Lab Session #07

Introduction

Welcome to Lab #7. The goal in this lab is to become familiar with the *scitkit-learn* library for Python, in particular how to use it for creating tf-idf vectors and other basic machine learning tasks.You can find a number of *"cheat sheets"* for working with Python at DataCamp; in particular, there is a nice one for working with *scikit-learn* as well.

Follow-up Lab #6

As always, please ask in the Moodle forum if something is unclear or doesn't work as expected!

Solution Task #1 (Cosine Similarity)

Well, the complete code to run this was basically given:

```
import numpy as np
movies = np.array([
[4, 8, 6, 3, 0, 0],
[0, 5, 0, 8, 5, 0],
[1, 4, 0, 3, 0, 10]])

print(movies)

from sklearn.metrics.pairwise import cosine_similarity
similarity_scores = cosine_similarity(movies)
print(similarity_scores)
```

Solution Task #2 (Movie Recommender System)

Here's a complete example program that reads the MovieLens dataset (hard-coded as stored in `"ml-latest-small/"`), creates the tag-based vectors and computes the similarity matrix. Then, it prints out the top-5 recommendations for a movie (again, hard-coded for `'Toy Story (1995)'`). Feel free to make it more generic and improve upon it!

Task #1: TF-IDF Vectors

In the lecture, we discussed how to represent natural language documents in form of tf-idf vectors. Scikit-learn also has built-in support for creating tf-idf vectors, the TfidfVectorizer. Let's try this out on some documents:

```
documents = (
"The sky is blue",
"The sun is bright",
"The sun in the sky is bright",
"We can see the shining sun, the bright sun")
```

You can now use a `TfidfVectorizer` similar to the `CountVectorizer` from the previous lab:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)
```

Print out the `tfidf_matrix.toarray()`, the `tfidf_matrix.shape`, as well as the `idf_` weights for each word:

```
import pandas as pd
df_idf = pd.DataFrame(tfidf_vectorizer.idf_,
index=tfidf_vectorizer.get_feature_names(),columns=["idf_weights"])
```

You could now compute a *cosine similarity matrix* on the documents, similar to the recommender engine exercise:

```
from sklearn.metrics.pairwise import cosine_similarity
similarity_scores = cosine_similarity(tfidf_matrix)
```

However, what we'll do next is to use the idea of cosine similarity to implement a basic search engine: Given a new document,

```
search_doc = "The dog is bright"
```

you want to find the best matching (most similar) documents in your index. You first have to vectorize the new document:

```
search_tfidf_vector = ...
```

Be careful: you can use the same `tfidf_vectorizer`, but you have to make sure you use the same vocabulary (feature names). Read the documentation and make sure you understand the difference between the methods `fit_transform` and `transform`. Check your features using `tfidf_vectorizer.get_feature_names()`.

After you successfully converted the search document, you can compute the similarity scores with the other documents:

```
similar_docs = cosine_similarity(...)
```

Now you can sort the documents by similarity and print them out by rank, search engine-style.

For more details, make sure to read the documentation on text feature extraction. In particular, if you want to check the tf-idf calculations we made on the worksheet, make

sure you check the various parameter settings for the vectorizer as explained in the documentation on tf-idf term weighting.

Task #2: Re-implement Google

We'll extend the ideas from the previous task to a mini-search engine, where you let the user enter some keywords (just like a Google search box), search your documents, and return the top-*n* matching results.

To get some more data into your program, we'll use one of the datasets that come with *scikit-learn*: The 20newsgroups dataset. You probably don't know about these newsgroups – that's ok, they are even older than the Web: they come from Usenet, which dates back to 1979. This dataset is often used for various natural language processing (NLP) experiments. Here's an example post, from the newsgroup `comp.sys.mac.hardware`:

```
Newsgroup: comp.sys.mac.hardware
document_id: 52139
From: rriegsec@iris.mbvlab.wpafb.af.mil (Randy Riegsecker)
Subject: Third party monitor on IIsi

So what's the deal with the PDS slot in the IIsi?

I recently purchased a Mac IIsi.  I want to add a non-Apple monitor to the
system.  I was told that you could buy a 90 degree angled PDS to NuBus adaptor
card so you can fit a standard NuBus card into the computer.

Am I mistaken or do have to buy a PDS monitor card specifically for the IIsi?
I've seen the PDS monitor cards for the si, but they seem expensive, and I'm
not exactly made of money.

Any ideas?  Help.  Clue me in!
```

Scikit-learn has some utility functions that will download the dataset for you the first time you use it (by default it will go into `~/scikit_learn_data/`). Re-running the program will then make use of the cached data:

```
from sklearn.datasets import fetch_20newsgroups
newsgroups_train = fetch_20newsgroups()
```

You can print out the topics using:

```
from pprint import pprint
pprint(list(newsgroups_train.target_names))
```

Ok, now it's your turn: First, create tfidf-vectors from this dataset, similar to last week:

```
tfidf_matrix = tfidf_vectorizer.fit_transform(newsgroups_train.data)
```

Now you can use your code from the previous week to (1) transform a search query into a tf-idf vector; and (2) use `cosine_similarity` between the query vector and the document vectors to find the best search results for the query. **Note:** for development & testing, you should probably only load a subset of the newsgroups, like here:

```
cats = ['alt.atheism', 'sci.space']
```

```
newsgroups_train = fetch_20newsgroups(subset='train', categories=cats)
```
Of course, if you ever have to build a real-life, scalable search system, better use a library like Apache Lucene and a search server like Elasticsearch or Solr (they all offer Python bindings as well).

Task #3: k-Means Clustering

Scikit-learn also provides an implementation for k-Means clustering. Let's try to cluster the documents from Task #1:

```
documents = (
"The sky is blue",
"The sun is bright",
"The sun in the sky is bright",
"We can see the shining sun, the bright sun")
```
To be able to cluster them, we will need to convert each document into a feature vector. Since we are dealing with natural language documents, tf-idf vectors are a better representation than simple binary or count vectors:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf_vectorizer = TfidfVectorizer()
tfidf_matrix = tfidf_vectorizer.fit_transform(documents)
```
Now you can use k-Means to create clusters. Remember that you have to provide k (the number of clusters) as input: Here we are using k=3, so three clusters:

```
kmeans = KMeans(n_clusters=3)
kmeans.fit(tfidf_matrix)
```
To see which document now belongs to which clusters, you can print:

```
print(kmeans.labels_)
```
Experiment with different documents and different numbers of clusters. For example, try to cluster the documents from the 20newsgroups dataset into different sized clusters. Remember that k-Means uses random starting positions for the centroids, so different runs will most likely give you different clusters as result.

That's all for this lab!