires to take that into account. Specifically, only the measurements of the current city can be considered in the process. Let's develop two solutions to address the problem, with their respective pros and cons.

Version A Since there are two columns to fill, we will define a function that accepts a list of string for the temperatures and a list of strings for the cities.

```
[5]: def fill_gaps(data, cities):  
      right_i = 0  
      right_v = 0  
  
      for i, value in enumerate(data):  
          if i == 0 or cities[i] != cities[i-1]:  
              left_v = 0  
          else:  
              left_v = data[i-1]  
  
          # reuse the right_v value, useful when there are multiple consecutive␣  
          ↳missing values  
          if i < right_i:  
              data[i] = (left_v + right_v) / 2  
              continue  
  
          if value == '':  
              for j in range(i+1, len(data)):  
                  if cities[j] != cities[i]: # this check must come before  
                      right_v = 0  
                      break  
                  elif data[j] != '':  
                      right_v = float(data[j])  
                      break
```

```

        if i == len(data)-1: # edge case: the last value of the last city
→is empty
            right_v = 0

            right_i = j
            data[i] = (left_v + right_v) / 2

        else:
            data[i] = float(data[i]) # parse to float all present values

```

We can test it with a toy example:

```

[6]: l = ['', '12', '', '', '', '15', '']
      c = ['Rome', 'Rome', 'Rome', 'Turin', 'Turin', 'Turin', 'Turin']

      print('Original list:', l)
      fill_gaps(l, c)
      print('Filled list:', l)

```

Original list: ['', '12', '', '', '', '15', '']

Filled list: [6.0, 12.0, 6.0, 7.5, 11.25, 15.0, 7.5]

Note: this example also lets you reflect on the definition of mutable and immutable objects. Notice that modifying a list, one of the mutables in the Python world, in the scope of a function that receives it as parameter, applies the changes to the original list.

Let's apply now the function to the lists of interest and verify that there are no more missing values.

```

[7]: avg_temp = dataset[1]
      avg_temp_unc = dataset[2]
      cities = dataset[3]

      fill_gaps(avg_temp, cities)
      fill_gaps(avg_temp_unc, cities)

      print('Missing values in AverageTemperature:', sum([1 for v in avg_temp if v ==
→'']))
      print('Missing values in AverageTemperatureUncertainty:', sum([1 for v in
→avg_temp_unc if v == '']))

```

Missing values in AverageTemperature: 0

Missing values in AverageTemperatureUncertainty: 0

Now that each missing values has been filled, we can save the dataset to a new file. You can find it at the following URL and use it to validate your results:

https://raw.githubusercontent.com/dbdmg/data-science-lab/master/datasets/GLT_filtered_filled.csv

```

[8]: with open('GLT_filtered_filled.csv', 'w') as fp:
      header = ','.join(col_names)

```

```

fp.write(f'{header}\n')
for i in range(rows_c):
    cols = []
    for j in range(len(dataset)):
        cols.append(str(dataset[j][i]))
    cols = ','.join(cols)
    fp.write(f'{cols}\n')

```

Even if this solution works, it is tailored to the way the input file is organized. What if the sorting keys were inverted, i.e. data were sorted by Date and then by City name? Working with positional indices this way would have been much more complex. Version B overcomes this eventual problem.

Version B The main idea is to work on measurements for each city separately. The data structure that helps here is the dictionary (this is true whenever you need to store and access quickly to something by any key value).

So, let's extract the distinct cities and count them.

```

[9]: cities = set(dataset[3])
print('Number of distinct cities:', len(cities))

```

Number of distinct cities: 100

For each city extract now its associated measurements.

```

[10]: city_avg_temp = {}
city_avg_temp_unc = {}

for city in cities:
    idxs = [i for i, c in enumerate(dataset[3]) if c == city] # extract the
    →indices
    city_avg_temp[city] = [dataset[1][i] for i in idxs]
    city_avg_temp_unc[city] = [dataset[2][i] for i in idxs]

```

Consequently, a modified version of the function `fill_gaps` is required now.

```

[11]: def fill_gaps(data):
    right_i = 0
    right_v = 0

    for i, value in enumerate(data):
        left_v = data[i-1] if i != 0 else 0

        # reuse the right_v value, useful when there are multiple consecutive
        →missing values
        if i < right_i:
            data[i] = (left_v + right_v) / 2
            continue

        if value == '':
            try:

```

```

        # use a generator to search for the first occurrence
        right_i, right_v = next((idx+i+1, float(v)) for idx, v in
→enumerate(data[i+1:]) if v != '')
        except StopIteration: # fired when the generator has no items left
→to iterate on
            right_i = len(data)
            right_v = 0
            data[i] = (left_v + right_v) / 2
        else:
            data[i] = float(data[i]) # parse to float all present values

```

We can now test again the function against a toy example and then apply it to our real dataset.

```

[12]: l = ['8', '', '3', '', '15', '', '']

print('Original list:', l)
fill_gaps(l)
print('Filled list:', l)

```

Original list: ['8', '', '3', '', '15', '', '']
 Filled list: [8.0, 5.5, 3.0, 9.0, 15.0, 7.5, 3.75]

```

[13]: for city in cities:
        fill_gaps(city_avg_temp[city])
        fill_gaps(city_avg_temp_unc[city])

```

As you can see, the function itself is more compact and uses a bunch of fundamental Python operators (you are likely going to read about this online as being more *pythonic*) to search the first following non-empty value: `next`, the generator `(i, float(v)) for i, v in enumerate(data[i+1:]) if v != ''` and the exception `StopIteration`. Moreover, it does not relies on any specific order of the data in the input file, since measurements are associated to cites by means of dictionaries.

Nevertheless, each value of both the dictionaries is a copy of the list from set dataset. Hence the memory required is at least twice the one by Version A. Also, the original structure of the dataset has not been changed (`dataset[1]` and `dataset[2]` still contain missing values), since we have worked on different objects, which is not desirable in certain cases. Major takeaway here: everything comes at a cost, you need to choice where and what to pay.

0.1.3 Exercise 1.3

Here we can use the dictionaries from Version B.

```

[14]: def print_hottest_cooltest(city, N, city_avg_temp):
        srtd = sorted(city_avg_temp[city], reverse=True)
        print(f'The top {N} hottest measurements taken in {city} are:', srtd[:N])
        print(f'The top {N} coolest measurements taken in {city} are:', srtd[-1:
→-(N+1):-1])

```

A note on the second slice. The notation `[-1:-(N+1):-1]` stands for “take items”: * from the last one `(-1)` * to the Nth-last one `-(N+1)` * with backward steps `(-1)`

```
[15]: print_hottest_coolest('Rome', 5, city_avg_temp)
```

The top 5 hottest measurements taken in Rome are: [25.951, 24.998, 24.873, 24.730999999999998, 24.48]

The top 5 coolest measurements taken in Rome are: [-1.4410000000000005, -1.3039999999999994, -1.0189999999999997, -0.871, -0.7829999999999999]

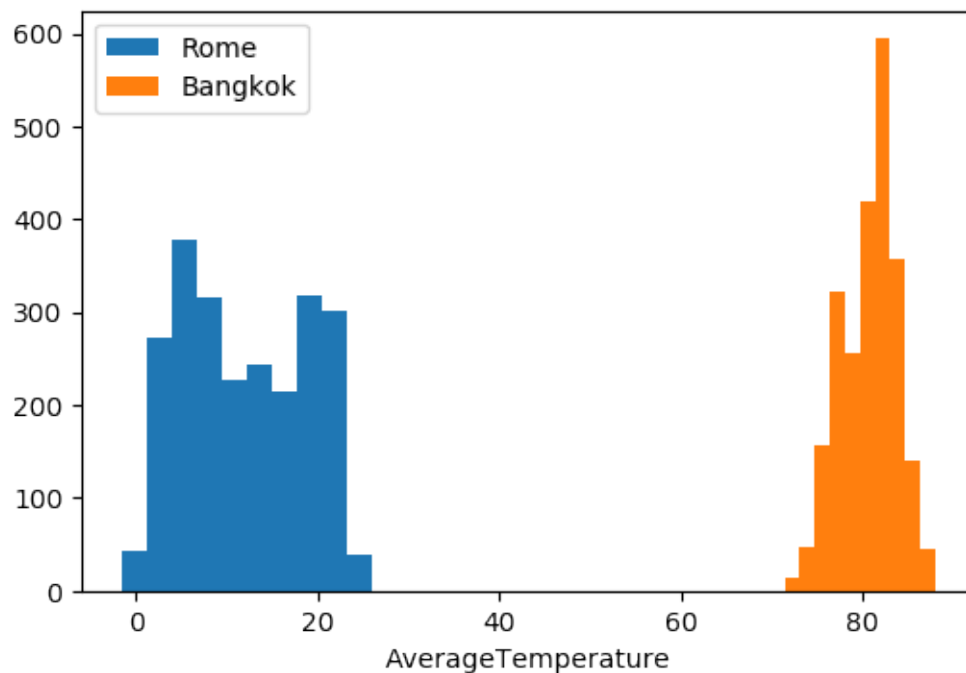
0.1.4 Exercise 1.4

We will use the matplotlib plotting function to inspect the temperature distributions in Rome and Bangkok.

```
[16]: import matplotlib.pyplot as plt
      %matplotlib inline
      plt.rcParams['figure.dpi'] = 100

      for city in ['Rome', 'Bangkok']:
          plt.hist(city_avg_temp[city], label=city)
      plt.legend()
      plt.xlabel('AverageTemperature')
```

```
[16]: Text(0.5, 0, 'AverageTemperature')
```



What we can see from the figure above is that Rome and Bangkok have had quite different temperatures. Looking at the histograms, it is evident the two cities have different weather conditions during the year.

Rome as a variable climate in the sense that temperatures vary almost uniformly between 0 and 30. Since this information is not present in the dataset, one can assume these values represent degrees Celsius. Bangkok, on the other end, have had a less variable weather, with land temperatures almost near to mean value: its temperature distribution looks like a gaussian one. Indeed, Bangkok's average land temperature is way higher than Rome's:

```
[17]: import numpy as np

print(f'Rome average temperature: {np.mean(city_avg_temp["Rome"]):.2f},  
      ↳Standard deviation: {np.std(city_avg_temp["Rome"]):.2f}')
print(f'Bangkok average temperature: {np.mean(city_avg_temp["Bangkok"]):.2f},  
      ↳Standard deviation: {np.std(city_avg_temp["Bangkok"]):.2f}')
```

Rome average temperature: 12.01, Standard deviation: 6.69

Bangkok average temperature: 80.73, Standard deviation: 3.04

An almost-constant 80 degrees Celsius average temperature would have made Bangkok not a nice place to live in. At least for human beings. This can lead us to think that Bangkok's sensors provide data in degrees Fahrenheit. Let's see how would this data look like in their Celsius counterparts.

0.1.5 Exercise 1.5

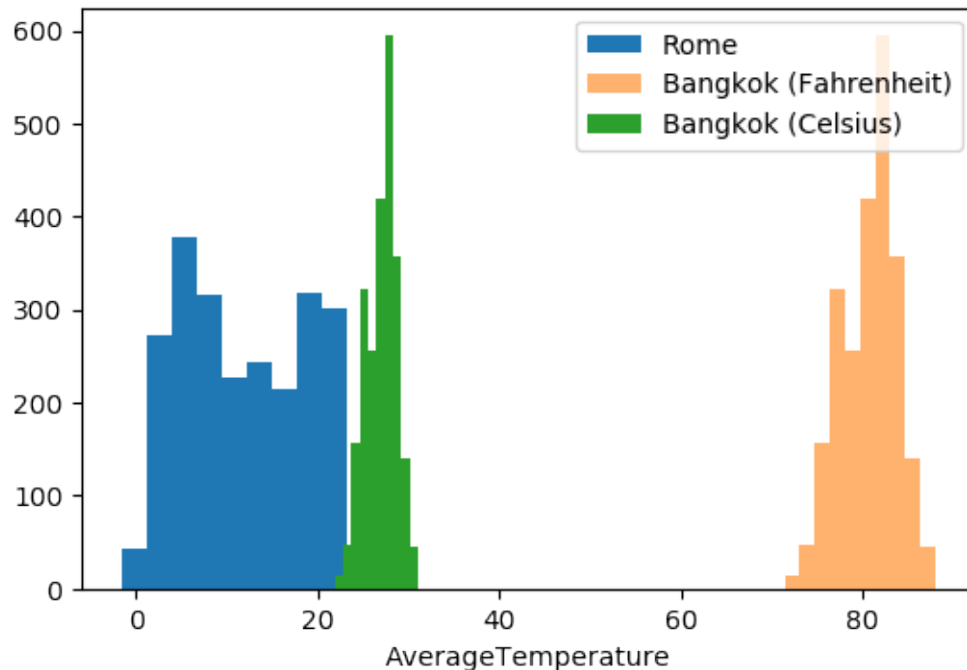
Keeping in mind that $T_F = 1.8 \cdot T_C + 32$, we can define a function to obtain T_C from T_F .

```
[18]: def fah2cel(deg_fah):
      return (deg_fah - 32) / 1.8
```

Then, apply it to Bangkok's data and plot the results.

```
[19]: bang_celsius = [fah2cel(t) for t in city_avg_temp['Bangkok']]
plt.rcParams['figure.dpi'] = 100
plt.hist(city_avg_temp['Rome'], label='Rome')
plt.hist(city_avg_temp['Bangkok'], label='Bangkok (Fahrenheit)', alpha=0.6)
plt.hist(bang_celsius, label='Bangkok (Celsius)')
plt.legend()
plt.xlabel('AverageTemperature')
```

```
[19]: Text(0.5, 0, 'AverageTemperature')
```



```
[20]: print(f'Rome average temperature: {np.mean(city_avg_temp["Rome"]):.2f},  
        ↳Standard deviation: {np.std(city_avg_temp["Rome"]):.2f}')  
print(f'Bangkok average temperature: {np.mean(bang_celsius):.2f}, Standard  
        ↳deviation: {np.std(bang_celsius):.2f}')
```

Rome average temperature: 12.01, Standard deviation: 6.69

Bangkok average temperature: 27.07, Standard deviation: 1.69

Bangkok has been pretty hotter though. A thing to consider if your are planning to move there.

0.2 IMDb reviews

Exercise 2.1

Let's begin loading the dataset as a list of lists. With a quick inspection, you can see that the file contains an header in the first line that tells you that rows contain two fields: review,label.

```
[21]: from collections import Counter  
  
reviews, labels = [], []  
# use the UTF-8 encoding to read the file  
with open('aclimdb_reviews_train.txt', encoding='utf-8') as fp:  
    reader = csv.reader(fp)  
    next(reader) #skip the header  
    for row in reader:  
        reviews.append(row[0])  
        labels.append(row[1])
```



```
[22]: print("Number of reviews in the dataset:", len(reviews))
      print("Number of 1's and 0's:", [(k, v) for k, v in Counter(labels).items()])
```

Number of reviews in the dataset: 25000

Number of 1's and 0's: [('1', 12500), ('0', 12500)]

0.2.1 Exercise 2.2

We can now define and apply the tokenization function provided in the text.

```
[23]: import string

def tokenize(docs):
    """Compute the tokens for each document.

    Input: a list of strings. Each item is a document to tokenize.
    Output: a list of lists. Each item is a list containing the tokens of the
    →relative document.
    """
    tokens = []
    for doc in docs:
        for punct in string.punctuation:
            doc = doc.replace(punct, " ")
        split_doc = [token.lower() for token in doc.split(" ") if token]
        tokens.append(split_doc)
    return tokens

token_list = tokenize(reviews)
```

token_list is now a list of lists. The *i*th item of the outer list is a list containing all the words found in the *i*th review. Notice that here duplicates can be present. For example:

```
[24]: print(token_list[0])
```

```
['for', 'a', 'movie', 'that', 'gets', 'no', 'respect', 'there', 'sure', 'are',
'a', 'lot', 'of', 'memorable', 'quotes', 'listed', 'for', 'this', 'gem',
' imagine', 'a', 'movie', 'where', 'joe', 'piscopo', 'is', 'actually', 'funny',
' maureen', 'stapleton', 'is', 'a', 'scene', 'stealer', 'the', 'moroni',
' character', 'is', 'an', 'absolute', 'scream', 'watch', 'for', 'alan', 'the',
' skipper', 'hale', 'jr', 'as', 'a', 'police', 'sgt']
```

0.2.2 Exercise 2.3

At this point we need to compute the *term-frequency* (TF) of each token (read *word* or *term*) within its document (read *review*). As usual, you can define a function for that.

```
[25]: def compute_TF(token_list):
      TF = []

      for document in token_list:
```

```

    tf = {}
    for token in document:
        tf[token] = tf.get(token, 0) + 1
    TF.append(tf)
    return TF

TF_list = compute_TF(token_list)

```

TF_list is now a list of dictionaries. The *ith* item of the outer list is a dictionary having the distinct tokens as keys and, for each token, the number of occurrences in the *ith* review as values. For example:

```
[26]: print(TF_list[0])
```

```

{'for': 3, 'a': 5, 'movie': 2, 'that': 1, 'gets': 1, 'no': 1, 'respect': 1,
'there': 1, 'sure': 1, 'are': 1, 'lot': 1, 'of': 1, 'memorable': 1, 'quotes': 1,
'listed': 1, 'this': 1, 'gem': 1, 'imagine': 1, 'where': 1, 'joe': 1, 'piscopo':
1, 'is': 3, 'actually': 1, 'funny': 1, 'maureen': 1, 'stapleton': 1, 'scene': 1,
'stealer': 1, 'the': 2, 'moroni': 1, 'character': 1, 'an': 1, 'absolute': 1,
'scream': 1, 'watch': 1, 'alan': 1, 'skipper': 1, 'hale': 1, 'jr': 1, 'as': 1,
'police': 1, 'sgt': 1}

```

Exercise 2.4

Now that we have the TF for each token in each document, we can compute the *inverse-term-frequency* (IDF). This number is assigned to each distinct token found in the whole collection of documents and inversely weights its presence among the documents, i.e.: * tokens that appear in only one document will have $IDF_t = \log N$ * tokens that appear in every document will have $IDF_t = 0$.

```

[27]: import math

def compute_IDF(TF_list):
    DF = {}
    N = len(TF_list)

    #~compute the document-frequency (DF), i.e. the number of documents in
    ~which each token appears at least once
    for review_tf in TF_list:
        for token, token_tf in review_tf.items():
            DF[token] = DF.get(token, 0) + 1
    # compute the actual IDF
    return {token: math.log(N / df) for token, df in DF.items()}

IDF_dict = compute_IDF(TF_list)

```

IDF_dict is now a dictionary. It has tokens as keys and their IDF as values.

We can now sort its keys by their IDF values in ascending order and print some of them.

```

[28]: sorted_view = sorted(IDF_dict.items(), key=lambda item: item[1])
sorted_view[:20]

```

```
[28]: [('the', 0.008314469604085238),
      ('a', 0.03351541933781697),
      ('and', 0.03401190259170586),
      ('of', 0.05226218466281087),
      ('to', 0.06293979977387414),
      ('this', 0.09924591465797242),
      ('is', 0.1086102347240488),
      ('it', 0.11536595914077863),
      ('in', 0.12606221366364628),
      ('that', 0.20722099077039452),
      ('i', 0.22800535073111738),
      ('s', 0.32335070173124136),
      ('but', 0.3296714147240428),
      ('for', 0.33502528396230163),
      ('with', 0.35861969087665957),
      ('was', 0.43602741369080433),
      ('as', 0.4389391649658812),
      ('on', 0.46451472090274043),
      ('movie', 0.4906962524708249),
      ('t', 0.5056390970786907)]
```

The words with the lowest IDF values show up in the majority of documents. Looking at them, you can notice that they are mainly articles, conjunctions, pronouns or even single letters. These words, which are commonly known as *stopwords*, are typically removed in the majority of text processing tasks. But why?

Labels, sentiment and similarity Your dataset contains a set of labels associated to review, each expressing whether a review is positive or not about the movie. This is known as the *sentiment* (e.g. which feeling is the user expressing about the movie? Is she/he complaining about something? Is there enthusiasm in her/his tone of voice?). Now, in order to exploit this information, we would like to understand which *elements* are shared between reviews with the same sentiment.

This, in turn, requires an instrument to estimate, quantitatively, how *similar* two or more texts are. The simplest idea would be that texts expressing the same sentiment do it with similar sets of words. The other way around, *different* texts should present different words. Under this assumption, the IDF comes to the aid here. The IDF (and so the TF-IDF will) value can be seen as a discriminative factor. The lower the value, the higher is the number of documents in which the relative word appears. In principle, since it is shared between many documents, both positive and negative, it would hard to discriminate based on its presence. It is said the the word carries a low discriminative significance.

This is evidently true for stopwords and that is why their usually set aside in textual analytics. Notice that, as it would be likely to happen in your dataset, also the word *movie* has a low IDF value and, in turn, a low discriminative significance.

0.2.3 Exercise 2.5

The final step requires the computation of the *term-frequency inverse-term-frequency* (TF-IDF). It is the effective weighting scheme that can be used to compute the similarity between two documents. As usual, let's define a function for that:

```
[29]: def compute_TFIDF(TF_list, IDF_dict):
    TFIDF = []

    for d in TF_list:
        tfidf = {}
        for t, t_tf in d.items():
            tfidf[t] = t_tf * IDF_dict[t]
        TFIDF.append(tfidf)
    return TFIDF

tf_idf = compute_TFIDF(TF_list, IDF_dict)
```

tf_idf is now a list of dictionaries. The *ith* item of the outer list is a dictionary having the distinct tokens as keys and, for each token, its TF-IDF weight. For example:

```
[30]: print(tf_idf[0])
```

```
{'for': 1.005075851886905, 'a': 0.16757709668908488, 'movie':
0.9813925049416498, 'that': 0.20722099077039452, 'gets': 2.257229391273248,
'no': 1.1141321003261466, 'respect': 3.9845936982629815, 'there':
0.837387134278689, 'sure': 2.3530366364901436, 'are': 0.5868431101899066, 'lot':
2.0319474551515233, 'of': 0.05226218466281087, 'memorable': 3.6936910111111585,
'quotes': 5.5940316106970815, 'listed': 5.339139361068292, 'this':
0.09924591465797242, 'gem': 4.291820366787733, 'imagine': 3.587045148232668,
'where': 1.655900786844441, 'joe': 4.137669686960474, 'piscopo':
7.418580902748128, 'is': 0.3258307041721464, 'actually': 1.982532640511814,
'funny': 2.0743346043116913, 'maureen': 6.437751649736401, 'stapleton':
7.561681746388801, 'scene': 1.8767946184246356, 'stealer': 7.487573774235079,
'the': 0.016628939208170476, 'moroni': 8.740336742730447, 'character':
1.641547966352334, 'an': 0.7166205367455873, 'absolute': 4.315490110873637,
'scream': 4.706096104578052, 'watch': 1.5199629060064976, 'alan':
4.625372893305611, 'skipper': 7.929406526514119, 'hale': 6.515713191206113,
'jr': 4.5932416151228175, 'as': 0.4389391649658812, 'police': 3.460947386067929,
'sgt': 6.4630694577206915}
```

By using TF-IDF, we can have insights on the discriminative significance of the single word in a single document. Let's look at the most significant ones in the first document:

```
[31]: print(sorted(tf_idf[0].items(), key=lambda item: item[1], reverse=True)[:10])
```

```
[('moroni', 8.740336742730447), ('skipper', 7.929406526514119), ('stapleton',
7.561681746388801), ('stealer', 7.487573774235079), ('piscopo',
7.418580902748128), ('hale', 6.515713191206113), ('sgt', 6.4630694577206915),
('maureen', 6.437751649736401), ('quotes', 5.5940316106970815), ('listed',
5.339139361068292)]
```

0.2.4 Exercise 2.6

Now that we have the TF-IDF we can measure how similar (or *close*) are two documents. This is made possible by the fact that now each text has a numerical representation. More specifically,