Tutorial 2: Dimensionality Reduction

Jiawei Li

Playlist Continuation

You may have used the automatic playlist continuation feature on Spotify. As a matter of fact, Spotify even hosts a challenge to develop a system for the task of automatic playlist continuation, with a dataset of 1 million playlists consist of over 2 million unique tracks. Let us think of a simplified way to solve this challenge:

1. Each playlist is a vector of 2 million unique tracks;
2. We can find the most similar playlist to what the user is playing;
3. Suggest the song which the user has not listened to.

It may sound easy, but finding a similar vector with 2 million dimensions is almost impossible. We need a way to reduce the dimensionality of these vectors. Random projection is one simple way. For a $m \times n$ matrix A, it is a projection from $\mathbf{R}_n \to \mathbf{R}_m$ when applied to n-dimension x:

$$Ax = b$$

Johnson-Lindenstraus theorem simply proves that there is a theoretical guarantee that the errors will be smaller than some number for any random projection $\mathbf{R}_n \to \mathbf{R}_m$. If you are fine with this number of errors, you can go ahead with this projection.

Let us say, we are fine to project the playlist vector to 2 thousand dimensions. Then we need a random matrix which has a shape 2 thousand $\times$ 2 million. This is still a fairly difficult problem, many solutions have been proposed. For this tutorial, we do a simple version of projecting $\mathbf{R}_{200} \to \mathbf{R}_{10}$ using Normal Distribution. We first import `numpy`.

```
import numpy as np
```

We have a playlist vector x.

```
x = np.random.randint(0, 2, size=200)
print(f"x:\n{x}")
```

```
x:
[1 1 0 0 1 0 0 1 1 0 1 1 0 1 1 0 0 0 0 0 0 1 1 0 0 1 1 1 0 1 1 0 1 1 1 1 0
 1 1 1 0 1 1 0 1 0 0 0 0 1 0 0 1 1 1 0 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 1 0 1
 0 0 0 0 0 1 1 1 0 1 0 1 0 0 1 1 0 0 0 0 0 1 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 1 0 1 0 1 1 1 1 1 0 0 1 1 1 1 1 1
 0 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 0 0
 0 0 0 1 1 1 1 1 1 1 0 1 1 1 0]
```

And a random matrix of Normal Distribution with mean of 0 and standard deviation of 1.

```
A = np.random.normal(0, 1, size=(10, 200))
print(f"A:\n{A}")
```

```
A:
[[ 0.36941365  3.18916979 -0.71034571 ... -0.55438948  0.34899222
   0.41341859]
 [ 0.35590628  0.13201672  1.883686   ... -0.3259074  -0.04603383
   1.5831421 ]
 [ 0.3557712  -0.73150686  0.47653511 ... -0.19760859 -0.2729818
  -0.33752149]
 ...
 [-0.49219096 -0.4141803   0.8105228  ...  0.70235948  0.72470347
  -1.37312265]
 [ 0.96650547  0.13047957  0.43977846 ...  0.30581234  1.58497764
  -0.93659066]
 [-1.07530345 -2.15496836  1.28924924 ...  0.98394855  0.07784941
  -0.91005704]]
```

Multiply and done.

```
b = A@x
print(f"b:\n{b}")
```

```
b:
[  2.92710602   5.31449187 -12.56711333  10.6497657    0.04954915
 -19.75514449  -5.96359757   5.09996888  -8.64185828 -20.84229521]
```

On the exercise, you are asked to create a random matrix with specific probabilities. You can refer to `numpy.random.choice()`.

Further Reading

Python Data Science Handbook
Nearest Neighbors in Python with Annoy
Database-friendly random projections: Johnson-Lindenstrauss with binary coins
CS168 Lecture 4: Dimensionality Reduction

Mushroom Hunting

Now, let us be mushroom hunters who want to understand the underlying attributes of mushrooms. There is a great dataset prepared by researchers at University of Marburg. The research is published in Nature Scientific Reports.

Now, we import the libraries and read the mushroom dataset.

```
import altair as alt
import pandas as pd
import numpy as np
```

```
def calculate_mean(x):
    """
    Transform a string of "[lower, higher]" into its mean.
    """
    from ast import literal_eval

    try:
        return int(x)
    except ValueError:
        x = literal_eval(x)
        lower = int(x[0])
        higher = int(x[-1])
        mean = (lower + higher) / 2
        return mean
```

We first investigate the size of the mushrooms. In particular, the stem height and stem width.

```
mushroom = pd.read_csv("https://mushroom.mathematik.uni-marburg.de/files/PrimaryData/prima
mushroom["stem-height"] = mushroom["stem-height"].apply(calculate_mean)
mushroom["stem-width"] = mushroom["stem-width"].apply(calculate_mean)
mushroom.loc[:5, ["stem-height", "stem-width"]]
```

	stem-height	stem-width
0	17.5	17.5
1	8.0	15.0
2	11.0	15.0
3	11.0	17.5
4	11.0	15.0
5	6.0	12.5

These two attributes are correlated. Tall and wide stems mean that the mushrooms is chunky, and short and thin stems means that the mushrooms is probably tiny. These two dimensions can be reduced to one, which means that we can apply Principal Component Analysis (PCA).

```
alt.Chart(mushroom).mark_circle().encode(
    x="stem-height",
    y="stem-width",
    tooltip=["stem-height", "stem-width"],
)
```

```
alt.Chart(...)
```

Let us start applying PCA. We first centre the data.

```
# Select columns of "stem-height" and "stem-width"
mushroom = mushroom.loc[:, ["stem-height", "stem-width"]]
# Center the data so each feature has zero mean
mushroom = mushroom - mushroom.mean()
mushroom_chart = alt.Chart(mushroom).mark_circle().encode(
    x="stem-height",
    y="stem-width",
    tooltip=["stem-height", "stem-width"],
)
mushroom_chart
```

```
alt.Chart(...)
```

Then we reconstruct the data in row format, this is a purely arbitrary choice that follows most mathematical literature. If you know Linear Algebra, you know how to apply the same procedure on column format data.

```
# Consturct data in row format
D = mushroom.to_numpy().transpose()
# Compute covariance matrix
S = np.cov(D)
print(f"S:\n{S}")
```

```
S:
[[10.65907716 13.97140073]
 [13.97140073 97.21387283]]
```

```
# This is equivalent to D@D.T / dof
# Note that we divide the covariance by D.shape[-1] - 1 because
# we have D.shape[-1] data points in total and need minus 1 degree of freedom
S = D @ D.T / (D.shape[-1] - 1)
print(f"S:\n{S}")
```

```
S:
[[10.65907716 13.97140073]
 [13.97140073 97.21387283]]
```

```
# Compute eigenvalue and eigenvector
# Note that we use np.linalg.eigh() because S is
# a symmetric matix, np.linalg.eigh() gives better
# performance than np.linalg.eig()
l, v = np.linalg.eigh(S)
print(f"l:\n{l},\nv:\n{v}")
```

```
l:
[ 8.45974247 99.41320753],
v:
[[-0.98783557  0.15550202]
 [ 0.15550202  0.98783557]]
```

```
# We sort the eigenvectors by their eigenvalues
# the higher the eigenvalue, the higher importance it associates
idx = np.argsort(l)[::-1]
l = l[idx]
v = v[:,idx]
print(f"v:\n{v}")
```

```
v:
[[ 0.15550202 -0.98783557]
 [ 0.98783557  0.15550202]]
```

With eigenvectors, we can plot the projection line on our graph.

```
eigenvector = pd.DataFrame(np.vstack([v[:, 0]*70, v[:, 0]*-30]), columns=["stem-height", "

eigenvector_chart = alt.Chart(eigenvector).mark_line(color="grey", opacity=0.8).encode(
    x="stem-height",
    y="stem-width",
    tooltip=["stem-height", "stem-width"],
)

eigenvector_chart + mushroom_chart
```

```
alt.LayerChart(...)
```

And do the projections.

```
# We can then use sorted eigenvectors to transform the orginal data
# For example, 2D -> 1D
mushroom["projection"] = (v.T[:1] @ D).reshape(-1)
mushroom.head(5)
```

	stem-height	stem-width	projection
0	10.910405	5.343931	6.975515
1	1.410405	2.843931	3.028657
2	4.410405	2.843931	3.495163
3	4.410405	5.343931	5.964752
4	4.410405	2.843931	3.495163

The process of covariance matrix and eigenvalue decomposition can be reduced to only using SVD decomposition on centred data. This gives better numerical stability because we don't have to calculate `D @ D.T` which could cause errors with floating numbers.

```
u, s, vh = np.linalg.svd(D)
# There is even no need to sort
# 2D -> 1D
mushroom["projection_svd"] = (u.T[:1] @ D).reshape(-1)
mushroom.head(5)
```

	stem-height	stem-width	projection	projection_svd
0	10.910405	5.343931	6.975515	6.975515
1	1.410405	2.843931	3.028657	3.028657
2	4.410405	2.843931	3.495163	3.495163
3	4.410405	5.343931	5.964752	5.964752
4	4.410405	2.843931	3.495163	3.495163

I demonstrated a simple way to project $\mathbf{R}_2 \to \mathbf{R}_1$. In practice, you normally do PCA on much higher dimensions. For example, here we can see that the cap diameter is correlated with stem sizes. You can try to find a projection that transform $\mathbf{R}_3 \to \mathbf{R}_2$ or $\mathbf{R}_3 \to \mathbf{R}_1$.

```
mushroom = pd.read_csv("https://mushroom.mathematik.uni-marburg.de/files/PrimaryData/prima
mushroom["cap-diameter"] = mushroom["cap-diameter"].apply(calculate_mean)
mushroom["stem-height"] = mushroom["stem-height"].apply(calculate_mean)
mushroom["stem-width"] = mushroom["stem-width"].apply(calculate_mean)
alt.Chart(mushroom).mark_circle().encode(
    x="stem-height",
    y="stem-width",
    size="cap-diameter",
    tooltip=["stem-height", "stem-width", "cap-diameter"],
)
```

```
alt.Chart(...)
```

Further Reading

Making sense of principal component analysis, eigenvectors & eigenvalues
Eigenfaces, for Facial Recognition
Manifold learning - Scikit-Learn
Visualizing Data Using t-SNE

CS168 Lecture 7: Understanding and Using Principal Component Analysis (PCA)
CS168 Lecture 8: How PCA Works