BUSA3020-Week6

February 13, 2023

BUSA3020 - Advanced Analytics Techniques

Week 6 Lecture & Computer Lab - Dimensionality Reduction via Data Compression

Unit Convenor & Lecturer

George Milunovich
george.milunovich@mq.edu.au

References

1. Python Machine Learning 3rd Edition by Raschka & Mirjalili - Chapter 5
2. Various open-source material

Week 6 Learning Objectives

- Principal Component Analysis (PCA) for Unsupervised Data Compression
- Linear Discriminant Analysis (LDA) for Supervised Dimensionality Reduction
- Kernel Principal Component Analysis (KPCA) for Nonlinear Dimensionality Reduction

---

Dimensionality Reduction and the Curse of Dimensionality

The **curse of dimensionality** refers to various phenomena that arise when analyzing and organizing data in high-dimensional spaces (many features) that do not occur in low-dimensional settings such as the three-dimensional physical space.

The expression was coined by Richard E. Bellman when considering problems in dynamic programming. - https://en.wikipedia.org/wiki/Curse_of_dimensionality.

In Predictive Analytics/Machine Learning working in high-dimensional feature spaces can be undesirable for two reasons: 1. Training a model / estimating parameters precisely requires a large amount of data -> length of the dataset required grows exponentially with the dimensionality (number of features) 2. Making predictions from big datasets without dimensionality reduction is usually computationally intractable

Dimensionality reduction is the transformation of data from a high-dimensional space (many features) into a low-dimensional space (fewer features) so that the low-dimensional representation

retains some meaningful properties of the original data.

In Week 5 we looked at two differnt techniques for Dimensionality Reduction: 1. Regularization (L1 and L2) 2. Feature Selection (Sequential Backward Selection)

Both of these methods attempt reduce the dimension of the feature space by **picking relevant features and discarding others** via some optimization technique

An alternative approach we consider this week is **feature extraction**: - **Summarise the information content** of a dataset by **transforming** the data into a new feature subspace of lower dimension - This is a type of data compression where we attempt to extract the most relevant information from a dataset

---

Principal Component Analysis (PCA) - Unsupervised Dimensionality Reduction

- PCA is an **unsupervised linear** transformation technique used for dimensionality reduction
  - PCA attempts to find orthogonal (right-angle/uncorrelated) features which explain most of the **variance** in high dimensional data

PCA Method - Start with d features $\mathbf{x} = \begin{pmatrix} x_1 & x_2 & ... & x_d \end{pmatrix}$ where $x \in \mathbb{R}^d$ - Find a matrix $W \in \mathbb{R}^{d \times k}$ where $k < d$ which will transform our data - Transform data $z = xW$ (this will transform one example/observation x) - New features are $\mathbf{z} = \begin{pmatrix} z_1 & z_2 & ... & z_k \end{pmatrix}$ where the elements of $z \in \mathbb{R}^k$ are called **principal components**. - To transform entire dataset do $Z = XW$

Notes: - The elements of x are likely to be correlated (as any two random variables can be) - Because W is constructed in a special way the elements of z will be uncorrelated

For instance if we wish to reduce 5 features ($d = 5$) into 2 features ($k = 2$) the transformation will be as follows

[ ]:

PCA Algorithm

So the key question is how do we find transformation matrix W

The main steps behind PCA Algorithm are

1. Standardise the original d-dimensional dataset
2. Construct the covariance matrix $= \begin{pmatrix} \sigma_1^2 & \sigma_{12} & ... & \sigma_{1d} \\ \sigma_{21} & \sigma_2^2 & ... & \sigma_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ \sigma_{d1} & \sigma_{d2} & ... & \sigma_d^2 \end{pmatrix}$
3. Decompose the covariance matrix into its eigenvectors and eigenvalues
   - Find d eigenvectors (ν) and d corresponding eigenvalues (λ) such that $\Sigma\nu = \lambda\nu$
   - Methods to find ν and λ to the above equation are taught in linear algebra courses
4. Sort the eigenvalues by decreasing the order to rank the corresponding eigenvectors
   - Sort λs such that $\lambda_1 > \lambda_2$ ... find correponding $\nu_1, \nu_2, ...$
   - Note that $\sum_{j=1}^{d} \sigma_j^2 = \sum_{j=1}^{d} \lambda_j$

- The jth principal component accounts for or "explains" $\frac{\lambda_j}{\sum_{j=1}^{d} \lambda_j}$ percent of the overall variability (this is called explained variance ratio)

5. Select k eigenvectors which correspond with the k largest eigenvalues, where k is the dimensionality of the new feature subspace ($k \leq d$)
   - These k eigenvectors contain most information (variance) of the original dataset
6. Construct a projection matrix W from the "top" k eigenvectors
7. Transform the d-dimensional input dataset x using the projection matrix W to obtain the new k-dimensional feature vector z.
   - The elements of z are called **principal components**.

Creating Principal Components

a) Import the Wine dataset from Week 5, split in into training and test (30%) datasets and standardize the data

```
import pandas as pd
pd.options.display.float_format = '{:,.4f}'.format

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# df = pd.read_csv('https://archive.ics.uci.edu/ml/'
#                        'machine-learning-databases/wine/wine.data',
#                        header=None)

# df.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
#                      'Alcalinity of ash', 'Magnesium', 'Total phenols',
#                      'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
#                      'Color intensity', 'Hue',
#                      'OD280/OD315 of diluted wines', 'Proline']

# df.to_excel('data/wine.xls', index = False)

df = pd.read_excel('data/wine.xls')
# df

y, X = df.iloc[:, 0], df.iloc[:, 1:]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, stratify=y, random_sta

sc = StandardScaler()
X_train_scaled = sc.fit_transform(X_train)
X_test_scaled = sc.transform(X_test)

X_train_scaled.shape

print(df.info())
df
```

[ ]:

---

b) Use `np.cov` to compute the covariance matrix of `X_train_scaled`

- https://numpy.org/doc/stable/reference/generated/numpy.cov.html

```
import numpy as np

cov_mat = np.cov(X_train_scaled.T) # .T to transpose (flip) array

print(cov_mat.shape)
pd.DataFrame(cov_mat)
```

[ ]:

---

c) Use `np.linalg.eig` to compute eigenvalues and eigenvectors from the covariance matrix

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html

```
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)

print(eigen_vals.shape, eigen_vecs.shape)

print(pd.Series(eigen_vals))
pd.DataFrame(eigen_vecs)
```

[ ]:

---

d)

- Plot explained variance ratios for each eigenvector.
- Also plot the cummulative sum explained by the first i eigenvectors using `np.cumsum`
- https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html

```
import matplotlib.pyplot as plt


total_variance = sum(eigen_vals)
var_exp = [(i / total_variance) for i in sorted(eigen_vals, reverse=True)]
cummulative_var_exp = np.cumsum(var_exp)
cummulative_var_exp
```

```
plt.bar(range(1, 14), var_exp, alpha = 0.5, align = 'center', label = 'Individual Explained Va
plt.step(range(1, 14), cummulative_var_exp, where='mid', label='Cummulative Explained Variance
plt.ylabel('Explained Variance Ratios')
plt.xlabel('Principal Component Index')
plt.legend(loc='best')
plt.tight_layout()
plt.show()
```

[ ]:

---

e) Create principal components z_1 and z_2 corresponding to 2 largest eigenvalues by following steps 5. - 7. above

```
print(eigen_vecs.shape)

W = eigen_vecs[:, :2]

print(W)

# Z = X_train_scaled.dot(W)
Z = np.dot(X_train_scaled, W)

print(Z.shape)
print('First 2 PCs:\n', Z[:10, :])
```

[ ]:

---

f) z_1 and z_2 now represent our new features, label them according to y and plot

```
import matplotlib.pyplot as plt


colors = ['r', 'b', 'g']
markers = ['s', 'x', 'o']
for lab, col, mark in zip(np.unique(y_train), colors, markers):
    print(lab, col, mark)
    plt.scatter(Z[y_train==lab, 0], Z[y_train==lab, 1], c = col, label = lab, marker = mark)

plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

[ ]:

PCA in scikit-learn

PCA class in scikit-learn is a `transformer` class - `from sklearn.decomposition import PCA` 1. Fit PCA using training data 2. Transform training and test data

Example - Wine Dataset 1. Use PCA from sklearn on Wine dataset 2. Train LogisticRegression on the first 2 principal components 3. Print forecast accuracy on both train and test datasets 4. Plot decision boundaries for the train and test dataset using `plot_decision_regions` function we wrote previously 5. Print explained variance ratios

```
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
import plot_decision_regions as pdr

# ----- Extract Principal Components
pca = PCA(n_components = 2)
X_train_pca = pca.fit_transform(X_train_scaled)
X_test_pca = pca.transform(X_test_scaled)



# ----- Fit logistic regression
lr = LogisticRegression(multi_class='ovr', random_state=1, solver='lbfgs')
lr.fit(X_train_pca, y_train)

print('Accuracy - Training dataset', lr.score(X_train_pca, y_train))


pdr.plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()

# ----- Test Data

print('Accuracy - Training dataset', lr.score(X_test_pca, y_test))
pdr.plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()

#--------- Explained Variance Ratios

pca2 = PCA(n_components = None)  # all principal components are kept
X_train_pca2 = pca2.fit_transform(X_train_scaled)
```

```
plt.bar(np.arange(1, pca2.explained_variance_ratio_.shape[0] + 1), pca2.explained_variance_rati
plt.xlabel('PC i')
plt.ylabel('% of variance explained')
plt.xticks(np.arange(1, pca2.explained_variance_ratio_.shape[0] + 1))
plt.tight_layout()
plt.show()

plt.show()
```

---

1. Use PCA from sklearn on Wine dataset

```
from sklearn.linear_model import LogisticRegression
from sklearn.decomposition import PCA
import plot_decision_regions as pdr

# ----- Extract Principal Components
pca = PCA(n_components = 2)

X_train_pca = pca.fit_transform(X_train_scaled)

X_test_pca = pca.transform(X_test_scaled)

X_train_pca[:10, :]
```

[ ]:

2. Train LogisticRegression on the first 2 principal components

```
lr = LogisticRegression(multi_class='ovr', random_state=1, solver='lbfgs')

lr.fit(X_train_pca, y_train)

pdr.plot_decision_regions(X_train_pca, y_train, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

[ ]:

3. Print forecast accuracy for both the training and test datasets

```
print(f'Accuracy - Training dataset {lr.score(X_train_pca, y_train):.3f}')
print(f'Accuracy - Test dataset {lr.score(X_test_pca, y_test):.3f}')

pdr.plot_decision_regions(X_test_pca, y_test, classifier=lr)
plt.xlabel('PC 1')
plt.ylabel('PC 2')
```

```
plt.legend(loc='lower left')
plt.tight_layout()
plt.show()
```

[ ]:

5. Print explained variance ratios

```
pca2 = PCA(n_components = None)  # all principal components are kept

X_train_pca2 = pca2.fit_transform(X_train_scaled)

plt.bar(np.arange(1, pca2.explained_variance_ratio_.shape[0] + 1), pca2.explained_variance_rati

plt.xlabel('PC i')
plt.ylabel('% of variance explained')
plt.xticks(np.arange(1, pca2.explained_variance_ratio_.shape[0] + 1))
plt.tight_layout()
plt.show()

plt.show()
```

[ ]:

Linear Disciminant Analysis (LDA)

Both PCA and LDA are **linear transformation** techniques used to reduce the number of dimensions in a dataset.

However - PCA computes orthogonal/uncorrelated components that account for most of the total variance of the **features** without taking into account y (unsupervised learning) - LDA finds features which optimize class separability - takes into account target variable y -> supervised learning

LDA assumes:
- Data is normally distributed - Different classes have identical covariance matrices - Training examples (observations) are statistically independent of each other

However, even if some assumptions are false LDA still performs reasonably well in practice

LDA Algorithm

The main steps behind LDA are outlined below. For details and a step-by-step implementation in python see textbook.

1. Standardise the original d-dimensional dataset

2. For each **class** $i \in \{1, \dots, c\}$ compute the d-dimensional mean vector $\mu_i = \begin{pmatrix} \mu_{i,1} \\ \mu_{i,2} \\ \dots \\ \mu_{i,d} \end{pmatrix}$

3. For each class compute a covariance matrix of elements belonging to that class $\Sigma_i = \frac{1}{n_i}\sum_{x \in D_i}(x - \mu_i)(x - \mu_i)^T$ and then sum them to compute **within-class** scatter matrix $S_W = \sum_{i=1}^{c} \Sigma_i = (\Sigma_1 + \cdots + \Sigma_c)$
4. Compute the **between-class** scatter matrix $S_B = \sum_{i=1}^{c} n_i(\mu_i - \mu)(\mu_i - \mu)^T$ where μ is the vector of overall means for each feature (note: this is like a variance of the means)
5. Compute the eigenvalues and corresponding eigenvectors of the matrix $S_W^{-1}S_B$
6. Stack the k eigenvectors that correspond to the k largest eigenvalues as columns into a $d \times d$ dimensional transformation matrix W
7. Transform the d-dimensional input dataset X using the projection matrix W to obtain the new k-dimensional feature vector $Z = XW$.

LDA via Scikit-Learn

- As with PCA we will implement LDA via scikit-learn

1. Train LDA transformer and extact 2 linear discriminents from the **training dataset** (now need to include y_train as well)
2. Fit train LogisticRegression on the 2 LDs
3. Print accuracy and plot decision regions
4. Extract 2 LDs from the **test** dataset, print LR accuracy and plot decision regions

```
#--------------  1.
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
lda = LDA(n_components = 2)
X_train_lda = lda.fit_transform(X_train_scaled, y_train)
pd.DataFrame(X_train_lda)

#--------------  2.
lr = LogisticRegression(multi_class='ovr', random_state=1)
lr.fit(X_train_lda, y_train)
#--------------  3.
print('Accuracy - Training:', lr.score(X_train_lda, y_train))
pdr.plot_decision_regions(X_train_lda, y_train, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_09.png', dpi=300)
plt.show()
#--------------  4.
X_test_lda = lda.transform(X_test_scaled)
print('Accuracy - Training:', lr.score(X_test_lda, y_test))

pdr.plot_decision_regions(X_test_lda, y_test, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
```

```
plt.tight_layout()
# plt.savefig('images/05_10.png', dpi=300)
plt.show()
```

---

1. Train LDA transformer and extact 2 linear discriminents from the training dataset (now need to include y_train as well)

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA


lda = LDA(n_components = 2)

X_train_lda = lda.fit_transform(X_train_scaled, y_train) # we used to call this Z above

pd.DataFrame(X_train_lda)
```

[ ]:

2. Fit train LogisticRegression on the first 2 LDs

```
lr = LogisticRegression(multi_class='ovr', random_state=1)

lr.fit(X_train_lda, y_train)
```

[ ]:

3. Print accuracy and plot decision regions

```
print('Accuracy - Training:', lr.score(X_train_lda, y_train))


pdr.plot_decision_regions(X_train_lda, y_train, classifier=lr)
plt.xlabel('LD 1')
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_09.png', dpi=300)
plt.show()
```

[ ]:

4. Extract 2 LDs from the test dataset, print LR accuracy and plot decision regions

```
X_test_lda = lda.transform(X_test_scaled)


print('Accuracy - Training:', lr.score(X_test_lda, y_test))

pdr.plot_decision_regions(X_test_lda, y_test, classifier=lr)
plt.xlabel('LD 1')
```

```
plt.ylabel('LD 2')
plt.legend(loc='lower left')
plt.tight_layout()
# plt.savefig('images/05_10.png', dpi=300)
plt.show()
```

[ ]:

Kernel Principal Component Analysis (KPCA) for Nonlinear Mappings

In real world situations where data is often not linearly separable emplying PCA and LDA, which are linear transformation techniques for dimensionality reduction may not be the best practice.

KPCA will transform data which is not linearly separable onto a new, lower-dimensional subspace which is linearly separable.

In SVM applications we used kernels to tackle nonlinear problems by transforming features to a higher dimensional space where the classes become linearly separable.

Kernel Function and the Kernel Trick

KPCA works as follows: 1. Use a nonlinear function ϕ to transform features onto a higher-dimensinal space - Nonlinear mapping $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ where $k > d$ - For example $x = [x_1 \quad x_2]^T \underset{\phi}{\rightarrow} z = [x_1^2 \quad \sqrt{2x_1x_2} \quad x_2^2]^T$ so here $d = 2, k = 3$ 2. Reduce dimension using standard PCA to project the data back onto a lower-dimensinal space where the data is linearly separable - Compute similarity (kernel) of nonlinear functions in a high-dimensional space as the dot product of the nonlinear functions

Problem: Step 2 is very computationally expensive
Solution: **Kernel Trick** - Compute similarity (kernel) of nonlinear functions in a high-dimensional space as a nonlinear function of the dot product of the original features in a low-dimensional space - The order of operations is reversed 1. Computationally expensive: dot product of non-linear functions in a high-dimsional space 2. Computationally inexpensive - Kernel Trick: a nonlinear function of the dot product of original features in a low-dimensional space

KPCA in scikit-learn

- We will not implement KPCA ourselves, but will use the libraries provided in scikit-learn
  - For those more adventurous follow the steps in the textbook to build your own KPCA Python library

Example 1: Separating Half-Moon Shapes
Example 2: Separating Concentric Circles
a) Import concentric circles data data

```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples = 1000, random_state = 123, noise=0.1, factor=0.2)
```

```
plt.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.tight_layout()
plt.show()

pd.DataFrame(np.hstack((X, y.reshape(-1,1))), columns=['x1', 'x2', 'Class'])
```

b) Extract standard PCA and see if the classes are linearly separable

```
from sklearn.decomposition import PCA
pca = PCA(n_components = 2)
X_pca = pca.fit_transform(X)


plt.scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```

c) Extract kernel PCA (rbf kernel) and see if classes are linearly separable

```
from sklearn.decomposition import KernelPCA
kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)
X_kpca = kpca.fit_transform(X)

plt.scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('K-PC1')
plt.ylabel('K-PC2')
plt.show()
```

Example 1: Separating Half-Moon Shapes

a) Import half-moons data

```
import matplotlib.pyplot as plt

from sklearn.datasets import make_moons

import pandas as pd

X, y = make_moons(n_samples = 100, random_state = 123)

plt.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.tight_layout()
plt.show()
```

```
pd.DataFrame(np.hstack((X, y.reshape(-1,1))), columns=['x1', 'x2', 'Class'])
```

[ ]:

b) Extract standard PCA and see if the classes are linearly separable

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)

X_pca = pca.fit_transform(X)


plt.scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```

[ ]:

c) Extract kernel PCA (rbf kernel) and see if classes are linearly separable

```
from sklearn.decomposition import KernelPCA

kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)

X_kpca = kpca.fit_transform(X)

plt.scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('K-PC1')
plt.ylabel('K-PC2')
plt.show()
```

[ ]:

Example 2: Separating Concentric Circles

a) Import concentric circles data data

```
from sklearn.datasets import make_circles

X, y = make_circles(n_samples = 1000, random_state = 123, noise=0.1, factor=0.2)

plt.scatter(X[y==0, 0], X[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X[y==1, 0], X[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.tight_layout()
plt.show()
```

```
pd.DataFrame(np.hstack((X, y.reshape(-1,1))), columns=['x1', 'x2', 'Class'])
```

[ ]:

b) Extract standard PCA and see if the classes are linearly separable

```
from sklearn.decomposition import PCA

pca = PCA(n_components = 2)

X_pca = pca.fit_transform(X)


plt.scatter(X_pca[y==0, 0], X_pca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_pca[y==1, 0], X_pca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```

[ ]:

c) Extract kernel PCA (rbf kernel) and see if classes are linearly separable

```
from sklearn.decomposition import KernelPCA

kpca = KernelPCA(n_components=2, kernel='rbf', gamma=15)

X_kpca = kpca.fit_transform(X)

plt.scatter(X_kpca[y==0, 0], X_kpca[y==0, 1], color='red', marker='^', alpha=0.5)
plt.scatter(X_kpca[y==1, 0], X_kpca[y==1, 1], color='blue', marker='o', alpha=0.5)
plt.xlabel('K-PC1')
plt.ylabel('K-PC2')
plt.show()
```

[ ]:

[ ]: