QBUS6850: Tutorial 12 – Matrix Factorization

Objectives

- To learn how to do NMF and SVD in Python;
- To learn the application of NMF and SVD.

1. NMF (or NNMF)

It is important to understand that NMF does not produce a unique solution or recover the original generative process exactly. This is because there exists a large number of basis vector and coefficient pairs that can reconstruct the data.

Below a synthetic situation is used to illustrate this point.

However please keep in mind that you can influence the solution towards a unique solution. For example, in spectral signal recovery we can apply temporal smoothness regularisation or sparsity on the coefficients.

The notations:

X: shape (n_samples, n_features). Data matrix to be decomposed

W: shape (n_samples, n_components). Basis matrix

H: shape (n_components, n_features). Coefficients matrix

Step 1: Create a synthetic matrix and a NMF model:

```
import numpy as np
import pandas as pd
from sklearn.decomposition import NMF

# 200 x 100  = 200 x 5 by 5 x 100
n = 200
m = 100
r = 5

# Low-rank basis (orthogonal columns)
basis_w = np.dot(np.random.rand(m, r), np.random.rand(r, r))

# Random coefficients
coefs_h = np.random.rand(r, n)

V = np.dot(basis_w, coefs_h)
model = NMF(n_components=r, init='random', random_state=0)
```

Step 2: Calling fit_transform will calculate the NMF decomposition and return the basis W and coefficients H

```
W = model.fit_transform(V)
print(W.shape)

H = model.components_
print(H.shape)
```

Step 3: Let's check how accurately we can reconstruct the original data. Increasing number of components will increase accuracy of reconstruction.

```
reco = np.dot(W, H)
diff = V - reco
print(np.linalg.norm(diff) / np.linalg.norm(V))
```

Then we can compare the true basis matrix and estimated basis matrix

```
#True one
pd.DataFrame(basis_w).head()
# Estimated
pd.DataFrame(W).head()
```

2. Market Movement Analysis

One of the powerful applications of NNMF is clustering via the underlying generative process.

This is in contrast to traditional clustering methods such as k-means. These traditional clustering methods rely on vector distance metrics. This usually leads to clustering of stocks that are of a similar price.

However, what we are usually more interested in is "which stocks move together?".

By performing NNMF we can produce a set of basis vectors and coefficients. The basis vectors can be roughly interpreted as unknown factors that affect the stock price. The coefficients of each stock are then treated as a similarity. In other words, if stocks have similar coefficient patterns then they will likely move together.

So, the process is to perform NNMF and then cluster stocks based on their coefficients.

In this example, we look at the 30 component stocks of the Dow Jones Industrial Average over 25 weeks.

Step 1: Prepare data to play with and plot the data

```
from sklearn.cluster import KMeans

np.random.seed(0)
```

```
dow_jones = pd.read_csv("dow_jones_index.data")

V = dow_jones.pivot(index = "stock", columns = "date", values = "close")

V = V.replace('[\$,]', '', regex=True).astype(float)
# V transpose has columns as stocks (examples) and rows as dates (features)
V= V.T
V.head()

import matplotlib.pyplot as plt

# Plot individual stock prices
fig1 = plt.figure()

plt.plot(V.values)
fig1
```

Step 2: Fit the NMF model can get the coefficient matrix.

```
model = NMF(n_components=5, init='random', random_state=0, max_iter=10000)
W = model.fit_transform(V)

print(W.shape)
H = model.components_

print(H.shape)
```

Step 3: Group stocks by their cluster assignment for interpretation.Or cluster stocks based on their coefficients.

```
# kmeans treats row as training example, so transpose H
kmeans = KMeans(n_clusters=7, random_state=0).fit(H.T)
# generate clustering labels for each training example
labels = kmeans.labels_

groups = [list() for i in range(7)]

for i in range(len(labels)):
    g = labels[i]
    groups[g].append(V.columns[i])

for i in range(len(groups)):
    print("Stock group {0}: ".format(i) + str(groups[i]))
```

3. Topic Modelling/Extraction

Given a collection of documents we can use NMF to automatically find words (features) which are most relevant to each topic (basis vector).

The number of topics is controlled by the number of basis vectors.

In this example, we will use the twenty newsgroup dataset to find 10 topics and the most important words in those topics.

Step 1: Import of data and set up

```
from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.datasets import fetch_20newsgroups



n_samples = 2000

n_features = 1000

n_components = 10



dataset = fetch_20newsgroups(shuffle=True, random_state=1,
remove=('headers', 'footers', 'quotes'))



data_samples = dataset.data[:n_samples]
```

Step 2: Build a tdidf representation that removes stop words and filters out words that occur in 95% of documents

```
tfidf_vectorizer = TfidfVectorizer(max_df=0.95, min_df=2,
max_features=n_features, stop_words='english')
tfidf = tfidf_vectorizer.fit_transform(data_samples)

# Transpose the data so that we have rows as features (1000) and columns
as samples (2000)
tfidf= tfidf.T
print(tfidf.shape)
```

Step 3: Each column of W represents one topic: 10 topics in total. And we can we display top 15 words in each topic.

```
nmf = NMF(n_components=n_components, init='random', random_state=0,
max_iter=10000)
W = nmf.fit_transform(tfidf)
W.shape
```

```
def print_top_words(model, feature_names, n_top_words):
    for topic_idx, topic in enumerate(W.T):
        message = "Topic #%d: " % topic_idx
        message += " ".join([feature_names[i]
                             for i in topic.argsort()[:-n_top_words - 1:-1]])
        print(message)


n_top_words = 15

tfidf_feature_names = tfidf_vectorizer.get_feature_names()
print_top_words(nmf, tfidf_feature_names, n_top_words)
```

4. **Face Feature Extraction.**

This example is from the lecture. We can extract image features for applications such as compression and feature engineering.

Step 1: load and pre-process the data

```
from sklearn.datasets import fetch_lfw_people

lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

# Images in the LFW dataset are 50*37 pixels in size.
# Therefore 1850 features in total.
# n_samples is the number of images

n_samples, a, b = lfw_people.images.shape
V = lfw_people.data
# Transpose the data so that we have rows as features (1850) and columns as samples (1288)
V= V.T
```

Display an example face (1st face of the input). Note here each column represents one face image.

```
fig = plt.figure()
plt.imshow(V[:,1].reshape((a, b)), cmap=plt.cm.gray)
fig
```

Step 2: Let's do a NNMF decomposition with 50 basis vectors and visualize the basic vectors.

```
model = NMF(n_components=50, init='random', random_state=0)

W = model.fit_transform(V)

H = model.components_

fig, axarr = plt.subplots(5, 10, figsize=(8, 6))
```

```
axarr_list = np.ravel(axarr)
for i in range(50):
    # each coolumn of W represent one basis vector
    axarr_list[i].imshow(W[:,i].reshape((a, b)), cmap=plt.cm.gray)
    axarr_list[i].axis('off')
fig
```

Step 3: Check the approximation WH (1st face of the approximation)

```
V_Approx= np.dot(W, H)

fig = plt.figure()
plt.imshow(V_Approx[:, 1].reshape((a, b)), cmap=plt.cm.gray)
fig
```

5. Singular Value Decomposition

SVD is a central part of these algorithms and is often the bottleneck. Below we illustrate the various types of SVD and their pros/cons.
There are four options:

- Full SVD
- Partial SVD (optional)
- Sparse SVD (optional)
- Approximate SVD (optional)

Step 1: Prepare some synthetic data

```
import scipy.sparse.linalg as sparse
import scipy as sp
import numpy as np
import time

n = 1000
m = 900
r = 5
A = np.dot(np.random.randn(n, r), np.random.randn(r, m))

print(A.shape)
print(np.linalg.matrix_rank(A))
```

Step 2: Full SVD
Full SVD decomposition generates full size matrices for U, S and V.
Full SVD, has approximately $O(n3)$ time complexity and does not scale well to large matrices or matrices with large rank.

```
tic = time.time()

U, s, V = sp.linalg.svd(A, full_matrices=True)

toc = time.time()
```

```
print("U: " + str(U.shape))
print("s: " + str(s.shape))
print("V: " + str(V.shape))


S = np.zeros((1000, 900))
S[:900, :900] = np.diag(s)
print("Running time: {0}s. Error: {1}".format(toc - tic,
np.linalg.norm(U.dot(S).dot(V) - A)))
```

Step 3: Impact of full_matrices parameter. When full_matrices=False, this is also called reduced SVD. Note the difference for the sizes of matrix U between full_matrices=True and full_matrices=False.

```
tic = time.time()

U, S, V = sp.linalg.svd(A, full_matrices=False)

toc = time.time()

print("U: " + str(U.shape))
print("S: " + str(S.shape))
print("V: " + str(V.shape))

print("Running time: {0}s. Error: {1}".format(toc - tic,
np.linalg.norm(U.dot(np.diag(S)).dot(V) - A)))
```

Step 4: Partial and Sparse SVD

A partial SVD is performing an SVD on only the top k singular values. Partial SVD is also known as truncated SVD or skinny SVD.

A sparse SVD is performing an SVD on a sparse matrix.

In scipy, the partial and sparse SVD function is shared. This function implements a different SVD solver that takes advantage of computing the SVD of a sparse input matrix OR computing a reduced number of singular values.

Notice that computing a partial SVD is significantly quicker and the error is still relatively small.

```
tic = time.time()
U, S, V = sparse.svds(A, k=r)
toc = time.time()

print("U: " + str(U.shape))
print("S: " + str(S.shape))
print("V: " + str(V.shape))

print("Running time: {0}s. Error: {1}".format(toc - tic,
np.linalg.norm(U.dot(np.diag(S)).dot(V) - A)))
```

Step 5: Approximate SVD
Approximate SVD is a technique that uses randomisation to quickly compute an approximate SVD result.
More details are available here:

- https://arxiv.org/pdf/0909.4061.pdf
- https://amath.colorado.edu/faculty/martinss/Talks/2015_02_chicago.pdf

Suffice to say that the approximate SVD scales nicely to huge matrices and results in a suprisingly accurate result. For massive scale machine learning it is the only solution.

```
def random_svd(A, k):
    omega = np.random.randn( A.shape[1], r )
    Y = np.dot(A, omega)
    Q = sp.linalg.orth(Y)
    B = np.dot(np.transpose(Q), A)
    Uhat, S, V = sp.linalg.svd(B, full_matrices=False)
    U = np.dot(Q, Uhat)

    return U, S, V

tic = time.time()
[U, S, V] = random_svd(A, k=5)
toc = time.time()

print("U: " + str(U.shape))
print("S: " + str(S.shape))
print("V: " + str(V.shape))

print("Running time: {0}s. Error: {1}".format(toc - tic,
np.linalg.norm(U.dot(np.diag(S)).dot(V) - A)))
```