Lab Session #06

Introduction

Welcome to Lab #06. This lab introduces more foundational concepts of machine learning (ML), a pivotal component of intelligent systems enabling computers to learn from data. ML techniques allow for automatic pattern recognition, predictive modeling, and decision-making in diverse applications such as natural language processing, computer vision, and predictive analytics. By applying ML, systems can adapt and improve over time, enhancing user experience, optimizing operations, and solving complex problems without explicit programming for every scenario.

The core of our practical exercises involves the *scikit-learn* library for Python, renowned for its comprehensive collection of ML algorithms (see the previous lab for installation instructions). Scikit-learn simplifies the implementation of numerous ML models, including regression, classification, clustering, and dimensionality reduction. Its design principle focuses on providing a consistent interface for ML models, making it easier to experiment with different approaches. Furthermore, scikit-learn integrates seamlessly with other Python libraries, such as NumPy and SciPy, for mathematical operations, and Matplotlib for data visualization, enabling a holistic approach to ML model development and evaluation.

For supplementary resources, DataCamp offers valuable cheat sheets, including one for *scikit-learn*. These cheat sheets are a quick reference for scikit-learn's API, helping to streamline the coding process.

Follow-up Lab #05

Here are some solution notes for the previous lab. 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 Document Vectors

In the lecture, we explored the representation of natural language documents as tf-idf vectors, a method that quantifies the importance of words within documents in a corpus. TF-IDF stands for Term Frequency-Inverse Document Frequency, combining the frequency of a term in a specific document (TF) with the inverse of the frequency of that term across the entire document set (IDF). This results in a numerical representation that highlights words that are unique and informative to a document within the context of a given corpus.

Building on the concept of cosine similarity from the previous lab, tf-idf vectors enable us to measure the similarity between documents more effectively. While cosine similarity allowed us to compare the orientation of two documents in a high-dimensional space, tf-idf vectors refine this comparison by weighting terms, thereby improving the accuracy of similarity measurements. Scikit-learn offers built-in support for creating tf-idf vectors through the TfidfVectorizer, making it straightforward to apply these concepts to our dataset. Let's apply this tool on some documents and explore the improved document similarity analysis it facilitates:

```
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_out(),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_out().

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 tf-idf-vectors from this dataset, similar to above:

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

Now you can use your code from Task #1 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.

Remember, when transforming your search query into a tf-idf vector, ensure you use the `transform` method of the already fitted `tfidf_vectorizer` from the previous step. This maintains the same vocabulary (feature set) for your query, allowing for accurate similarity comparisons with the document vectors.

Note: Working with a subset of the data is highly recommended for development and testing phases. This approach not only speeds up the iteration cycle but also helps in managing computational resources effectively, especially when dealing with large volumes of data. You can load a subset of the data in the following way:

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

newsgroups_train = fetch_20newsgroups(subset='train', categories=cats)
```

Building Scalable Document Search Systems

Of course, if you ever have to build a real-life, scalable search system, you should use a library like Apache Lucene, a high-performance, full-featured text search engine library written entirely in Java. It's the backbone of many search platforms, providing the indexing and search technology but requiring you to build the interface and integration yourself. Lucene is available under the Apache License 2.0, offering flexibility for both commercial and open-source projects. For a more comprehensive solution, you might look into a search server like Elasticsearch or Solr.

Elasticsearch, now provided under the commercial "Elastic License", is a distributed, RESTful search and analytics engine known for its simple REST APIs, distributed nature, speed, and scalability, addressing a wide variety of use cases.

Solr, a popular, blazing-fast open source enterprise search platform built on Apache Lucene, offers powerful full-text search, hit highlighting, faceted search, dynamic clustering, database integration, and rich document handling. Distributed under the Apache License 2.0, Solr is an excellent choice for projects of all sizes. Both Elasticsearch and Solr provide Python bindings, making them accessible for a wide range of projects and applications.

In addition to Apache Lucene, Elasticsearch, and Solr, other noteworthy technologies in the domain of scalable search systems include MeiliSearch, a fast, open-source search engine that prioritizes simplicity and ease of use, and Typesense, which is designed for typo tolerance and offers an out-of-the-box search experience. On the cloud front, services like Amazon CloudSearch, a managed service in the AWS ecosystem, and Azure Cognitive Search, offering rich search capabilities including AI-enriched content understanding, provide scalable, fully-managed search solutions.

Task #3: k-Means Clustering

Clustering is a type of unsupervised learning that groups data points into several clusters based on their similarity. One of the most popular clustering algorithms is k-means, which partitions data into k distinct clusters based on distance metrics, typically aiming to minimize the variance within each cluster. In the context of intelligent systems, k-means can be used for document clustering, serving applications such as organizing large sets of documents, information retrieval, and enhancing the user experience by grouping similar content. Document clustering use cases range from news aggregation, where articles about similar topics are grouped, to customer support systems, where inquiries with similar themes are clustered for efficient response. By applying k-means, we can discover inherent groupings within our documents, aiding in the organization and retrieval of information.

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)
```

Choosing the right number of clusters (*k*) is crucial for the k-Means algorithm's performance and the meaningfulness of the clusters. There's no one-size-fits-all answer for selecting *k*, but methods like the elbow method can help determine a suitable number by plotting the within-cluster sum of squares (WCSS) against the number of clusters and looking for the 'elbow point' where the rate of decrease sharply changes.

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.

Keep in mind that k-Means clustering involves random initialization of cluster centroids, which can affect the final clustering outcome. This randomness can lead to different results upon different algorithm runs, especially for complex datasets. To mitigate this, scikit-learn's implementation allows setting a `random_state` to ensure reproducibility, or using the `n_init` parameter to run the algorithm multiple times with different centroid seeds and choosing the best output based on inertia.

Visualization of Clustering Results

Visualizing the results of the k-means clustering can help us understand the distribution of documents across the clusters. Since our data is in a high-dimensional space due to the TF-IDF vectorization, we'll use Principal Component Analysis (PCA) to reduce the dimensionality of our data to two dimensions for easy visualization. This step simplifies the visualization while retaining the essence of the clusters.

Here's how you can perform PCA and visualize the clusters:

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

# Assuming 'tfidf_matrix' is the TF-IDF matrix for your documents
# and 'kmeans' is your fitted k-means clustering model

# Reduce dimensions
pca = PCA(n_components=2)
reduced_data = pca.fit_transform(tfidf_matrix.toarray())

# Plot each cluster
plt.figure(figsize=(10, 6))
colors = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'orange', 'purple', 'brown']
for i in range(len(colors)):
    x = reduced_data[:, 0][kmeans.labels_ == i]
    y = reduced_data[:, 1][kmeans.labels_ == i]
```

```
    plt.scatter(x, y, c=colors[i], label=f'Cluster {i}')
plt.title('Document Clusters after PCA Reduction')
plt.xlabel('PCA1')
plt.ylabel('PCA2')
plt.legend()
plt.show()
```

This scatter plot shows the documents reduced to two principal components, colored by their assigned cluster. This visualization allows us to observe the natural groupings within our document set and can provide insights into the effectiveness of our clustering approach.

For a more interactive visualization experience, consider using the Plotly library. Plotly enables the creation of dynamic, interactive plots that allow you to hover over points to see additional details or to zoom in and out to better understand the distribution of clusters. By replacing the Matplotlib code with Plotly's syntax, you can enhance the visualization with interactive capabilities, making it easier to explore and interpret the clustering results in depth.

Conclusions

In this lab, you've delved into key machine learning techniques for processing and analyzing text data. You learned to represent natural language documents using tf-idf vectors, a crucial step for many text analysis applications. Through k-means clustering, you gained practical experience in segmenting documents into clusters, revealing underlying patterns and relationships.

These techniques have broad applications across various fields. For instance, document clustering powers the organizational logic in content management systems, helping categorize and retrieve documents efficiently. In customer service, analyzing and grouping customer feedback or inquiries can enhance response strategies and product insights. Moreover, in the field of legal and medical document analysis, clustering helps manage vast repositories of texts by grouping documents with similar topics, significantly aiding in research and review processes.

The skills you've developed in this lab lay the groundwork for further exploration in machine learning and its applications in intelligent systems. By understanding these foundational techniques, you're better prepared to tackle more complex challenges and contribute to projects that leverage machine learning to process and analyze data effectively, enhancing decision-making and user experience across various sectors.

That's all for this lab!