Mathematics and Big Data

Han Jiatao Jerry
Raffles Institution
27/6/2017

Abstract

"Sensors, sensors everywhere." The Smart Nation initiative looks to turn the island into a "living laboratory". Part of that plan is a network of sensors placed across the island that officials hope can solve many fundamental problems.

How can we then utilise the data collected effectively and generate profound insights? Data visualisation is an important part of the process, as it helps to transform "big data" into a form that is easily comprehended by the human mind. Two techniques involved are clustering and dimensionality reduction. Though these visualisation methods are implemented in computers, they also have their foundation in mathematics.

ESSAY

Note: The implementation of these algorithms discussed can be found in the Appendices.

Clustering

Clustering is the process of grouping similar data points. This allows us to identify intrinsic groups in unlabelled data.

Some real world examples include:

Field/Domain	Purpose
Marketing	Identifying customers with similar behaviour
Business	Identifying ideal locations for cell-phone towers
Biology	Classifying plants/animals
Seismology	Identifying danger zones by grouping earthquake epicentres and health care
Health care	Identifying ideal locations of emergency wards

We will be examining K-means clustering. In mathematical terms, given a set of observations $(x_1, \ldots x_n)$, where each observation is a d-dimensional real vector, we aim to partition the n observations into $k \leq n$ sets $S = \{S_1, S_2, \ldots, S_k\}$ so as to minimise the within-cluster sum of squares (i.e. the sum of distance functions of each point in the cluster to the K centre). Thus, the objective of K means clustering is to find:

$$\underset{\mathbf{S}}{\arg\min} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} |\mathbf{x} - \mu_i|^2$$

Where μ_i is the mean of points in S_i. [1]

To compute this, we can use Lloyd's algorithm. Given an initial set of k means $m_1, m_2, \ldots, m_k$, the algorithm proceeds by alternating between two steps [2][3]

Assignment step

Assign each observation to the cluster whose mean yields the least within-cluster sum of squares. Mathematically, this means partitioning the observations according to the Voronoi diagram generated by the means.

$$S_i^{(t)} = \{x_p: \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \forall j, 1 \leq j \leq k\}$$

Where each x_p is assigned to exactly one $S^{(t)}$, even if it could be assigned to two or more of them.

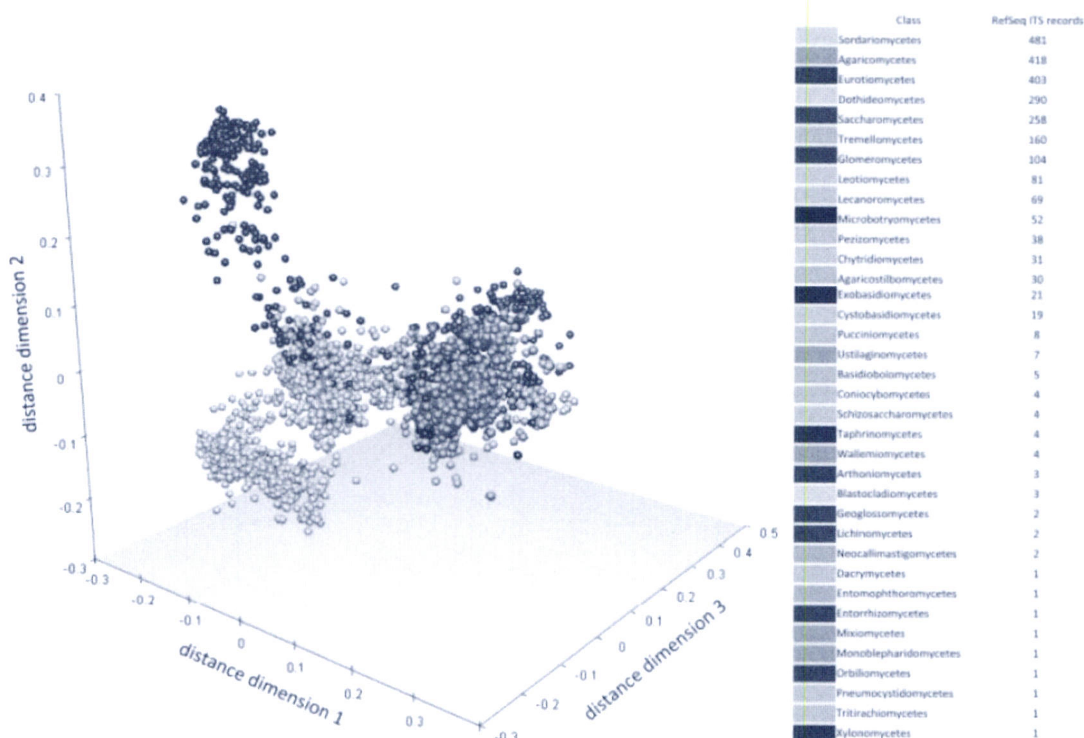
Update step:

Calculate the new means to be the centroids of the observations in the new clusters

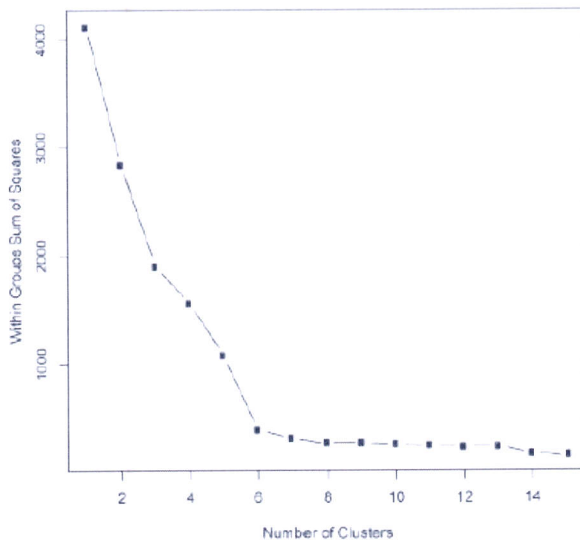
$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

Since the arithmetic mean is a least-squares estimator, this also minimizes the within-cluster sum of squares objective.

The algorithm will converge when the assignments no longer change. Since both steps optimize the WCSS objective, and there only exits a finite number of such partitionings, the algorithm must converge to a local optimum. There is no guarantee though that the global optimum can be found using this algorithm. By colouring each cluster a unique colour and plotting it out, a visualisation can be created:



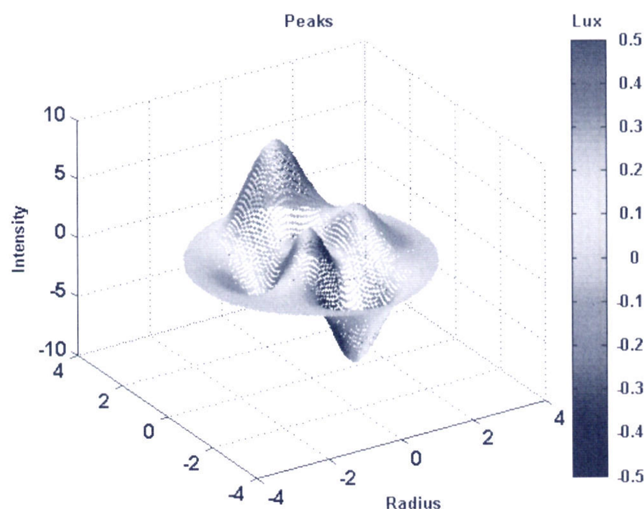
Of course, we are not able to predetermine the optimal number of clusters. To determine the optimum value of k, we can use the elbow method. [4] We plot the distortion (within-cluster sum of squares) for each value of k, as shown in the graph below



We then choose the k on the elbow, in this case it would be 6. This indicates that there are most likely 6 groups within this data set. Following which, we use the K means algorithm to assign each data point to a group and plot it out.

Dimensionality Reduction

Sometimes data are collected on a large number of variables from a single population. With a large number of variables, the dispersion (covariance) matrix which is used in analysis of the data frequently may be too large to study and interpret properly. There would be too many pairwise correlations between the variables to consider. Graphical display of data may also not be of particular help in case the data set is very large. Consider plotting a graph of 4 or more dimensions. The complexity of the graph would definitely be beyond the comprehension of a human being! Just try visualising this in your head:



I am sure that you have been left rather confused. This then entirely defeats the purpose of the graph as a visualisation tool. Even if we were to plot three variables at a

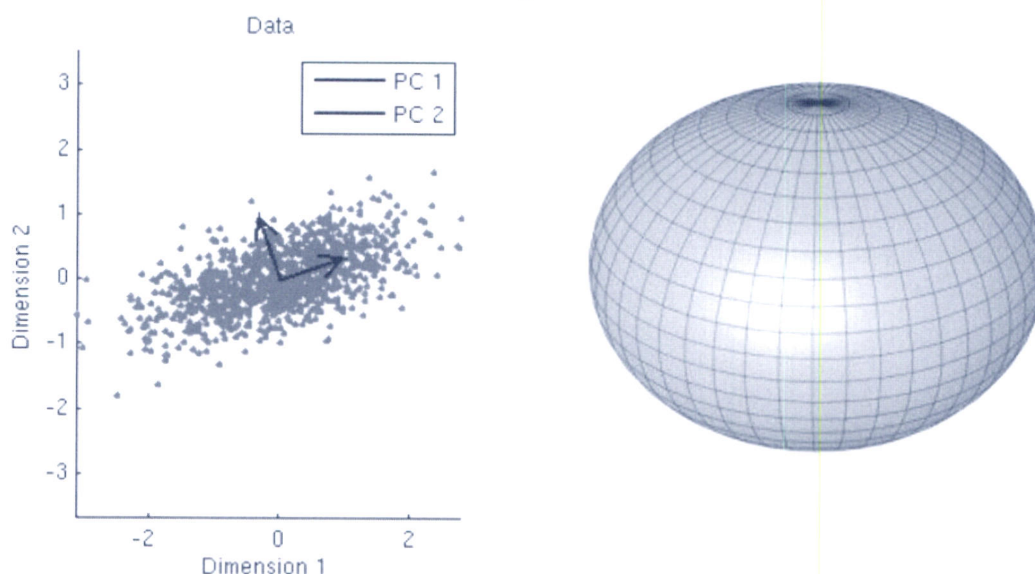
time, there are still problems. With 12 variables, for example, there will be more than 200 three-dimensional scatterplots to be studied!

To interpret the data in a more meaningful form, it is therefore necessary to reduce the number of variables to a few, interpretable linear combinations of the data. Each linear combination will correspond to a principal component. This is done through Principal Components Analysis.

Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. [5]

PCA can be thought of as fitting an n-dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipsoid is small, then the variance along that axis is also small, and by omitting that axis and its corresponding principal component from our representation of the dataset, we lose only a commensurately small amount of information.

This process can be visualised in this diagram:



In layman's terms, we are simply trying to fit the ellipsoid you see on the right to our data you see on the left. Then we take the axes of the ellipsoid which are significant.

To find the axes of the ellipsoid, we must first subtract the mean of each variable from the dataset to centre the data around the origin. Then, we compute the covariance matrix of the data, and calculate the eigenvalues and corresponding eigenvectors of this covariance matrix. Then, we must orthogonalize the set of eigenvectors, and normalize each to become unit vectors. Once this is done, each of the mutually orthogonal, unit eigenvectors can be interpreted as an axis of the ellipsoid fitted to the data. The proportion of the variance that each eigenvector represents can be calculated by dividing the eigenvalue corresponding to that eigenvector by the sum of all eigenvalues.

PCA is mathematically defined as an orthogonal linear transformation that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

The mathematical procedure is as such: [6]

Suppose that we have a random vector $\mathbf{X}$.

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \\ \vdots \\ X_p \end{pmatrix}$$

with population variance-covariance matrix

$$\text{var}(\mathbf{X}) = \Sigma = \begin{pmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1p} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{p1} & \sigma_{p2} & \cdots & \sigma_p^2 \end{pmatrix}$$

Consider the linear combinations

$$\begin{aligned} Y_1 &= e_{11}X_1 + e_{12}X_2 + \cdots + e_{1p}X_p \\ Y_2 &= e_{21}X_1 + e_{22}X_2 + \cdots + e_{2p}X_p \\ &\vdots \\ Y_p &= e_{p1}X_1 + e_{p2}X_2 + \cdots + e_{pp}X_p \end{aligned}$$

Each of these can be thought of as a linear regression, predicting Y_i from $X_1, X_2, \ldots, X_p$. There is no intercept, but $e_{i1}, e_{i2}, \ldots, e_{ip}$ can be viewed as regression coefficients.

Note that Y_i is a function of our random data, and so is also random. Therefore it has a population variance

$$\text{var}(Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{il}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_i$$

Moreover, Y_i and Y_j will have a population covariance

$$\text{cov}(Y_i, Y_j) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{jl}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_j$$

Here the coefficients e_{ij} are collected into the vector

$$\mathbf{e}_i = \begin{pmatrix} e_{i1} \\ e_{i2} \\ \vdots \\ e_{ip} \end{pmatrix}$$

First Principal Component (Y_1)

We select $e_{11}, e_{12}, \ldots, e_{1p}$ that maximizes

$$\text{var}(Y_1) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{1k}e_{1l}\sigma_{kl} = \mathbf{e}_1'\Sigma\mathbf{e}_1$$

subject to the constraint that

$$\mathbf{e}_1'\mathbf{e}_1 = \sum_{j=1}^{p} e_{1j}^2 = 1$$

Ith Principal Component (Y_i)

We select $e_{11}, e_{12}, \ldots, e_{1p}$ that maximizes

$$\text{var}(Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{ik}e_{il}\sigma_{kl} = \mathbf{e}_i'\Sigma\mathbf{e}_i$$

subject to the constraint that the sums of squared coefficients add up to one, along with the additional constraint that this new component will be uncorrelated with all the previously defined components.

$$\mathbf{e}_i'\mathbf{e}_i = \sum_{j=1}^{p} e_{ij}^2 = 1$$

$$\text{cov}(Y_1, Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{1k}e_{il}\sigma_{kl} = \mathbf{e}_1'\Sigma\mathbf{e}_i = 0,$$

$$\text{cov}(Y_2, Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{2k}e_{il}\sigma_{kl} = \mathbf{e}_2'\Sigma\mathbf{e}_i = 0,$$

$$\vdots$$

$$\text{cov}(Y_{i-1}, Y_i) = \sum_{k=1}^{p}\sum_{l=1}^{p} e_{i-1,k}e_{il}\sigma_{kl} = \mathbf{e}_{i-1}'\Sigma\mathbf{e}_i = 0$$

This will give us principal components that are uncorrelated with one another.

After we simplify our data set, clustering can then be used to provide interpretable visualisations for insights to be gained.

Conclusion

The purpose of clustering and classification algorithms is to make sense of and extract value from large sets of structured and unstructured data. If one is working with huge volumes of unstructured data, it only makes sense to try to partition the data into some sort of logical groupings before attempting to analyse it.

Clustering and classification allows you to take a sweeping glance of your data en masse, and then form some logical structures based on what you find there before going deeper into the nuts-and-bolts analysis.

At the same time, to facilitate the process of data analysis, dimensionality reduction algorithms can also be applied. There are many advantages, such as reduced time and storage space needed, improvement of machine-learning models due to removal of multi-collinearity as well as greater ease of visualisation.

All in all, these two tools with their foundations rooted in mathematics do come in very handy for data scientists.

Appendix 1 – An Implementation of the K Means Algorithm in Python

```
import numpy as np
import scipy.cluster
data = np.array('''your data points here''')

data = scipy.cluster.vq.whiten(data)

for i in range(1,10): #iterates all possible number of clusters up till a reasonable limit
  codebook,distortion = scipy.cluster.vq.kmeans(data,i)

#You can then plot out the distortion graph to apply the elbow method
#Afterwards, choose the optimal value of k.
codebook,distortion = scipy.cluster.vq.kmeans(data,2) #Suppose 2 is the optimal value in this case
index, distarray = scipy.cluster.vq.vq(data,codebook)

#the first value the vq function returns is an ndarray that contains the codebook index for each data point,
#and the second value returned is the distortion array (showing distortion values for individual observations).

#index contains the information regarding the cluster centroids whilst
#codebook contains the information regarding each data point and which cluster they are assigned to
```

Source code for ‘whiten’, ‘kmeans’ and ‘vq’: [7]

```
1.  import warnings
2.
3.  import numpy as np
4.  from scipy._lib._util import _asarray_validated
5.  from scipy._lib import _numpy_compat
6.
7.  from . import _vq
8.
9.
10. class ClusterError(Exception):
11.     pass
12.
13.
14. def whiten(obs, check_finite=True):
15.     """
16.     Normalize a group of observations on a per feature basis.
17.     Before running k-means, it is beneficial to rescale each feature
18.     dimension of the observation set with whitening. Each feature is
19.     divided by its standard deviation across all observations to give
20.     it unit variance.
21.     Parameters
22.     ----------
23.     obs : ndarray
24.         Each row of the array is an observation.  The
25.         columns are the features seen during each observation.
26.         >>> #         f0    f1    f2
27.         >>> obs = [[  1.,   1.,   1.],  #o0
28.         ...        [  2.,   2.,   2.],  #o1
29.         ...        [  3.,   3.,   3.],  #o2
30.         ...        [  4.,   4.,   4.]]  #o3
31.     check_finite : bool, optional
32.         Whether to check that the input matrices contain only finite numbers.
33.         Disabling may give a performance gain, but may result in problems
34.         (crashes, non-termination) if the inputs do contain infinities or NaNs.
35.         Default: True
36.     Returns
37.     -------
38.     result : ndarray
39.         Contains the values in `obs` scaled by the standard deviation
40.         of each column.
41.     Examples
42.     --------
43.     >>> from scipy.cluster.vq import whiten
44.     >>> features  = np.array([[1.9, 2.3, 1.7],
45.     ...                       [1.5, 2.5, 2.2],
46.     ...                       [0.8, 0.6, 1.7,]])
47.     >>> whiten(features)
48.     array([[ 4.17944278,  2.69811351,  7.21248917],
```

```
49.            [ 3.29956009,  2.93273208,  9.33380951],
50.            [ 1.75976538,  0.7038557 ,  7.21248917]])
51.     """
52.     obs = _asarray_validated(obs, check_finite=check_finite)
53.     std_dev = np.std(obs, axis=0)
54.     zero_std_mask = std_dev == 0
55.     if zero_std_mask.any():
56.         std_dev[zero_std_mask] = 1.0
57.         warnings.warn("Some columns have standard deviation zero. "
58.                       "The values of these columns will not change.",
59.                       RuntimeWarning)
60.     return obs / std_dev
61.
62.
63. def vq(obs, code_book, check_finite=True):
64.     """
65.     Assign codes from a code book to observations.
66.     Assigns a code from a code book to each observation. Each
67.     observation vector in the 'M' by 'N' `obs` array is compared with the
68.     centroids in the code book and assigned the code of the closest
69.     centroid.
70.     The features in `obs` should have unit variance, which can be
71.     achieved by passing them through the whiten function.  The code
72.     book can be created with the k-means algorithm or a different
73.     encoding algorithm.
74.     Parameters
75.     ----------
76.     obs : ndarray
77.         Each row of the 'M' x 'N' array is an observation.  The columns are
78.         the "features" seen during each observation. The features must be
79.         whitened first using the whiten function or something equivalent.
80.     code_book : ndarray
81.         The code book is usually generated using the k-means algorithm.
82.         Each row of the array holds a different code, and the columns are
83.         the features of the code.
84.          >>> #              f0     f1     f2    f3
85.          >>> code_book = [
86.          ...             [  1.,    2.,    3.,    4.],  #c0
87.          ...             [  1.,    2.,    3.,    4.],  #c1
88.          ...             [  1.,    2.,    3.,    4.]]  #c2
89.     check_finite : bool, optional
90.         Whether to check that the input matrices contain only finite numbers.
91.         Disabling may give a performance gain, but may result in problems
92.         (crashes, non-termination) if the inputs do contain infinities or NaNs.
93.         Default: True
94.     Returns
95.     -------
96.     code : ndarray
97.         A length M array holding the code book index for each observation.
98.     dist : ndarray
99.         The distortion (distance) between the observation and its nearest
100.            code.
101.        Examples
102.        --------
103.        >>> from numpy import array
104.        >>> from scipy.cluster.vq import vq
105.        >>> code_book = array([[1.,1.,1.],
106.        ...                    [2.,2.,2.]])
107.        >>> features  = array([[  1.9,2.3,1.7],
108.        ...                    [  1.5,2.5,2.2],
109.        ...                    [  0.8,0.6,1.7]])
110.        >>> vq(features,code_book)
111.        (array([1, 1, 0],'i'), array([ 0.43588989,  0.73484692,  0.83066239]))
112.        """
113.        obs = _asarray_validated(obs, check_finite=check_finite)
114.        code_book = _asarray_validated(code_book, check_finite=check_finite)
```

```
115.            ct = np.common_type(obs, code_book)
116.
117.            c_obs = obs.astype(ct, copy=False)
118.
119.            if code_book.dtype != ct:
120.                c_code_book = code_book.astype(ct)
121.            else:
122.                c_code_book = code_book
123.
124.            if ct in (np.float32, np.float64):
125.                results = _vq.vq(c_obs, c_code_book)
126.            else:
127.                results = py_vq(obs, code_book)
128.            return results
129.
130.        def _kmeans(obs, guess, thresh=1e-5):
131.            """ "raw" version of k-means.
132.            Returns
133.            -------
134.            code_book
135.                the lowest distortion codebook found.
136.            avg_dist
137.                the average distance a observation is from a code in the book.
138.                Lower means the code_book matches the data better.
139.            See Also
140.            --------
141.            kmeans : wrapper around k-means
142.            Examples
143.            --------
144.            Note: not whitened in this example.
145.            >>> from numpy import array
146.            >>> from scipy.cluster.vq import _kmeans
147.            >>> features  = array([[ 1.9,2.3],
148.            ...                    [ 1.5,2.5],
149.            ...                    [ 0.8,0.6],
150.            ...                    [ 0.4,1.8],
151.            ...                    [ 1.0,1.0]])
152.            >>> book = array((features[0],features[2]))
153.            >>> _kmeans(features,book)
154.            (array([[ 1.7       ,  2.4       ],
155.                   [ 0.73333333,  1.13333333]]), 0.40563916697728591)
156.            """
157.
158.            code_book = np.array(guess, copy=True)
159.            avg_dist = []
160.            diff = np.inf
161.            while diff > thresh:
162.                nc = code_book.shape[0]
163.                # compute membership and distances between obs and code_book
164.                obs_code, distort = vq(obs, code_book)
165.                avg_dist.append(np.mean(distort, axis=-1))
166.                # recalc code_book as centroids of associated obs
167.                if(diff > thresh):
168.                    code_book, has_members = _vq.update_cluster_means(obs, obs_code,
    nc)
169.                    code_book = code_book.compress(has_members, axis=0)
170.                if len(avg_dist) > 1:
171.                    diff = avg_dist[-2] - avg_dist[-1]
172.
173.            return code_book, avg_dist[-1]
174.
175.
176.        def kmeans(obs, k_or_guess, iter=20, thresh=1e-5, check_finite=True):
177.            """
178.            Performs k-means on a set of observation vectors forming k clusters.
179.            The k-means algorithm adjusts the centroids until sufficient
```

```
180.            progress cannot be made, i.e. the change in distortion since
181.            the last iteration is less than some threshold. This yields
182.            a code book mapping centroids to codes and vice versa.
183.            Distortion is defined as the sum of the squared differences
184.            between the observations and the corresponding centroid.
185.            Parameters
186.            ----------
187.            obs : ndarray
188.               Each row of the M by N array is an observation vector. The
189.               columns are the features seen during each observation.
190.               The features must be whitened first with the `whiten` function.
191.            k_or_guess : int or ndarray
192.               The number of centroids to generate. A code is assigned to
193.               each centroid, which is also the row index of the centroid
194.               in the code_book matrix generated.
195.               The initial k centroids are chosen by randomly selecting
196.               observations from the observation matrix. Alternatively,
197.               passing a k by N array specifies the initial k centroids.
198.            iter : int, optional
199.               The number of times to run k-means, returning the codebook
200.               with the lowest distortion. This argument is ignored if
201.               initial centroids are specified with an array for the
202.               ``k_or_guess`` parameter. This parameter does not represent the
203.               number of iterations of the k-means algorithm.
204.            thresh : float, optional
205.               Terminates the k-means algorithm if the change in
206.               distortion since the last k-means iteration is less than
207.               or equal to thresh.
208.            check_finite : bool, optional
209.                Whether to check that the input matrices contain only finite numbers
    .
210.                Disabling may give a performance gain, but may result in problems
211.                (crashes, non-
    termination) if the inputs do contain infinities or NaNs.
212.                Default: True
213.            Returns
214.            -------
215.            codebook : ndarray
216.               A k by N array of k centroids. The i'th centroid
217.               codebook[i] is represented with the code i. The centroids
218.               and codes generated represent the lowest distortion seen,
219.               not necessarily the globally minimal distortion.
220.            distortion : float
221.               The distortion between the observations passed and the
222.               centroids generated.
223.            See Also
224.            --------
225.            kmeans2 : a different implementation of k-means clustering
226.               with more methods for generating initial centroids but without
227.               using a distortion change threshold as a stopping criterion.
228.            whiten : must be called prior to passing an observation matrix
229.               to kmeans.
230.            Examples
231.            --------
232.            >>> from numpy import array
233.            >>> from scipy.cluster.vq import vq, kmeans, whiten
234.            >>> import matplotlib.pyplot as plt
235.            >>> features  = array([[ 1.9,2.3],
236.            ...                    [ 1.5,2.5],
237.            ...                    [ 0.8,0.6],
238.            ...                    [ 0.4,1.8],
239.            ...                    [ 0.1,0.1],
240.            ...                    [ 0.2,1.8],
241.            ...                    [ 2.0,0.5],
242.            ...                    [ 0.3,1.5],
243.            ...                    [ 1.0,1.0]])
```

```
244.            >>> whitened = whiten(features)
245.            >>> book = np.array((whitened[0],whitened[2]))
246.            >>> kmeans(whitened,book)
247.            (array([[ 2.3110306 ,  2.86287398],    # random
248.                   [ 0.93218041,  1.24398691]]), 0.85684700941625547)
249.            >>> from numpy import random
250.            >>> random.seed((1000,2000))
251.            >>> codes = 3
252.            >>> kmeans(whitened,codes)
253.            (array([[ 2.3110306 ,  2.86287398],    # random
254.                   [ 1.32544402,  0.65607529],
255.                   [ 0.40782893,  2.02786907]]), 0.5196582527686241)
256.            >>> # Create 50 datapoints in two clusters a and b
257.            >>> pts = 50
258.            >>> a = np.random.multivariate_normal([0, 0], [[4, 1], [1, 4]], size=pts
    )
259.            >>> b = np.random.multivariate_normal([30, 10],
260.            ...                                   [[10, 2], [2, 1]],
261.            ...                                   size=pts)
262.            >>> features = np.concatenate((a, b))
263.            >>> # Whiten data
264.            >>> whitened = whiten(features)
265.            >>> # Find 2 clusters in the data
266.            >>> codebook, distortion = kmeans(whitened, 2)
267.            >>> # Plot whitened data and cluster centers in red
268.            >>> plt.scatter(whitened[:, 0], whitened[:, 1])
269.            >>> plt.scatter(codebook[:, 0], codebook[:, 1], c='r')
270.            >>> plt.show()
271.            """
272.            obs = _asarray_validated(obs, check_finite=check_finite)
273.            if int(iter) < 1:
274.                raise ValueError('iter must be at least 1.')
275.
276.            # Determine whether a count (scalar) or an initial guess (array) was pas
    sed.
277.            k = None
278.            guess = None
279.            try:
280.                k = int(k_or_guess)
281.            except TypeError:
282.                guess = _asarray_validated(k_or_guess, check_finite=check_finite)
283.
284.            if guess is not None:
285.                if guess.size < 1:
286.                    raise ValueError("Asked for 0 cluster ? initial book was %s" %
287.                                     guess)
288.                result = _kmeans(obs, guess, thresh=thresh)
289.            else:
290.                if k != k_or_guess:
291.                    raise ValueError('if k_or_guess is a scalar, it must be an integ
    er')
292.                # initialize best distance value to a large value
293.                best_dist = np.inf
294.                No = obs.shape[0]
295.                k = k_or_guess
296.                if k < 1:
297.                    raise ValueError("Asked for 0 cluster ? ")
298.                for i in range(iter):
299.                    # the initial code book is randomly selected from observations
300.                    k_random_indices = np.random.randint(0, No, k)
301.                    if np.any(_numpy_compat.unique(k_random_indices,
302.                                                   return_counts=True)[1] > 1):
303.                        # randint can give duplicates, which is incorrect.  Only fix

304.                        # the issue if it occurs, to not change results for users wh
    o
```

```
305.                    #  use a random seed and get no duplicates.
306.                    k_random_indices = np.random.permutation(No)[:k]
307.
308.                guess = np.take(obs, k_random_indices, 0)
309.                book, dist = _kmeans(obs, guess, thresh=thresh)
310.                if dist < best_dist:
311.                    best_book = book
312.                    best_dist = dist
313.            result = best_book, best_dist
314.        return result
```

Appendix 2 – An Implementation of the PCA Algorithm in Python

```
1.  import pandas as pd
2.  from sklearn import PCA
3.  df = #dataframe of your choice
4.  df.columns = #list of columns (n columns)
5.  # split data table into data and class labels
6.  data = df.ix[:,0:n-1].values
7.  labels = df.ix[:,n-1].values
8.  #Now to perform PCA:
9.  pca = PCA(n_components= #choose the number of components, whiten=True)
10. pca.fit(data)
11. #now we can look at the attributes of our pca object
12. print(pca.components_)
13. print(explained_variance_ratio_)
```

Source code for PCA algorithm: [8]

```
1.  """ Principal Component Analysis
2.  """
3.
4.  # Author: Alexandre Gramfort <alexandre.gramfort@inria.fr>
5.  #         Olivier Grisel <olivier.grisel@ensta.org>
6.  #         Mathieu Blondel <mathieu@mblondel.org>
7.  #         Denis A. Engemann <denis-alexander.engemann@inria.fr>
8.  #         Michael Eickenberg <michael.eickenberg@inria.fr>
9.  #         Giorgio Patrini <giorgio.patrini@anu.edu.au>
10. #
11. # License: BSD 3 clause
12.
13. from math import log, sqrt
14.
15. import numpy as np
16. from scipy import linalg
17. from scipy.special import gammaln
18. from scipy.sparse import issparse
19.
20. from ..externals import six
21.
22. from .base import _BasePCA
23. from ..base import BaseEstimator, TransformerMixin
24. from ..utils import deprecated
25. from ..utils import check_random_state, as_float_array
26. from ..utils import check_array
27. from ..utils.extmath import fast_dot, fast_logdet, randomized_svd, svd_flip
28. from ..utils.extmath import stable_cumsum
29. from ..utils.validation import check_is_fitted
30. from ..utils.arpack import svds
31.
32.
33. def _assess_dimension_(spectrum, rank, n_samples, n_features):
34.     """Compute the likelihood of a rank ``rank`` dataset
35.     The dataset is assumed to be embedded in gaussian noise of shape(n,
```

```
36.     dimf) having spectrum ``spectrum``.
37.     Parameters
38.     ----------
39.     spectrum : array of shape (n)
40.         Data spectrum.
41.     rank : int
42.         Tested rank value.
43.     n_samples : int
44.         Number of samples.
45.     n_features : int
46.         Number of features.
47.     Returns
48.     -------
49.     ll : float,
50.         The log-likelihood
51.     Notes
52.     -----
53.     This implements the method of `Thomas P. Minka:
54.     Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604`
55.     """
56.     if rank > len(spectrum):
57.         raise ValueError("The tested rank cannot exceed the rank of the"
58.                          " dataset")
59.
60.     pu = -rank * log(2.)
61.     for i in range(rank):
62.         pu += (gammaln((n_features - i) / 2.) -
63.                log(np.pi) * (n_features - i) / 2.)
64.
65.     pl = np.sum(np.log(spectrum[:rank]))
66.     pl = -pl * n_samples / 2.
67.
68.     if rank == n_features:
69.         pv = 0
70.         v = 1
71.     else:
72.         v = np.sum(spectrum[rank:]) / (n_features - rank)
73.         pv = -np.log(v) * n_samples * (n_features - rank) / 2.
74.
75.     m = n_features * rank - rank * (rank + 1.) / 2.
76.     pp = log(2. * np.pi) * (m + rank + 1.) / 2.
77.
78.     pa = 0.
79.     spectrum_ = spectrum.copy()
80.     spectrum_[rank:n_features] = v
81.     for i in range(rank):
82.         for j in range(i + 1, len(spectrum)):
83.             pa += log((spectrum[i] - spectrum[j]) *
84.                       (1. / spectrum_[j] - 1. / spectrum_[i])) + log(n_samples)
85.
86.     ll = pu + pl + pv + pp - pa / 2. - rank * log(n_samples) / 2.
87.
88.     return ll
89.
90.
91. def _infer_dimension_(spectrum, n_samples, n_features):
92.     """Infers the dimension of a dataset of shape (n_samples, n_features)
93.     The dataset is described by its spectrum `spectrum`.
94.     """
95.     n_spectrum = len(spectrum)
96.     ll = np.empty(n_spectrum)
97.     for rank in range(n_spectrum):
98.         ll[rank] = _assess_dimension_(spectrum, rank, n_samples, n_features)
99.     return ll.argmax()
100.
101.
```

```
102.        class PCA(_BasePCA):
103.            """Principal component analysis (PCA)
104.            Linear dimensionality reduction using Singular Value Decomposition of th
    e
105.            data to project it to a lower dimensional space.
106.            It uses the LAPACK implementation of the full SVD or a randomized trunca
    ted
107.            SVD by the method of Halko et al. 2009, depending on the shape of the in
    put
108.            data and the number of components to extract.
109.            It can also use the scipy.sparse.linalg ARPACK implementation of the
110.            truncated SVD.
111.            Notice that this class does not support sparse input. See
112.            :class:`TruncatedSVD` for an alternative with sparse data.
113.            Read more in the :ref:`User Guide <PCA>`.
114.            Parameters
115.            ----------
116.            n_components : int, float, None or string
117.                Number of components to keep.
118.                if n_components is not set all components are kept::
119.                    n_components == min(n_samples, n_features)
120.                if n_components == 'mle' and svd_solver == 'full', Minka\'s MLE is u
    sed
121.                to guess the dimension
122.                if ``0 < n_components < 1`` and svd_solver == 'full', select the num
    ber
123.                of components such that the amount of variance that needs to be
124.                explained is greater than the percentage specified by n_components
125.                n_components cannot be equal to n_features for svd_solver == 'arpack
    '.
126.            copy : bool (default True)
127.                If False, data passed to fit are overwritten and running
128.                fit(X).transform(X) will not yield the expected results,
129.                use fit_transform(X) instead.
130.            whiten : bool, optional (default False)
131.                When True (False by default) the `components_` vectors are multiplie
    d
132.                by the square root of n_samples and then divided by the singular val
    ues
133.                to ensure uncorrelated outputs with unit component-wise variances.
134.                Whitening will remove some information from the transformed signal
135.                (the relative variance scales of the components) but can sometime
136.                improve the predictive accuracy of the downstream estimators by
137.                making their data respect some hard-wired assumptions.
138.            svd_solver : string {'auto', 'full', 'arpack', 'randomized'}
139.                auto :
140.                    the solver is selected by a default policy based on `X.shape` an
    d
141.                    `n_components`: if the input data is larger than 500x500 and the

142.                    number of components to extract is lower than 80% of the smalles
    t
143.                    dimension of the data, then the more efficient 'randomized'
144.                    method is enabled. Otherwise the exact full SVD is computed and
145.                    optionally truncated afterwards.
146.                full :
147.                    run exact full SVD calling the standard LAPACK solver via
148.                    `scipy.linalg.svd` and select the components by postprocessing
149.                arpack :
150.                    run SVD truncated to n_components calling ARPACK solver via
151.                    `scipy.sparse.linalg.svds`. It requires strictly
152.                    0 < n_components < X.shape[1]
153.                randomized :
154.                    run randomized SVD by the method of Halko et al.
155.                .. versionadded:: 0.18.0
156.            tol : float >= 0, optional (default .0)
```

```
157.                Tolerance for singular values computed by svd_solver == 'arpack'.
158.                .. versionadded:: 0.18.0
159.            iterated_power : int >= 0, or 'auto', (default 'auto')
160.                Number of iterations for the power method computed by
161.                svd_solver == 'randomized'.
162.                .. versionadded:: 0.18.0
163.            random_state : int, RandomState instance or None, optional (default None
    )
164.                If int, random_state is the seed used by the random number generator
    ;
165.                If RandomState instance, random_state is the random number generator
    ;
166.                If None, the random number generator is the RandomState instance use
    d
167.                by `np.random`. Used when ``svd_solver`` == 'arpack' or 'randomized'
    .
168.                .. versionadded:: 0.18.0
169.            Attributes
170.            ----------
171.            components_ : array, shape (n_components, n_features)
172.                Principal axes in feature space, representing the directions of
173.                maximum variance in the data. The components are sorted by
174.                ``explained_variance_``.
175.            explained_variance_ : array, shape (n_components,)
176.                The amount of variance explained by each of the selected components.

177.                .. versionadded:: 0.18
178.            explained_variance_ratio_ : array, shape (n_components,)
179.                Percentage of variance explained by each of the selected components.

180.                If ``n_components`` is not set then all components are stored and th
    e
181.                sum of explained variances is equal to 1.0.
182.            singular_values_ : array, shape (n_components,)
183.                The singular values corresponding to each of the selected components
    .
184.                The singular values are equal to the 2-
    norms of the ``n_components``
185.                variables in the lower-dimensional space.
186.            mean_ : array, shape (n_features,)
187.                Per-feature empirical mean, estimated from the training set.
188.                Equal to `X.mean(axis=1)`.
189.            n_components_ : int
190.                The estimated number of components. When n_components is set
191.                to 'mle' or a number between 0 and 1 (with svd_solver == 'full') thi
    s
192.                number is estimated from input data. Otherwise it equals the paramet
    er
193.                n_components, or n_features if n_components is None.
194.            noise_variance_ : float
195.                The estimated noise covariance following the Probabilistic PCA model

196.                from Tipping and Bishop 1999. See "Pattern Recognition and
197.                Machine Learning" by C. Bishop, 12.2.1 p. 574 or
198.                http://www.miketipping.com/papers/met-mppca.pdf. It is required to
199.                computed the estimated data covariance and score samples.
200.            References
201.            ----------
202.            For n_components == 'mle', this class uses the method of `Thomas P. Mink
    a:
203.            Automatic Choice of Dimensionality for PCA. NIPS 2000: 598-604`
204.            Implements the probabilistic PCA model from:
205.            M. Tipping and C. Bishop, Probabilistic Principal Component Analysis,
206.            Journal of the Royal Statistical Society, Series B, 61, Part 3, pp. 611-
    622
207.            via the score and score_samples methods.
```

```
208.            See http://www.miketipping.com/papers/met-mppca.pdf
209.            For svd_solver == 'arpack', refer to `scipy.sparse.linalg.svds`.
210.            For svd_solver == 'randomized', see:
211.            `Finding structure with randomness: Stochastic algorithms
212.            for constructing approximate matrix decompositions Halko, et al., 2009
213.            (arXiv:909)`
214.            `A randomized algorithm for the decomposition of matrices
215.            Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert`
216.            Examples
217.            --------
218.            >>> import numpy as np
219.            >>> from sklearn.decomposition import PCA
220.            >>> X = np.array([[-1, -1], [-2, -1], [-3, -
    2], [1, 1], [2, 1], [3, 2]])
221.            >>> pca = PCA(n_components=2)
222.            >>> pca.fit(X)
223.            PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,

224.              svd_solver='auto', tol=0.0, whiten=False)
225.            >>> print(pca.explained_variance_ratio_)  # doctest: +ELLIPSIS
226.            [ 0.99244...  0.00755...]
227.            >>> print(pca.singular_values_)  # doctest: +ELLIPSIS
228.            [ 6.30061...  0.54980...]
229.            >>> pca = PCA(n_components=2, svd_solver='full')
230.            >>> pca.fit(X)                 # doctest: +ELLIPSIS +NORMALIZE_WHITESPAC
    E
231.            PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,

232.              svd_solver='full', tol=0.0, whiten=False)
233.            >>> print(pca.explained_variance_ratio_)  # doctest: +ELLIPSIS
234.            [ 0.99244...  0.00755...]
235.            >>> print(pca.singular_values_)  # doctest: +ELLIPSIS
236.            [ 6.30061...  0.54980...]
237.            >>> pca = PCA(n_components=1, svd_solver='arpack')
238.            >>> pca.fit(X)
239.            PCA(copy=True, iterated_power='auto', n_components=1, random_state=None,

240.              svd_solver='arpack', tol=0.0, whiten=False)
241.            >>> print(pca.explained_variance_ratio_)  # doctest: +ELLIPSIS
242.            [ 0.99244...]
243.            >>> print(pca.singular_values_)  # doctest: +ELLIPSIS
244.            [ 6.30061...]
245.            Notes
246.            -----
247.            PCA uses the maximum likelihood estimate of the eigenvalues, which does
    not
248.            include the Bessel correction, though in practice this should rarely mak
    e a
249.            difference in a machine learning context.
250.            See also
251.            --------
252.            KernelPCA
253.            SparsePCA
254.            TruncatedSVD
255.            IncrementalPCA
256.            """
257.
258.            def __init__(self, n_components=None, copy=True, whiten=False,
259.                         svd_solver='auto', tol=0.0, iterated_power='auto',
260.                         random_state=None):
261.                self.n_components = n_components
262.                self.copy = copy
263.                self.whiten = whiten
264.                self.svd_solver = svd_solver
265.                self.tol = tol
266.                self.iterated_power = iterated_power
```

```
267.                self.random_state = random_state
268.
269.            def fit(self, X, y=None):
270.                """Fit the model with X.
271.                Parameters
272.                ----------
273.                X : array-like, shape (n_samples, n_features)
274.                    Training data, where n_samples in the number of samples
275.                    and n_features is the number of features.
276.                Returns
277.                -------
278.                self : object
279.                    Returns the instance itself.
280.                """
281.                self._fit(X)
282.                return self
283.
284.            def fit_transform(self, X, y=None):
285.                """Fit the model with X and apply the dimensionality reduction on X.

286.                Parameters
287.                ----------
288.                X : array-like, shape (n_samples, n_features)
289.                    Training data, where n_samples is the number of samples
290.                    and n_features is the number of features.
291.                Returns
292.                -------
293.                X_new : array-like, shape (n_samples, n_components)
294.                """
295.                U, S, V = self._fit(X)
296.                U = U[:, :self.n_components_]
297.
298.                if self.whiten:
299.                    # X_new = X * V / S * sqrt(n_samples) = U * sqrt(n_samples)
300.                    U *= sqrt(X.shape[0])
301.                else:
302.                    # X_new = X * V = U * S * V^T * V = U * S
303.                    U *= S[:self.n_components_]
304.
305.                return U
306.
307.            def _fit(self, X):
308.                """Dispatch to the right submethod depending on the chosen solver.""
    "

309.

310.                # Raise an error for sparse input.
311.                # This is more informative than the generic one raised by check_arra
    y.
312.                if issparse(X):
313.                    raise TypeError('PCA does not support sparse input. See '
314.                                    'TruncatedSVD for a possible alternative.')
315.
316.                X = check_array(X, dtype=[np.float64], ensure_2d=True,
317.                                copy=self.copy)
318.
319.                # Handle n_components==None
320.                if self.n_components is None:
321.                    n_components = X.shape[1]
322.                else:
323.                    n_components = self.n_components
324.
325.                # Handle svd_solver
326.                svd_solver = self.svd_solver
327.                if svd_solver == 'auto':
328.                    # Small problem, just call full PCA
329.                    if max(X.shape) <= 500:
```

```
330.                 svd_solver = 'full'
331.             elif n_components >= 1 and n_components < .8 * min(X.shape):
332.                 svd_solver = 'randomized'
333.             # This is also the case of n_components in (0,1)
334.             else:
335.                 svd_solver = 'full'
336.
337.         # Call different fits for either full or truncated SVD
338.         if svd_solver == 'full':
339.             return self._fit_full(X, n_components)
340.         elif svd_solver in ['arpack', 'randomized']:
341.             return self._fit_truncated(X, n_components, svd_solver)
342.         else:
343.             raise ValueError("Unrecognized svd_solver='{0}'"
344.                              "".format(svd_solver))
345.
346.     def _fit_full(self, X, n_components):
347.         """Fit the model by computing full SVD on X"""
348.         n_samples, n_features = X.shape
349.
350.         if n_components == 'mle':
351.             if n_samples < n_features:
352.                 raise ValueError("n_components='mle' is only supported "
353.                                  "if n_samples >= n_features")
354.         elif not 0 <= n_components <= n_features:
355.             raise ValueError("n_components=%r must be between 0 and "
356.                              "n_features=%r with svd_solver='full'"
357.                              % (n_components, n_features))
358.
359.         # Center data
360.         self.mean_ = np.mean(X, axis=0)
361.         X -= self.mean_
362.
363.         U, S, V = linalg.svd(X, full_matrices=False)
364.         # flip eigenvectors' sign to enforce deterministic output
365.         U, V = svd_flip(U, V)
366.
367.         components_ = V
368.
369.         # Get variance explained by singular values
370.         explained_variance_ = (S ** 2) / n_samples
371.         total_var = explained_variance_.sum()
372.         explained_variance_ratio_ = explained_variance_ / total_var
373.         singular_values_ = S.copy()  # Store the singular values.
374.
375.         # Postprocess the number of components required
376.         if n_components == 'mle':
377.             n_components = \
378.                 _infer_dimension_(explained_variance_, n_samples, n_features
    )
379.         elif 0 < n_components < 1.0:
380.             # number of components for which the cumulated explained
381.             # variance percentage is superior to the desired threshold
382.             ratio_cumsum = stable_cumsum(explained_variance_ratio_)
383.             n_components = np.searchsorted(ratio_cumsum, n_components) + 1
384.
385.         # Compute noise covariance using Probabilistic PCA model
386.         # The sigma2 maximum likelihood (cf. eq. 12.46)
387.         if n_components < min(n_features, n_samples):
388.             self.noise_variance_ = explained_variance_[n_components:].mean()

389.         else:
390.             self.noise_variance_ = 0.
391.
392.         self.n_samples_, self.n_features_ = n_samples, n_features
393.         self.components_ = components_[:n_components]
```

```
394.                self.n_components_ = n_components
395.                self.explained_variance_ = explained_variance_[:n_components]
396.                self.explained_variance_ratio_ = \
397.                    explained_variance_ratio_[:n_components]
398.                self.singular_values_ = singular_values_[:n_components]
399.
400.                return U, S, V
401.
402.            def _fit_truncated(self, X, n_components, svd_solver):
403.                """Fit the model by computing truncated SVD (by ARPACK or randomized
    )
404.                on X
405.                """
406.                n_samples, n_features = X.shape
407.
408.                if isinstance(n_components, six.string_types):
409.                    raise ValueError("n_components=%r cannot be a string "
410.                                     "with svd_solver='%s'"
411.                                     % (n_components, svd_solver))
412.                elif not 1 <= n_components <= n_features:
413.                    raise ValueError("n_components=%r must be between 1 and "
414.                                     "n_features=%r with svd_solver='%s'"
415.                                     % (n_components, n_features, svd_solver))
416.                elif svd_solver == 'arpack' and n_components == n_features:
417.                    raise ValueError("n_components=%r must be stricly less than "
418.                                     "n_features=%r with svd_solver='%s'"
419.                                     % (n_components, n_features, svd_solver))
420.
421.                random_state = check_random_state(self.random_state)
422.
423.                # Center data
424.                self.mean_ = np.mean(X, axis=0)
425.                X -= self.mean_
426.
427.                if svd_solver == 'arpack':
428.                    # random init solution, as ARPACK does it internally
429.                    v0 = random_state.uniform(-1, 1, size=min(X.shape))
430.                    U, S, V = svds(X, k=n_components, tol=self.tol, v0=v0)
431.                    # svds doesn't abide by scipy.linalg.svd/randomized_svd
432.                    # conventions, so reverse its outputs.
433.                    S = S[::-1]
434.                    # flip eigenvectors' sign to enforce deterministic output
435.                    U, V = svd_flip(U[:, ::-1], V[::-1])
436.
437.                elif svd_solver == 'randomized':
438.                    # sign flipping is done inside
439.                    U, S, V = randomized_svd(X, n_components=n_components,
440.                                             n_iter=self.iterated_power,
441.                                             flip_sign=True,
442.                                             random_state=random_state)
443.
444.                self.n_samples_, self.n_features_ = n_samples, n_features
445.                self.components_ = V
446.                self.n_components_ = n_components
447.
448.                # Get variance explained by singular values
449.                self.explained_variance_ = (S ** 2) / n_samples
450.                total_var = np.var(X, axis=0)
451.                self.explained_variance_ratio_ = \
452.                    self.explained_variance_ / total_var.sum()
453.                self.singular_values_ = S.copy()  # Store the singular values.
454.                if self.n_components_ < n_features:
455.                    self.noise_variance_ = (total_var.sum() -
456.                                            self.explained_variance_.sum())
457.                else:
458.                    self.noise_variance_ = 0.
```

```
459.
460.                return U, S, V
461.
462.            def score_samples(self, X):
463.                """Return the log-likelihood of each sample.
464.                See. "Pattern Recognition and Machine Learning"
465.                by C. Bishop, 12.2.1 p. 574
466.                or http://www.miketipping.com/papers/met-mppca.pdf
467.                Parameters
468.                ----------
469.                X : array, shape(n_samples, n_features)
470.                    The data.
471.                Returns
472.                -------
473.                ll : array, shape (n_samples,)
474.                    Log-likelihood of each sample under the current model
475.                """
476.                check_is_fitted(self, 'mean_')
477.
478.                X = check_array(X)
479.                Xr = X - self.mean_
480.                n_features = X.shape[1]
481.                log_like = np.zeros(X.shape[0])
482.                precision = self.get_precision()
483.                log_like = -.5 * (Xr * (np.dot(Xr, precision))).sum(axis=1)
484.                log_like -= .5 * (n_features * log(2. * np.pi) -
485.                                  fast_logdet(precision))
486.                return log_like
487.
488.            def score(self, X, y=None):
489.                """Return the average log-likelihood of all samples.
490.                See. "Pattern Recognition and Machine Learning"
491.                by C. Bishop, 12.2.1 p. 574
492.                or http://www.miketipping.com/papers/met-mppca.pdf
493.                Parameters
494.                ----------
495.                X : array, shape(n_samples, n_features)
496.                    The data.
497.                Returns
498.                -------
499.                ll : float
500.                    Average log-likelihood of the samples under the current model
501.                """
502.                return np.mean(self.score_samples(X))
503.
504.
505.        @deprecated("RandomizedPCA was deprecated in 0.18 and will be removed in "
506.                    "0.20. "
507.                    "Use PCA(svd_solver='randomized') instead. The new implementatio
    n "
508.                    "DOES NOT store whiten ``components_``. Apply transform to get "

509.                    "them.")
510.        class RandomizedPCA(BaseEstimator, TransformerMixin):
511.            """Principal component analysis (PCA) using randomized SVD
512.            .. deprecated:: 0.18
513.                This class will be removed in 0.20.
514.                Use :class:`PCA` with parameter svd_solver 'randomized' instead.
515.                The new implementation DOES NOT store whiten ``components_``.
516.                Apply transform to get them.
517.            Linear dimensionality reduction using approximated Singular Value
518.            Decomposition of the data and keeping only the most significant
519.            singular vectors to project the data to a lower dimensional space.
520.            Read more in the :ref:`User Guide <RandomizedPCA>`.
521.            Parameters
522.            ----------
```

```
523.            n_components : int, optional
524.                Maximum number of components to keep. When not given or None, this
525.                is set to n_features (the second dimension of the training data).
526.            copy : bool
527.                If False, data passed to fit are overwritten and running
528.                fit(X).transform(X) will not yield the expected results,
529.                use fit_transform(X) instead.
530.            iterated_power : int, default=2
531.                Number of iterations for the power method.
532.                .. versionchanged:: 0.18
533.            whiten : bool, optional
534.                When True (False by default) the `components_` vectors are multiplie
    d
535.                by the square root of (n_samples) and divided by the singular values
     to
536.                ensure uncorrelated outputs with unit component-wise variances.
537.                Whitening will remove some information from the transformed signal
538.                (the relative variance scales of the components) but can sometime
539.                improve the predictive accuracy of the downstream estimators by
540.                making their data respect some hard-wired assumptions.
541.            random_state : int, RandomState instance or None, optional, default=None

542.                If int, random_state is the seed used by the random number generator
    ;
543.                If RandomState instance, random_state is the random number generator
    ;
544.                If None, the random number generator is the RandomState instance use
    d
545.                by `np.random`.
546.            Attributes
547.            ----------
548.            components_ : array, shape (n_components, n_features)
549.                Components with maximum variance.
550.            explained_variance_ratio_ : array, shape (n_components,)
551.                Percentage of variance explained by each of the selected components.

552.                If k is not set then all components are stored and the sum of explai
    ned
553.                variances is equal to 1.0.
554.            singular_values_ : array, shape (n_components,)
555.                The singular values corresponding to each of the selected components
    .
556.                The singular values are equal to the 2-
    norms of the ``n_components``
557.                variables in the lower-dimensional space.
558.            mean_ : array, shape (n_features,)
559.                Per-feature empirical mean, estimated from the training set.
560.            Examples
561.            --------
562.            >>> import numpy as np
563.            >>> from sklearn.decomposition import RandomizedPCA
564.            >>> X = np.array([[-1, -1], [-2, -1], [-3, -
    2], [1, 1], [2, 1], [3, 2]])
565.            >>> pca = RandomizedPCA(n_components=2)
566.            >>> pca.fit(X)                 # doctest: +ELLIPSIS +NORMALIZE_WHITESPAC
    E
567.            RandomizedPCA(copy=True, iterated_power=2, n_components=2,
568.                   random_state=None, whiten=False)
569.            >>> print(pca.explained_variance_ratio_)  # doctest: +ELLIPSIS
570.            [ 0.99244...  0.00755...]
571.            >>> print(pca.singular_values_)  # doctest: +ELLIPSIS
572.            [ 6.30061...  0.54980...]
573.            See also
574.            --------
575.            PCA
576.            TruncatedSVD
```

```
577.            References
578.            ----------
579.            .. [Halko2009] `Finding structure with randomness: Stochastic algorithms

580.              for constructing approximate matrix decompositions Halko, et al., 2009

581.              (arXiv:909)`
582.            .. [MRT] `A randomized algorithm for the decomposition of matrices
583.              Per-Gunnar Martinsson, Vladimir Rokhlin and Mark Tygert`
584.            """
585.
586.        def __init__(self, n_components=None, copy=True, iterated_power=2,
587.                     whiten=False, random_state=None):
588.            self.n_components = n_components
589.            self.copy = copy
590.            self.iterated_power = iterated_power
591.            self.whiten = whiten
592.            self.random_state = random_state
593.
594.        def fit(self, X, y=None):
595.            """Fit the model with X by extracting the first principal components
              .
596.            Parameters
597.            ----------
598.            X : array-like, shape (n_samples, n_features)
599.                Training data, where n_samples in the number of samples
600.                and n_features is the number of features.
601.            Returns
602.            -------
603.            self : object
604.                Returns the instance itself.
605.            """
606.            self._fit(check_array(X))
607.            return self
608.
609.        def _fit(self, X):
610.            """Fit the model to the data X.
611.            Parameters
612.            ----------
613.            X : array-like, shape (n_samples, n_features)
614.                Training vector, where n_samples in the number of samples and
615.                n_features is the number of features.
616.            Returns
617.            -------
618.            X : ndarray, shape (n_samples, n_features)
619.                The input data, copied, centered and whitened when requested.
620.            """
621.            random_state = check_random_state(self.random_state)
622.            X = np.atleast_2d(as_float_array(X, copy=self.copy))
623.
624.            n_samples = X.shape[0]
625.
626.            # Center data
627.            self.mean_ = np.mean(X, axis=0)
628.            X -= self.mean_
629.            if self.n_components is None:
630.                n_components = X.shape[1]
631.            else:
632.                n_components = self.n_components
633.
634.            U, S, V = randomized_svd(X, n_components,
635.                                     n_iter=self.iterated_power,
636.                                     random_state=random_state)
637.
638.            self.explained_variance_ = exp_var = (S ** 2) / n_samples
639.            full_var = np.var(X, axis=0).sum()
```

```
640.                self.explained_variance_ratio_ = exp_var / full_var
641.                self.singular_values_ = S  # Store the singular values.
642.
643.                if self.whiten:
644.                    self.components_ = V / S[:, np.newaxis] * sqrt(n_samples)
645.                else:
646.                    self.components_ = V
647.
648.                return X
649.
650.            def transform(self, X, y=None):
651.                """Apply dimensionality reduction on X.
652.                X is projected on the first principal components previous extracted
653.                from a training set.
654.                Parameters
655.                ----------
656.                X : array-like, shape (n_samples, n_features)
657.                    New data, where n_samples in the number of samples
658.                    and n_features is the number of features.
659.                Returns
660.                -------
661.                X_new : array-like, shape (n_samples, n_components)
662.                """
663.                check_is_fitted(self, 'mean_')
664.
665.                X = check_array(X)
666.                if self.mean_ is not None:
667.                    X = X - self.mean_
668.
669.                X = fast_dot(X, self.components_.T)
670.                return X
671.
672.            def fit_transform(self, X, y=None):
673.                """Fit the model with X and apply the dimensionality reduction on X.

674.                Parameters
675.                ----------
676.                X : array-like, shape (n_samples, n_features)
677.                    New data, where n_samples in the number of samples
678.                    and n_features is the number of features.
679.                Returns
680.                -------
681.                X_new : array-like, shape (n_samples, n_components)
682.                """
683.                X = check_array(X)
684.                X = self._fit(X)
685.                return fast_dot(X, self.components_.T)
686.
687.            def inverse_transform(self, X, y=None):
688.                """Transform data back to its original space.
689.                Returns an array X_original whose transform would be X.
690.                Parameters
691.                ----------
692.                X : array-like, shape (n_samples, n_components)
693.                    New data, where n_samples in the number of samples
694.                    and n_components is the number of components.
695.                Returns
696.                -------
697.                X_original array-like, shape (n_samples, n_features)
698.                Notes
699.                -----
700.                If whitening is enabled, inverse_transform does not compute the
701.                exact inverse operation of transform.
702.                """
703.                check_is_fitted(self, 'mean_')
704.
```

```
705.                X_original = fast_dot(X, self.components_)
706.                if self.mean_ is not None:
707.                    X_original = X_original + self.mean_
708.                return X_original
```

References

[1] Kriegel, H., Schubert, E., & Zimek, A. (2016). The (black) art of runtime evaluation: Are we comparing algorithms or implementations? *Knowledge and Information Systems.* doi:10.1007/s10115-016-1004-2

[2] MacQueen, J. (1967, June). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability* (Vol. 1, No. 14, pp. 281-297).

[3] MacKay, D. J. (2003). *Information theory, inference and learning algorithms.* Cambridge university press.

[4] Thorndike, R. L. (1953). Who belongs in the family?. *Psychometrika, 18*(4), 267-276.

[5] Michael, R. (2003). Children's cognitive skill development in Britain and the United States. *International Journal of Behavioral Development, 27*(5), 396-408.

[6] (n.d.). Retrieved May 20, 2017, from https://onlinecourses.science.psu.edu/stat505/book/export/html/49

[7] S. (n.d.). Scipy/scipy. Retrieved May 20, 2017, from https://github.com/scipy/scipy/blob/master/scipy/cluster/vq.py

[8] S. (n.d.). Scikit-learn/scikit-learn. Retrieved May 20, 2017, from https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/decomposition/pca.py