Lecture 13: Dimensionality Reduction

Chapter 5

Basic concepts

- **Feature selection**: maintain the original features
- **Feature extraction**: transform or project the data onto a new feature space.
  - Improve storage space
  - Improve the computational efficiency of the learning algorithm
  - Improve the predictive performance by reducing the *curse of dimensionality*. This is particularly useful when we work with non-regularized models.
  - Principal component analysis (**PCA**) for unsupervised dimensionality reduction
  - Linear discriminant analysis (**LDA**) as supervised dimensionality reduction
  - Kernel principal component analysis (**KPCA**) for nonlinear dimensionality reduction

Principal Component Analysis (PCA)

- Principal Component Analysis (PCA) is an unsupervised linear transformation technique.
- PCA helps identify patterns in data based on the **correlation** between features.
  - PCA aims to find the directions of **maximum variance** in high-dimensional data and projects it onto a new subspace with equal or fewer dimensions than the original one.
  - The new feature axes are orthogonal to each other.
  - The orthogonal axes (principal components) of the new subspace can be interpreted as the directions of maximum variance.

PCA

- We construct a d × k-dimensional transformation matrix **W** $\in \mathbb{R}^{d \times k}$**.**
- This matrix allows us to map a vector $\mathbf{x} \in \mathbb{R}^{d}$, which is a training sample, to a new k-dimensional feature subspace $z \in \mathbb{R}^{k}$.

- Given $\mathbf{x} = [x_1, x_2, \ldots, x_d]$
- Calculate a transformation matrix **W**
- Project $\boldsymbol{x}$ to the new space: $\mathrm{x}W = z$ and get $\mathbf{z} = [z_1, z_2, \ldots, z_k]$

PCA

- Transforming $\mathbf{x} \in \mathbb{R}^d$ to $z \in \mathbb{R}^k$ where $d \gg k$
  - Typically, *k* is much smaller than *d*.
  - The first principal component will have the largest possible variance.
  - The resulting principal components are uncorrelated (orthogonal) to each each other.
  - PCA directions are highly sensitive to data scaling. We **need to standardize the features prior to PCA** if the features were in different scales.

Background

- The **covariance** between two features *j* and *k* is calculated as

$$\sigma_{jk} = \frac{1}{n}\sum_{i=1}^{n}\left(x_j^{(i)} - \mu_j\right)\left(x_k^{(i)} - \mu_k\right)$$

  - where μ_j and μ_k are the sample mean of features *j* and *k*.
- Note that the mean is zero if we standardize the data.
- **Covariance matrix**: *d* × *d* where *d* is the number of dimensions in the dataset.

- A 3× 3 covariance matrix can be represented as $\sum = \begin{pmatrix} \sigma_1^2 & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_2^2 & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_3^2 \end{pmatrix}$

Covariance properties

- A positive covariance between two features indicates that the features increase or decrease together.
- A negative covariance indicates that the features vary in opposite directions.
- The first principal component will have the largest possible variance.

Eigenvectors of covariance matrix

- The eigenvectors of the covariance matrix represent the principal components.
- The corresponding eigenvalues define their magnitude.
- An eigenvector $\mathbf{v}$ satisfies the equation $\sum \mathbf{v} = \lambda \mathbf{v}$
  - λ is a scalar, called the eigenvalue.
- Eigenvalues define the magnitude of eigenvectors.
- For PCA analysis, we need to sort eigenvalues by decreasing magnitude.

Get eigenpairs

- We can use the **linalg.eig** function from the NumPy to obtain the eigenpairs.
- **numpy.cov** function calculates the covariance matrix of a dataset.
  - Note the dataset needs to be represented in the format of $d \times n$ format where each row represent one feature. Otherwise, the covariance matrix is the covariance of the instances.
- **linalg.eig** function conducts the eigen decomposition
  - Returns (i) a vector (eigen_vals) of d eigenvalues and (ii) the corresponding eigenvectors stored as columns in a $d \times d$ matrix (eigen_vecs)

Example

```
# import wine dataset
# partition the data to training and testing datasets
# standardize the features, X_train_std

cov_mat = np.cov(X_train_std.T)
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)

print('\nEigenvalues \n%s' % eigen_vals)
```

```
Eigenvalues
[4.84274532 2.41602459 1.54845825 0.96120438 0.84166161 0.6620634
0.51828472 0.34650377 0.3131368 0.10754642 0.21357215 0.15362835
0.1808613 ]
```

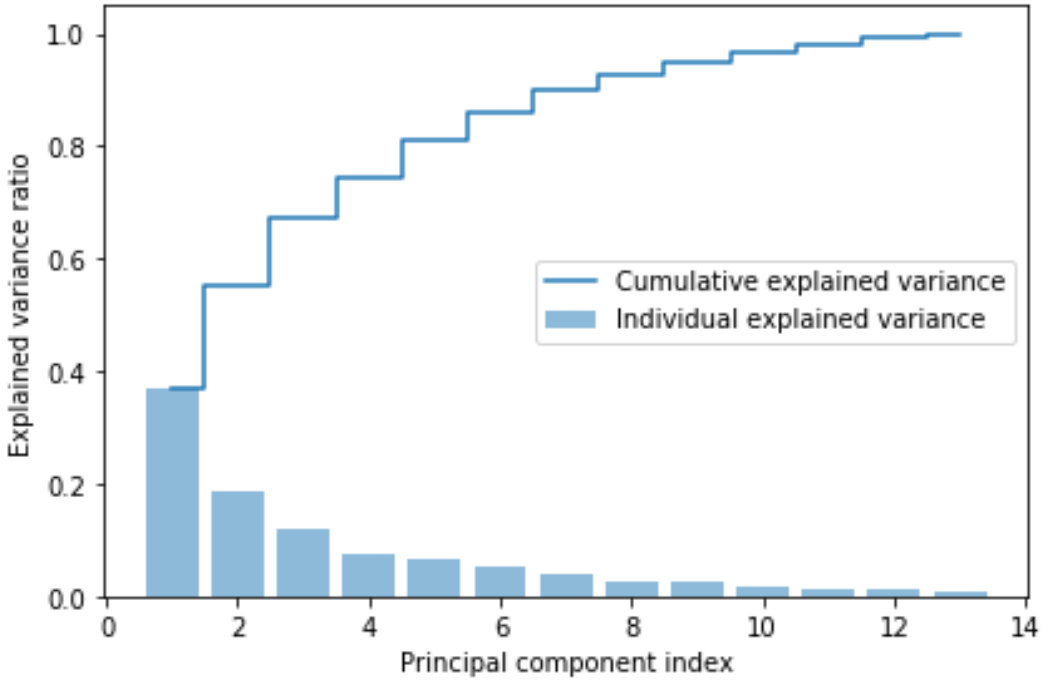
Variance explained ratios

- The variance explained ratios of an eigenvalue λ_j is $\frac{\lambda_j}{\sum_{i=1}^{d} \lambda_i}$
- The first PC accounts for ~ 40% of the variance, the first two PCs account for almost 60% of the variance.

```
# calculating eigen values (see previous slides)

tot = sum(eigen_vals)
var_exp = [(i / tot) for i in sorted(eigen_vals, reverse=True)]
cum_var_exp = np.cumsum(var_exp)

# plotting code (see text book)
```



Conduct PCA step by step

- Standardize the *d*-dimensional dataset
- Construct the covariance matrix.
- Decompose the covariance matrix into its eigenvectors and eigenvalues.
- Select *k* eigenvectors which correspond to the *k* largest eigenvalues, where *k* is the dimensionality of the new feature subspace.
- Construct a projection matrix **W** from the top *k* eigenvectors.
- Transform the *d*-dimensional input dataset **X** using the projection matrix **W** to obtain the new *k*-dimensional feature subspace.

Steps 1-3: standardization, covariance matrix, eigen-decomposition

```
# import wine dataset
# partition the data to training and testing datasets

# STEP 1: Standardize the d−dimensional dataset
sc = StandardScaler()
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# STEP 2: Construct the covariance matrix
print(X_Train_std.T.shape)
cov_mat = np.cov(X_train_std.T)

# STEP 3: Decompose the covariance matrix into its eigenvectors and eigenvalues
eigen_vals, eigen_vecs = np.linalg.eig(cov_mat)
```

Step 4: select *k* eigenvectors which correspond to the *k* largest eigenvalues

```
# STEP 4: Select k eigenvectors which correspond to the k largest eigenvalues
# Make a list of (eigenvalue, eigenvector) tuples
eigen_pairs = [(np.abs(eigen_vals[i]), eigen_vecs[:, i])
               for i in range(len(eigen_vals))]

# Sort the (eigenvalue, eigenvector) tuples from high to low
eigen_pairs.sort(key=lambda k: k[0], reverse=True)
```

- Python list's sort function. **sort(*, key=None, reverse=False).**
  - **key** specifies a function to be called on each list element prior to making comparisons. (for example, key=str.lower).
  - **reverse** is a boolean value. If set to True, then the list elements are sorted as if each comparison were reversed.
- More explanations about sort function and python lambda function.

Step 4

```
print(eigen_pairs[0])
print(eigen_pairs[1])

print(eigen_pairs[0][1])
```

```
(4.842745315655895, array([-0.13724218, 0.24724326, -0.02545159, 0.20694508, -0.15436582, -0.39376952, -0.41735106, 0.30572896, -0.30668347, 0.07554066, -0.32613263, -0.36861022, -0.29669651]))
(2.416024587035225, array([ 0.50303478, 0.16487119, 0.24456476, -0.11352904, 0.28974518, 0.05080104, -0.02287338, 0.09048885, 0.00835233, 0.54977581, -0.20716433, -0.24902536, 0.38022942]))

[-0.13724218 0.24724326 -0.02545159 0.20694508 -0.15436582 -0.39376952 -0.41735106 0.30572896 -0.30668347 0.07554066 -0.32613263 -0.36861022 -0.29669651]
```

Step 5: Construct a projection matrix **W** from the top *k* eigenvectors

```
w = np.hstack((eigen_pairs[0][1][:, np.newaxis],
               eigen_pairs[1][1][:, np.newaxis]))
print('Matrix W:\n', w)
```

```
Matrix W:
[[-0.13724218 0.50303478]
 [ 0.24724326 0.16487119]
 [-0.02545159 0.24456476]
 [ 0.20694508 -0.11352904]
 [-0.15436582 0.28974518]
 [-0.39376952 0.05080104]
 [-0.41735106 -0.02287338]
 [ 0.30572896 0.09048885]
 [-0.30668347 0.00835233]
 [ 0.07554066 0.54977581]
 [-0.32613263 -0.20716433]
 [-0.36861022 -0.24902536]
 [-0.29669651 0.38022942]]
```

- **numpy.hstack**(): Stack arrays in sequence horizontally (column wise). This is equivalent to concatenation along the second axis

Step 6: Transform $\mathbf{X}$ to obtain the new *k*-dimensional feature subspace.

- We can transform one instance in the original space to the new space by calculating $\mathbf{x}' = \mathbf{x}\mathbf{W}$.
- Or, we can transform the entire training dataset onto the two principal components by calculating $\mathbf{X}' = \mathbf{X}\mathbf{W}$.

Step 6

```
# STEP 6: Transform the d−dimensional input dataset X using the projection matrix W
# to obtain the new k−dimensional feature subspace
X_train_pca = X_train_std.dot(w)
print("1st original instance: ", X_train_std[0])
print("Instance in PC space: ", X_train_pca[0])
print("Variance in PC1 = %.2f" %np.var(X_train_pca[:,0]))
print("Variance in PC2 = %.2f" %np.var(X_train_pca[:,1]))
```

```
1st original instance: [ 0.71225893 2.22048673 -0.13025864 0.05962872 -0.50432733 -0.52831584 -
1.24000033 0.84118003 -1.05215112 -0.29218864 -0.20017028 -0.82164144 -0.62946362]
Instance in PC space: [2.38299011 0.45458499]
Variance in PC1 = 4.80
Variance in PC2 = 2.40
```

- It is clear that the data is more spread along the first PC than the second PC. This is consistent with the explained variance ratio plot.

Using scikit-learn library to conduct PCA

- Check the explained variance ratios of the different PCs, we can set **n_components** to be None (which keeps all the PCs). Then, we can assess the explained variance ratio via the **explained_variance_ratio_** attribute.
- Implement Steps 2-6 in a black box

```
from sklearn.decomposition import PCA

pca = PCA()
X_train_pca = pca.fit_transform(X_train_std)
pca.explained_variance_ratio_
```

```
array([0.36951469, 0.18434927, 0.11815159, 0.07334252, 0.06422108, 0.05051724, 0.03954654, 0.02643918,
0.02389319, 0.01629614, 0.01380021, 0.01172226, 0.00820609])
```

Example

```
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_std)

tree_model = DecisionTreeClassifier(criterion='gini',
                                    max_depth=4, random_state=1)
tree_model.fit(X_train_lda, y_train)

X_test_pca = pca.transform(X_test_std)
y_pred = tree_model.predict(X_test_pca)

acc = accuracy_score(y_pred, y_test)
print("DT+PCA acc=", acc)
```

```
DT+PCA acc= 0.9259259259259259
```

- Wine dataset
  - Accuracy (DT) is 0.89
  - Accuracy (DT+PCA) is 0.92

Discussions

- A natural measure is to pick the eigenvectors that explain *p*% of the data variance.

- PCA is not optimal for classification
  - There is no mention of the class label.
  - Keeping the dimensions of largest energy (variance) is a good idea but not always.
  - The discriminant dimensions could be thrown out.