Kernel PCA

TUTORIAL PAPER

ARSHIA SHARMA

TABLE OF CONTENTS

I do not give consent for this to be used as a teaching resource.

1 INTRODUCTION

Many problems in deep learning involve complex datasets with thousands of dimensions. Dimensionality reduction is an important task to make data more manageable and extract the most useful pieces of information. Principal Component Analysis (PCA) is a useful technique; it linearly transforms variables into a smaller set of uncorrelated variables, known as principal components. Its limitation is that it is purely a linear method. Kernel PCA (KPCA) is an extension of PCA that is used to extract non-linear features from data.

This tutorial paper covers the theoretical components of KPCA and provides a demonstration of one of the many possible applications of the technique.

2 THEORY

2.1 PCA REVIEW

To understand the steps of kernel PCA, standard PCA is briefly revisited.

PCA is a technique used to reduce dimensions of data without losing information; the aim is to retain most of information in the data while making it simpler. PCA has five steps. These are summarised briefly below.

1. Standardise the initial variables of the data. This ensures each variable contributes equally to the analysis.
2. Compute the $n \times n$ covariance matrix where n is the number of dimensions of the data. This matrix measures the direction of the relationship between all possible pairs of variables.
3. Compute the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors of the covariance matrix represent the directions of axes where there is the most variance. They are the principal components.
4. Construct a feature vector from a selection of principal components.
5. Recast the data along the principal component axes.

This is a relatively straightforward yet powerful technique. However, it is only applicable to datasets that are linearly separable. This is because it looks for linear dependencies between features; the orthonormal transformation of original coordinate system is meaningless when variables are not linearly related.

Further, applying linear PCA to a non-linearly separable dataset may result in subpar dimension reduction. The dataset will not be made linearly separable after applying PCA. A linearly separable dataset is necessary for some classification algorithms, such as Naïve Bayes Classifier. Further, PCA is not efficient in some facial recognition applications. This is because the variation between images of the same face due to illumination and viewing direction are larger than changes within faces themselves. So, completing PCA on an image dataset may reduce class separation and lead to poor results. KPCA is a solution to both problems. It can separate nonlinearly separable datasets and reduce dimensions of an image in a way that maximises class separation.

2.2 KERNEL PCA

Kernel PCA is like the PCA algorithm. The method uses a kernel function to project the dataset into a higher dimensional space where it is linearly separable. PCA is completed in this feature space.

1. Select a kernel function k. Choosing kernel functions is explored later.
2. Construct Gram Matrix
3. Find eigenvalues and eigenvectors of Gram Matrix.
4. Project data into new space

2.2.1 KERNEL FUNCTION

Suppose we have a dataset and (x_j) denotes coordinates of the j^{th} point. The dataset can be represented as 2-dimensional vector.

$$\mathbb{x} = [x_1, y_1 \ldots\ldots x_n, y_n]^T$$

Suppose we transform the data set to a higher dimensional feature space. Each data point is projected to a new point $\phi(\mathbb{x}_i)$. Since this is a higher dimension, the data is linearly separable and PCA can be performed. This linear algorithm operating in this higher dimensional space will behave non-linearly in the original input space.

However, explicitly calculating the mapping $\phi(x_i)$ is computationally quite difficult. A kernel function can be used instead here. A function k is a kernel if it is of the form:

$$k(x_i, x_j) = \phi(x_i)^T \phi(x_j)$$

That is, it is equal to the dot product of one point of transformed data to all transformed points. To generalise to more than two dimensions, a kernel functions computes the inner product of all possible pairs of transformed data points.

$$k(x_i, x_j) = < \phi(x_i) \phi(x_j) >$$

Computing this inner product is computationally challenging. It would require finding a valid mapping to a higher dimensional feature space. With the use of kernels, this calculation can be skipped. Instead, this calculation can be done using initial inputs x and y. To illustrate this equivalence, an example is provided using the polynomial kernel.

A polynomial kernel is defined for a d-degreed polynomial as k: $R^L \times R^L \rightarrow R$

To make the working less confusing, $x = x_i, y = x_j$

$$k(x, y) = (x^T y + c)^d$$

We choose L = d = 2. Let c = 1. These choices are arbitrary. We must show this is equivalent to (1) kernel equation above.

Then,

$$k(x_i, y) = (x^T y + c)^2$$

$$k(x, y) = (x^T y + 1)^2$$

$$k(x, y) = (x_1 y_1 + x_2 y_2 1)^2$$

$$k(x, y) = x_1^2 y_1^2 + x_2^2 y_2^2 + 1 + 2x_1 x_2 y_1 y_2 + 2x_1 y_1 + 2x_2 y_2$$

$$k(x, y) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2\right) \cdot \left(1, \sqrt{2}y_1, \sqrt{2}y_2, \sqrt{2}y_1 y_2, y_1^2, y_2^2\right)$$

Let, $\phi(x) = \left(1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1 x_2, x_1^2, x_2^2\right)$

Then, the above expression is:

$$k(x, y) = \phi(x)^T \phi(y)$$

This is the 'kernel trick'; the data can be transformed to a higher dimension and be used linearly while never having to explicitly compute the data in $\phi(x)$. The points are viewed as mappings(x) in that feature space, however, we don't need to explicitly calculate mapping(x) for each point in data set, but only pairwise dot products.

The kernel function for a specific application is chosen via cross-validation. In standard PCA, reconstruction error is the performance measure that is used on the test set. Kernel PCA reconstruction error can also be computed; however, the distance measured is in the target feature space. It is not comparable between different kernels. Different kernels correspond to different target feature spaces. Instead, reconstruction error can be calculated the original space; not in the target space. We can find a point in the original space that would be mapped as close as possible to the reconstruction point, and then measure the distance between these points as reconstruction error. Alternatively, cross-validation can also be completed by measuring the performance of different KPCA implementations on a classifier.

2.2.2 GRAM MATRIX

We revisit the formulation of the covariance matrix and compare it to the kernel matrix covariance. In linear PCA, the covariance matrix is constructed. However, covariance quantifies a linear relationship between points. Since the data is non-linear, we use kernel function is used to calculate the matrix. We will not provide the full derivation of this matrix; however, will provide the general intuition behind it.

Suppose all data in feature space has mean 0. $\sum_{i=1}^{m} \phi(x) = 0$

Then, covariance matrix is:

$$C = \frac{1}{m} \sum_{i=1}^{m} \phi(x_i)\phi(x_i)^T$$

Then, the eigenvectors are:

$$Cv_j = \lambda_j v_j \;\; j = 1 \dots N$$

So,

$$\frac{1}{m} \sum_{i=1}^{m} \phi(x_i)\phi(x_i)^T = \lambda_j v_j \;\; j = 1 \dots N$$

This is the equation we solve using kernels. After algebra, we get the following matrix:

$$\widehat{K} = K - 1_{\frac{1}{N}} K - K 1_{\frac{1}{N}} + 1_{\frac{1}{N}} K 1_{\frac{1}{N}}$$

Where $1_{\frac{1}{m}}$ is the n x n matrix with all elements equal to $1/m$.

Where K is the kernel matrix, constructed using

The kernel matrix consists of solving the kernel function for every possible pair of points in the dataset. It is a set of pairwise comparisons, where the comparison is the kernel function.

$K_{i,j} = k(x_i, x_j)$

3 DEMONSTRATION

To demonstrate how Kernel PCA can be implemented in practise, an example is provided in this section. A common application of PCA and Kernel PCA is in facial recognition. Both methods are used to classify images in the Olivetti Faces data set. Code snippets are provided here; the full code is available in Appendix A.

3.1 EXPLORATION.

In this section, we explore what the Olivetti data set looks like. The image dataset contains 400 images of 40 subjects. The images are 64 × 64 pixels in size. The photos were taken under controlled lighting and vary in pose as well as scale. The ten images of the first subject are displayed below. The dataset is split into random train splits. 80% of the data is used for training. 20% of the data is used for testing.

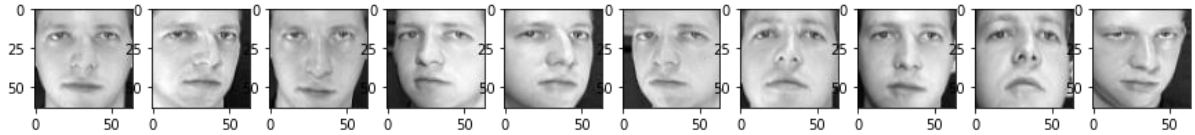

Fig1: Images of the first subject.

We perform PCA on the dataset with and find that most variance can be explained with 50 principal components. Both training and testing datasets are projected onto the trained principal components. Scikit-learn's method of Support Vector Classification (SVC) was fitted using the training PCA transformation and predictions were made on the test PCA projection. The accuracy of the classification was 89%.

Kernel PCA was applied as well with a simple gaussian kernel and a gamma value of 0. This code is provided below. In Scikit-Learn, Kernel PCA is applied similarly to standard PCA.

```
1.  kpca = KernelPCA(kernel = 'rbf')
2.  X_train = kpca.fit_transform(x_train)
3.  X_test = kpca.transform(x_test)
4.
```

3.2 TUNING HYPERPARAMETERS

Kernel PCA is an unsupervised learning algorithm. There is no obvious performance measure to select the best kernel and hyperparameter values. Grid search can be used to find the best values. A grid search method is presented here, that uses various kernel types and gamma values. It calculates the performance for each combination and selects the best value for the hyperparameters. Scikit-Learn's GridSearchCV has been used here. The kernel functions used were: Gaussian, Linear and Polynomial kernels. Gamma values from 0 to 0.5 were also used.

Scikit-Learn's KernelPCA method can compute the pre-image of a point if inverse_transform parameter is set to true. The inverse transform method approximates the inverse transformation into the original space using the learned preimage.

GridSearchCV requires a scoring function (score in code snippet) to evaluate the performance of the parameters. A possible scoring function could be simply just testing SVC classifier performance and aiming for the best classification accuracy. However, the approach used here is to find parameters with lowest reconstruction error. This has been implemented as it was described in Section 2.2

```
def score(estimator, X=x_train):
    x_red = estimator.transform(X)
    x_pre = estimator.inverse_transform(x_red)
    return -1 * mean_squared_error(X, x_pre)


parameters = [{
        "gamma": np.linspace(0, 0.5, 20),
        "kernel": ["rbf", "linear", "poly"]
    }]

kpca=KernelPCA(remove_zero_eig=True, fit_inverse_transform=True, n_jobs=-1)
grid_search = GridSearchCV(estimator=kpca, param_grid=parameters, scoring=score)
grid_search.fit(x_train)
print(grid_search.best_params_)

```

Another hyperparameter that may be tuned with KPCA is the number of components to include. This approach is presented as an exercise in section 4. Using that approach, we use 100 components. Putting it all together, the KPCA achieves an accuracy of 99% on the Olivetti dataset.

```
from sklearn.metrics import accuracy_score

kpca = KernelPCA(n_components=100, kernel = 'rbf', gamma=0.02)
X_train = kpca.fit_transform(x_train)
X_test = kpca.transform(x_test)

clf = SVC()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)


print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))
```

4 Exercises

Exercise 1

Implement PCA from scratch with 2 components on the iris dataset. This is purely a revision exercise.

Exercise 2

Implement KPCA from scratch in python.

Exercise 3

The Sheffield (previously UMIST) Face Database contains 564 images of 20 individuals. Unlike the Olivetti databased used in the demonstration section, this database contains side and frontal views. A stub to load the data from OpenML.

Perform Linear PCA and Kernel PCA on this dataset and then use a classifier and compare the accuracy results.

```
from sklearn.datasets import fetch_openml
import numpy as np

umist = fetch_openml(data_id=41084)
images = umist.data.to_numpy()
labels = umist.target.to_numpy()
print(images.shape)

```

Exercise 4

A hyperparameter that was not optimised in the demonstration was the number of components to include in the KPCA. One rudimentary approach is to plot the eigenvalues of KPCA and compare their scale. We can discard the eigenvectors related to eigenvalues that are not very large.

Complete this exercise.

5 Solutions

Exercise 1

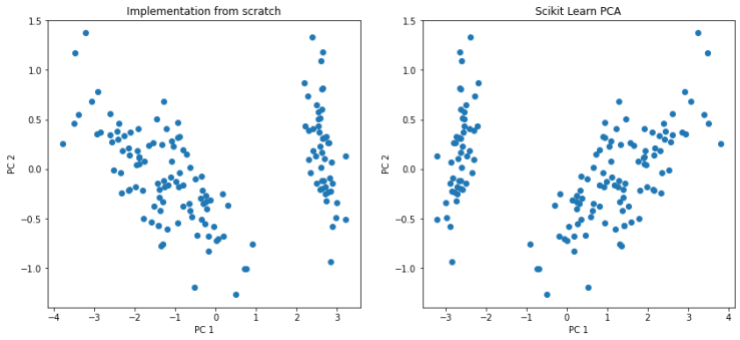

Fig 2: Comparison of PCA from scratch and sklearn PCA.

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from sklearn.decomposition import PCA
4.  from sklearn.datasets import load_iris
5.
6.  def pca_two(X, n_components=2):
7.
8.      # Presprocessing - Standard Scaler
9.      X_standard = X - np.mean(X, axis=0)
10.
11.     #Calculate covariance matrix
12.     cov_mat = np.cov(X_standard.T)
13.
14.     # Get eigenvalues and eigenvectors
15.     eig_vals, eig_vecs = np.linalg.eigh(cov_mat)
16.
17.     # Matrix of sorted eigenvectors.
18.     sorted_eigen = np.column_stack([eig_vecs[:,-i] for i in
    range(1,n_components+1)])
19.
20.     # Get the PCA reduced data
21.     Xpca = X_standard.dot(sorted_eigen)
22.
23.     return Xpca
24. # Sample
25. x, y = load_iris.data()
26. Xpca1 = pca_two(x, 2)
27. # Scikit-learn PCA
28. pca1 = PCA(n_components=2)
29. Xpca2 = pca1.fit_transform(x)
30.
31. fig, ax = plt.subplots(1, 2, figsize=(14, 6))
32.
33. ax[0].scatter(Xpca1[:,0], Xpca1[:,1])
34. ax[0].set_xlabel('PC 1')
35. ax[0].set_ylabel('PC 2')
36. ax[0].set_title('Implementation from scratch')
37. ax[1].scatter(Xpca2[:,0], Xpca2[:,1])
38. ax[1].set_xlabel('PC 1')
39. ax[1].set_ylabel('PC 2')
40. ax[1].set_title('Scikit Learn PCA')
```

```
41. plt.show()
```

Exercise 2

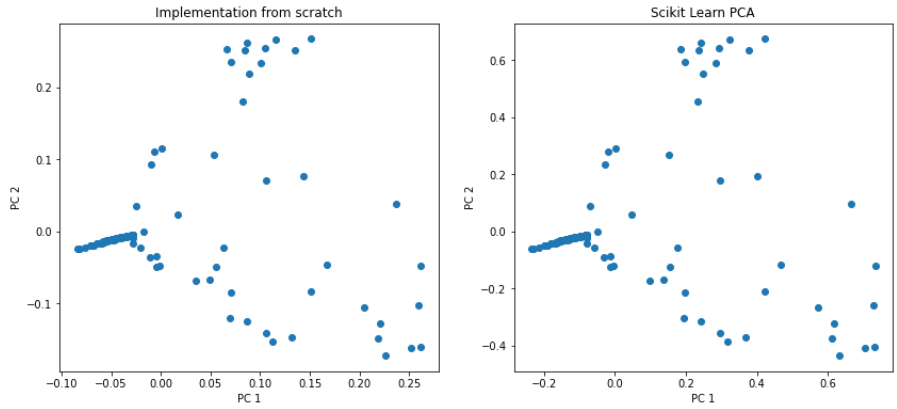

Fig3: KPCA comparison.

```
1.  import numpy as np
2.  import matplotlib.pyplot as plt
3.  from sklearn.decomposition import KernelPCA
4.  from sklearn.metrics.pairwise import euclidean_distances
5.  from sklearn.preprocessing import KernelCenterer
6.  from sklearn.datasets import load_iris
7.
8.  def KPCA(X, n_components=3, gamma=10):
9.
10.     # calculate distance between points in data set.
11.     dist = euclidean_distances(X, X, squared=True)
12.
13.     # calc gauss kernel matrix
14.     K = np.exp(-gamma * dist)
15.     Kc = KernelCenterer().fit_transform(K)
16.
17.     # Get eigenvalues and eigenvectors of the kernel matrix
18.     eig_vals, eig_vecs = np.linalg.eigh(Kc)
19.
20.     return np.column_stack([eig_vecs[:,-i] for i in range(1,n_components+1)])
21.
22. # use iris data
23. x = load_iris().data
24. Xpca1 = KPCA(x, 3)
25.
26. # Scikit-learn PCA
27. pca1 = KernelPCA(n_components=3, kernel='rbf', gamma=10)
28. Xpca2 = pca1.fit_transform(x)
29.
30. fig, ax = plt.subplots(1, 2, figsize=(14, 6))
31.
32. ax[0].scatter(Xpca1[:,0], Xpca1[:,1])
33. ax[0].set_xlabel('PC 1')
34. ax[0].set_ylabel('PC 2')
35. ax[0].set_title('Implementation from scratch')
36.
37. ax[1].scatter(Xpca2[:,0], Xpca2[:,1])
38. ax[1].set_xlabel('PC 1')
39. ax[1].set_ylabel('PC 2')
40. ax[1].set_title('Scikit Learn PCA')
41.
42. plt.show()
43.
```

Exercise 3

The results are discussed here briefly, the full code is available in Appendix B. The following classification accuracy rates were found in the sample solutions:

- Linear PCA – 99%
- Kernel PCA with a polynomial kernel – 98%
- Kernel PCA with tuned parameters by optimising for lowest reconstruction error – 10%
- Kernel PCA with tuned parameters by optimising for classification error – 99%

The purpose of this exercise was to show that the grid search method of tuning hyperparameters is not always ideal. Cross-validation via reconstruction error should always be cross-validated with the method of lowest classification error. Interestingly, the second grid search optimising for classification error resulted in a linear kernel being the best option for the dataset. KPCA with a linear kernel is simply standard PCA.

Exercise 4

```
1.  import matplotlib.pyplot as plt
2.  import numpy as np
3.  from sklearn.decomposition import KernelPCA
4.
5.  component_pca = KernelPCA(kernel='poly', gamma=0.001)
6.  component_pca.fit(x_train)
7.  eigen_values  = component_pca.eigenvalues_
8.
9.  fig, (ax1, ax2) = plt.subplots(1,2, figsize=(15,3))
10. x = np.linspace(0, len(eigen_values), len(eigen_values))
11. ax1.plot(x,eigen_values)
12. x2 = np.linspace(0, 50) # plot just first 50 components to zoom in
13. ax2.plot(x2, eigen_values[0:50])
14.
15. plt.show()
16.
```

SOURCES

Géron, A., & Demarest, R. (2019). Hands-on machine learning with Scikit-Learn and TensorFlow. Sebastopol (Clif.) [etc.]: O'Reilly.

Ming-Hsuan, Y. (2001). Face Recognition using Kernel Methods [Ebook]. Retrieved from https://proceedings.neurips.cc/paper/2001/file/4d6b3e38b952600251ee92fe603170ff-Paper.pdf

Pelliccia, D. (2010). PCA and kernel PCA explained • NIRPY Research. Retrieved 16 May 2022, from https://nirpyresearch.com/pca-kernel-pca-explained/

Wang, Q. (2014). Kernel Principal Component Analysis and its applications in face recognition and active shape models. Retrieved from https://arxiv.org/pdf/1207.3538.pdf

APPENDIX A

Full demonstration code.

```
1. import matplotlib.pyplot as plt
2. import sklearn.datasets
3. from sklearn.model_selection import train_test_split
4. from sklearn.decomposition import PCA
5. from sklearn.svm import SVC
6. from sklearn.metrics import accuracy_score
7. from sklearn.decomposition import KernelPCA
8. from sklearn.metrics import mean_squared_error
9. from sklearn.model_selection import GridSearchCV
10.
11.
12. olivetti = sklearn.datasets.fetch_olivetti_faces()
13. images = olivetti.data
14. faces = olivetti.images
15. labels = olivetti.target
16.
17. # Show what one face of subject looks liek.
18. def show_faces(images):
19.       fig, ax = plt.subplots(1,10, figsize=(15,7))
20.       id = 0
21.       for i in range(10):
22.         image_index = i
23.         ax[i].imshow(images[image_index], cmap="gray")
24.
25. show_faces(faces)
26. plt.show()
27.
28. # Split the data into training and testing data.
29. x_train, x_test, y_train, y_test = train_test_split(images, labels,
30.                                                    test_size=0.2,
    stratify=labels, random_state=1000)
31. # Complete standard PCA.
32. pca = PCA()
33. pca.fit(images)
34. x_pca = pca.transform(images)
35.
36. # Find the number of optimal components using explained variance graph.
37. pca_var = PCA()
38. pca_var.fit(images)
39.
40. plt.figure(1,figsize=(12,8))
41.
42. plt.plot(pca_var.explained_variance_ )
43. plt.xlabel("Components ")
44. plt.ylabel("Explained Variance")
45. plt.show()
46.
47. # optimal components = 50 from the graph
48. pca = PCA(n_components=50)
49. pca.fit(x_train)
50.
51. # Project training and testing data onto principal components.
52. X_train_pca = pca.transform(x_train)
53. X_test_pca = pca.transform(x_test)
54.
55. # Train classifier.
56. clf = SVC()
57. clf.fit(X_train_pca, y_train)
58. y_pred = clf.predict(X_test_pca)
59.
60.
61. print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))
62.
63. # Run initial kernel pca and get accuracy score.
```

```
kpca = KernelPCA(kernel = 'rbf')
X_train = kpca.fit_transform(x_train)
X_test = kpca.transform(x_test)

clf = SVC()
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
from sklearn.metrics import accuracy_score

print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))

# Run Grid search
# Code adapted from book: Hands-On Machine Learning with Scikit-Learn...
def score(estimator, X=x_train):
    x_red = estimator.transform(X)
    x_pre = estimator.inverse_transform(x_red)
    return -1 * mean_squared_error(X, x_pre)


parameters = [{
        "gamma": np.linspace(0, 0.5, 20),
        "kernel": ["rbf", "linear", "poly"]
    }]

kpca=KernelPCA(remove_zero_eig=True, fit_inverse_transform=True, n_jobs=-1)
grid_search = GridSearchCV(estimator=kpca, param_grid=parameters, scoring=score)
grid_search.fit(x_train)
print(grid_search.best_params_)

# Find optimal number of components by plotting eigenvalues.
component_pca = KernelPCA(kernel='poly', gamma=0.001)
component_pca.fit(x_train)
eigen_values  = component_pca.eigenvalues_

fig, (ax1, ax2) = plt.subplots(1,2, figsize=(15,3))
x = np.linspace(0, len(eigen_values), len(eigen_values))
 ax1.plot(x,eigen_values)
 x2 = np.linspace(0, 50) # plot just first 50 components to zoom in
 ax2.plot(x2, eigen_values[0:50])

 plt.show()

```

Using the hyperparameters from gridsearch (these are printed), and the number of components chosen to be used in the analysis, we run the final Kernel PCA.

```
 # Run final KPCA with hyperparameter values from the above code.
 kpca = KernelPCA(n_components=100, kernel = 'rbf', gamma=0.02)
 X_train = kpca.fit_transform(x_train)
 X_test = kpca.transform(x_test)

 clf = SVC()
 clf.fit(X_train, y_train)
 y_pred = clf.predict(X_test)

print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))

```

APPENDIX B

Code for Exercise 3

```
1.  import matplotlib.pyplot as plt
2.  import sklearn.datasets
3.  from sklearn.model_selection import train_test_split
4.  from sklearn.decomposition import PCA
5.  from sklearn.svm import SVC
6.  from sklearn.metrics import accuracy_score
7.  from sklearn.decomposition import KernelPCA
8.  from sklearn.metrics import mean_squared_error
9.  from sklearn.model_selection import GridSearchCV
10. from sklearn.datasets import fetch_openml
11. import numpy as np
12.
13. uminst = fetch_openml(data_id=41084)
14. images = uminst.data.to_numpy()
15. labels = uminst.target.to_numpy()
16.
17. # Show what one face of subject looks liek.
18. def show_faces(images):
19.       fig, ax = plt.subplots(1,10, figsize=(15,7))
20.       id = 0
21.       for i in range(10):
22.         image_index = i
23.         ax[i].imshow(images[image_index], cmap="gray")
24.
25. show_faces(faces)
26. plt.show()
27.
28. # Split the data into training and testing data.
29. x_train, x_test, y_train, y_test = train_test_split(images, labels,
30.                                                     test_size=0.2,
    stratify=labels, random_state=1000)
31.
32. # Complete standard PCA.
33. pca = PCA(n_components=2)
34. pca.fit(images)
35. x_pca = pca.transform(images)
36.
37. # Find the number of optimal components using explained variance graph.
38. pca_var = PCA()
39. pca_var.fit(images)
40.
41. plt.figure(1,figsize=(12,8))
42.
43. plt.plot(pca_var.explained_variance_ )
44. plt.xlabel("Components ")
45. plt.ylabel("Explained Variance")
46. plt.show()
47.
48. # optimal components = 50 from the graph
49. pca = PCA(n_components=50)
50. pca.fit(x_train)
51.
52. # Project training and testing data onto principal components.
53. X_train_pca = pca.transform(x_train)
54. X_test_pca = pca.transform(x_test)
55.
56. # Train classifier.
57. clf = SVC()
58. clf.fit(X_train_pca, y_train)
59. y_pred = clf.predict(X_test_pca)
60.
```

```
61.
62. print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))
63.
64. # Run initial kernel pca and get accuracy score.
65. kpca = KernelPCA(kernel = 'rbf')
66. X_train = kpca.fit_transform(x_train)
67. X_test = kpca.transform(x_test)
68.
69. clf = SVC()
70. clf.fit(X_train, y_train)
71. y_pred = clf.predict(X_test)
72. from sklearn.metrics import accuracy_score
73.
74. print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))
75.
76. # Run Grid search
77. # Code adapted from book: Hands-On Machine Learning with Scikit-Learn...
78. def score(estimator, X=x_train):
79.     x_red = estimator.transform(X)
80.     x_pre = estimator.inverse_transform(x_red)
81.     return -1 * mean_squared_error(X, x_pre)
82.
83.
84. parameters = [{
85.         "gamma": np.linspace(0, 0.5, 20),
86.         "kernel": ["rbf", "linear", "poly"]
87.     }]
88.
89. kpca=KernelPCA(remove_zero_eig=True, fit_inverse_transform=True, n_jobs=-1)
90. grid_search = GridSearchCV(estimator=kpca, param_grid=parameters, scoring=score)
91. grid_search.fit(x_train)
92. print(grid_search.best_params_)
93.
94. # Find optimal number of components by plotting eigenvalues.
95. component_pca = KernelPCA(kernel='poly', gamma=0.001)
96. component_pca.fit(x_train)
97. eigen_values  = component_pca.eigenvalues_
98.
99. fig, (ax1, ax2) = plt.subplots(1,2, figsize=(15,3))
100.  x = np.linspace(0, len(eigen_values), len(eigen_values))
101.  ax1.plot(x,eigen_values)
102.  x2 = np.linspace(0, 50) # plot just first 50 components to zoom in
103.  ax2.plot(x2, eigen_values[0:50])
104.
105.  plt.show()
```

Run KPCA using hyperparameeers from above.

```
1.  # Run final KPCA with hyperparameter values from the above code.
2.  kpca = KernelPCA(n_components=100, kernel = 'rbf', gamma=0.02)
3.  X_train = kpca.fit_transform(x_train)
4.  X_test = kpca.transform(x_test)
5.
6.  clf = SVC()
7.  clf.fit(X_train, y_train)
8.  y_pred = clf.predict(X_test)
9.
10. print("accuracy score:{:.2f}".format(accuracy_score(y_test, y_pred)))
```

Gridsearch using classification accuracy.

```
# Running a second type of grid search.
from sklearn.metrics import accuracy_score
def scorer(estimator, X=x_train, y=y_train):
    clf = SVC()
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    return accuracy_score(y_test, y_pred)


param_grid = [{
        "gamma": np.linspace(0, 0.5, 20),
        "kernel": ["rbf", "linear", "poly"]
    }]

kpca=KernelPCA(remove_zero_eig=True, fit_inverse_transform=True, n_jobs=-1)
grid_search = GridSearchCV(kpca, param_grid, cv=3, scoring=scorer)
grid_search.fit(x_train)

print(grid_search.best_params_)

```