PRINCIPLE COMPONENT ANALYSIS

Contents

Introduction 2

1. Mathematical interpretation of PCA 2

1.1 Maximizing projection variance 2

1.2 Singular value decomposition and minimizing total perpendicular square distance 6

1.3 Singular value decomposition of a matrix 9

1.4 Relationship between SVD and PCA 12

1.5 Fast computation of principle components 13

2. Application of principle component analysis algorithm 16

2.1 PCA algorithm 16

2.2 Select the number of principle components 17

2.3 Apply PCA on machine learning examples 19

2.3.1 Visualize PCA on Iris dataset 19

2.3.2 Apply PCA in faces recognition 24

Conclusion 31

References 31

Introduction

Curse of dimensionality is defined as the variety of problems that would occur when analysing data in high-dimensional spaces which are unlikely to happen in low-dimensional space. For example, when the dimension of the data increases, the efficiency of most machine learning algorithms would decrease. To counter this difficulty, dimensional reduction or data reduction is always considered. This essay would introduce an important technique for dimensional reduction which is called the principle component analysis (PCA) from two aspects, its mathematical interpretation and its basic machine learning application.

1. Mathematical interpretation of PCA

Principle component analysis or PCA is defined as a linear transformation that projects the original d-dimensional data points to a new k-dimensional coordinate system. And the k orthogonal vectors that span the k-dimensional subspace are called principle components (Shalizi C. , 2012).

There are at least two ways of defining principle components, one is maximizing the total projection variance while the other is minimizing the total perpendicular square distance of the data points to the subspace. For the remaining of the section we are going to show how both interpretations work and proof they are actually equivalent.

1.1 Maximizing projection variance

The simplest definition of principle components is a subspace that would maximize the projection variance. The first principle component is the direction of the unit vector that would result in the largest projection variance, the second principle component is the direction of the unit vector that would maximize the projection variance of all directions

perpendicular to the first component. The ith principle component is then the direction of the unit vector that would maximize the projection variance which is perpendicular to the first i-1 principle component. We would now show how to calculate the first principle component under this definition.

Consider the following projection problem. Suppose that there is a d-dimensional space, we want to project all n **centred**[1] data points of the d-dimensional space on a d*1 vector, such that the variance of the resulting data on the 1-dimensional space is as large as possible.

Before solving this problem, let's introduce some notations. Let **S** be the covariance matrix of the sample data, **M** be a centralized n*d matrix such that each row represents a data point, **u** be the d * 1 vector of unit length that data points projects. Hence, the above problem could be formally written as

$Suppose\ M\ is\ a\ n \times d\ (assume\ n > d) matrix, and\ \boldsymbol{u}\ \in R^d, find\ unit\ vector\ \boldsymbol{u}\ such$

$that\ Var(M\boldsymbol{u}) is\ maximized.$

Lemma 1: For any **centred** n*d matrix M, $M^T M = (n-1)\boldsymbol{S}$, which S is the covariance matrix of M.

Proof: Let **I** represents the ith-column of the matrix M and **J** represents the jth-column of the matrix M. Then, the (i, j) entry of $(M^T M)_{i.j} = I^T J$.

Consider the (i, j) entry of the covariance matrix.

[1] Centred refers to rescale the data such that each feature has zero mean.

$$S_{i,j} = Cov(I,J) = \frac{1}{n-1}\big((I - E[I]) \cdot (J - E[J])\big) = \frac{1}{n-1} I^T J (the\ data\ has\ been\ centralized)$$

$$= \frac{1}{n-1}(M^T M)_{i.j}$$

Hence, we conclude that $M^T M = (n-1)\boldsymbol{S}$ (1.1.1).

Let's return to the original problem. Let Mi be the ith row of M.

Then by definition of variance we have $Var(M\boldsymbol{u}) = \frac{1}{n}\sum_i (M_i u)^2 = \frac{1}{n}(M_i u)^T M_i u = \frac{\mathbf{1}}{\boldsymbol{n}}\boldsymbol{u}^T M^T M\boldsymbol{u}$.

By lemma 1, we can get $Var(M\boldsymbol{u}) = \boldsymbol{u}^T\boldsymbol{S}\boldsymbol{u}$.

Hence, the problem could be transformed to an optimization problem. Find max ($\boldsymbol{u}^T\boldsymbol{S}\boldsymbol{u}$), such that ||**u**|| = 1. We apply the Lagrange multiplier (Shalizi C. , 2012).

$$L(\boldsymbol{u},\lambda) = \boldsymbol{u}^T\boldsymbol{S}\boldsymbol{u} - \boldsymbol{\lambda}\big(||\boldsymbol{u}|| - 1\big) = \boldsymbol{u}^T\boldsymbol{S}\boldsymbol{u} - \boldsymbol{\lambda}(\boldsymbol{u}^T\boldsymbol{u} - 1)$$

The optimal condition is reached when $\frac{\partial L}{\partial u} = 0$.

Hence, $\frac{\partial L}{\partial u} = 2Su - 2\lambda u = 0, which\ is\ Su = \lambda u$ (1.1.2).

Therefore, we can conclude that the vector u should be the eigenvector of the covariance matrix of M.

Note that $Var(M\boldsymbol{u}) = \boldsymbol{u}^T\boldsymbol{S}\boldsymbol{u} = \boldsymbol{u}^T\lambda u = \lambda$, which means, the larger the eigenvalue, the larger the projection variance can be. Thus, we define the eigenvector that corresponds to the largest eigenvector of the covariance matrix as the **first principle component**. And we can extend the definition to the kth principle component: the **kth principle component** is the eigenvector corresponds to the kth largest eigenvalue of S (Shalizi C. R., 2019).

Note that we have previously stated that principle components must be mutually orthogonal, this can be proved by showing that the distinct eigenvectors of the covariance matrix are mutually orthogonal. Let's proof a lemma below.

Lemma 2: different eigenvectors of a symmetric matrix are mutually orthogonal.

Proof. Get $\boldsymbol{x1}, \boldsymbol{x2}, such\ that\ \boldsymbol{x1}, \boldsymbol{x2}\ are\ 2\ distinct\ eigenvectors\ of\ the\ matrix\ M.$

By definition of eigenvectors, $\exists\ \lambda 1 \neq 0, \lambda 2\ \neq 0\ such\ that\ Mx1 = \lambda 1 x1\ and\ Mx2 = \lambda 2 x2$. Since M is symmetric, $M = M^T$, we could get

$$x1^T Mx2 = \quad x1^T M^T x2 \qquad (M\ is\ symmetric)$$

$$= (Mx1)^T x \qquad (definition\ of\ transpose)$$

$$= (\lambda 1 x1)^T x2 \qquad (definition\ of\ eigenvector)$$

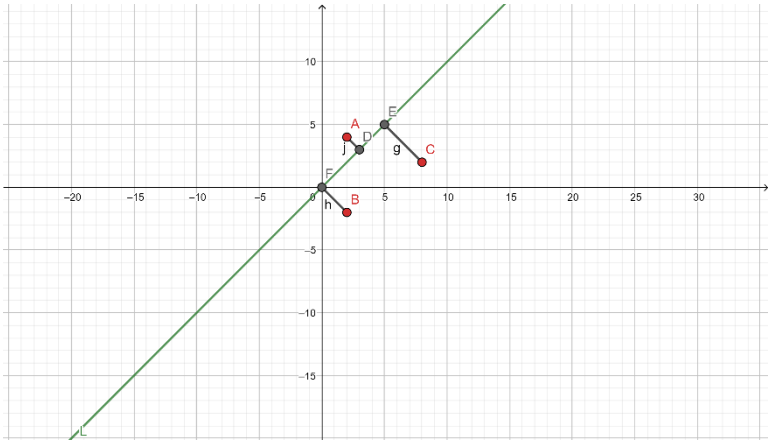
$$= \lambda 1 x1^T x2\ .$$

Note that $x1^T Mx2 = x1^T\ \lambda 2 x2 = \lambda 2 x1^T x2$. We get $\lambda 1 x1^T x2 = \lambda 2 x1^T x2$, which is, $(\lambda 1 - \lambda 2) x1^T x2 = 0$. This means that as long as $\lambda 1 - \lambda 2$ is not equal to 0, $x1^T x2 = 0$. Therefore, different eigenvectors of a symmetric matrix are mutually orthogonal (Taboga).

Since the covariance matrix is symmetric and the principle components are distinct vectors, we can infer that these vectors are mutually orthogonal by the above lemma.

1.2 Singular value decomposition and minimizing total perpendicular square distance

Principle components can also be defined as the linear subspace that minimize the total square projection distance of the data points. Let's use the image on the left as an example, if



we project 2-dimensional data points A, B, C to a line L, then L is the principle component if and only if $AD^2 + BF^2 + CE^2$ is minimized (Williams, 2016).

Note that this problem setting is exactly the same as the problem setting of another important concept **singular value decomposition** (SVD).

Singular value decomposition is defined as the procedure of finding the best-fit subspace. Suppose that there are n d-dimensional data points, represented by a n*d matrix. Then singular value decomposition would derive the p-dimensional (in most cases p << d) subspace that best-fit the original data. Here best-fit is defined as the minimum sum of squares of perpendicular distance from the data points to the projection subspace.

Let's firstly consider the case of projection to a 1-dimensional vector. Suppose that there is a n * d data matrix M (each row represents the transpose of a d-dimensional data vector), we want to find a unit d dimensional vector **v** such that $\sum_{i=1}^{n}(dist(xi, v))^2$ is minimized (Avrim Blum, 2018).

By Pythagoras theorem, we could get $dist(\boldsymbol{x_i}, \boldsymbol{v})^2 = \left||\boldsymbol{x_i}|\right|^2 - \left||proj(\boldsymbol{xi}, \boldsymbol{v})|\right|^2$.

Hence, $\sum_{i=1}^{n}\left(dist(xi, v)\right)^2 = \sum(\left||\boldsymbol{x_i}|\right|^2 - \left||proj(\boldsymbol{xi}, \boldsymbol{v})|\right|^2) = constant - ||\boldsymbol{Mv}||^2$. Thus, minimizing $\sum_{i=1}^{n}(dist(xi, v))^2$is equivalent to maximizing $||\boldsymbol{Mv}||$. By definition, the unit vector **v1** that maximize $||\boldsymbol{Mv}||$ is called the **first singular vector** of matrix M (Avrim Blum, 2018).

$$v1 = \underset{||v||=1}{\operatorname{argmax}} ||Mv||$$

Similarly, we define the **second singular vector** as the unit vector that is perpendicular to the first singular vector that maximize $||\boldsymbol{Mv}||$, and the ith singular vector as the unit vector that is perpendicular to all of the i-1 singular vectors that maximize $||\boldsymbol{Mv}||$ is called the ith singular vector (Avrim Blum, 2018).

$$\boldsymbol{v_i} = \underset{||v||=1, v \cdot v_p = 0 (1 \le p < i)}{\operatorname{argmax}} ||Mv|| \qquad (1.2.1)$$

Like eigenvectors and eigenvalues, for each singular vector there's a singular value corresponds to it, let's define the **ith singular value** as

$$\sigma_i = ||\boldsymbol{Mv_i}|| \qquad (1.2.2)$$

So now we have a greedy algorithm, we iterate for p times, each time we find a unit vector that maximize $||Mv||$ and is perpendicular to all unit vectors found in the previous p-1 iterations. After the iteration terminates or at a point the singular value is equal to 0, the subspace spanned by the p vectors is the best-fit p-dimensional space of the original data.

However, our original objective is to minimize the total perpendicular square distance from the data points to the p-dimensional subspace. We need to show that greedily select the first p singular vectors could satisfy our objective.

Proof by induction. Let P(K) is the statement that greedily select the first K singular vectors would form a best-fit K dimensional subspace.

Note that P(1) holds, by the definition of the first singular vector. Suppose that P(K-1) holds, we have $\mathbf{v_1, v_2, \dots v_{k-1}}$ as the unit vector that form the basis of the K-1 dimensional subspace. Consider a best-fit K dimensional subspace W with orthogonal basis $\mathbf{w_1, w_2, \dots w_k}$ and let's assume the kth singular vector that is perpendicular to the best-fit K-1 dimensional subspace is $\mathbf{v_k}$. Let's choose $\mathbf{w_k}$ as the vector that is perpendicular to $\mathbf{v_1, v_2, \dots v_{k-1}}$. We then have,

$$\left|\left|\boldsymbol{M v_k}\right|\right| \geq \left|\left|\boldsymbol{M w_k}\right|\right| (definition\ of\ kth\ singular\ vector)$$

Therefore, $\sum_{i=1}^{k} ||\boldsymbol{M v_i}||^2 \geq \sum_{i=1}^{k} ||\boldsymbol{M w_i}||^2$, which means that select $\mathbf{v_k}$ would not make the answer worse. Hence, we could infer that the subspace formed by the first k singular vectors is a best-fit subspace, which means that P(K) holds. By induction P(i) holds for all I >= 1. Thus, the greedy algorithm works (Avrim Blum, 2018).

We can furtherly show that the summation of the square of the length of each data points is equal to the sum of square of projections of M on each singular vector (Avrim Blum, 2018).

$$\sum_{i=1}^{n} \left|\left|xi\right|\right|^2 = \sum_{j=1}^{p} \left|\left|Mv_j\right|\right|^2.$$

Where, $\mathbf{x_i}$ corresponds to the ith data point and $\mathbf{v_j}$ corresponds to the jth singular vector of M and p refers to the total singular vectors of M.

Proof. Let's consider the ith data point $\mathbf{x_i}$, it is not difficult to show that

$$\boldsymbol{x_i} = \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} (\boldsymbol{x_i} \cdot \boldsymbol{v_j}) \boldsymbol{v_j}$$

since the singular vectors form an orthogonal basis of M.

Then, we have

$$||\boldsymbol{x_i}||^2 = \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} (\boldsymbol{x_i} \cdot \boldsymbol{v_j})^2$$

Therefore, $\sum_{i=1}^{n} ||x_i||^2 = \sum_{i=1}^{n} \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} (\boldsymbol{x_i} \cdot \boldsymbol{v_j})^2 = \sum_{j=1}^{p} \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} (\boldsymbol{x_i} \cdot \boldsymbol{v_j})^2 = \sum_{\boldsymbol{j=1}}^{\boldsymbol{p}} ||\boldsymbol{M v_j}||^2$ (1.2.3).

1.3 Singular value decomposition of a matrix

In the above section, we proved that the best-fit K dimensional subspace is the subspace spanned by the first K singular vectors of the original n * d data matrix. This section would explain how to do SVD on a matrix.

We can make the following claim, all the n*d (n and d are positive numbers) matrices can be decomposed in the following way,

$$M = USV^T$$

where S is a diagonal r*r matrix with the ith value equals to the ith singular value σi of M, V^T is a r*d matrix which the ith row equals to the transpose of the ith singular vector $\mathbf{v_i}$ of M, and U is an n*r matrix which the ith column is equal to $\frac{\boldsymbol{M v_i}}{\sigma i}$.

$$\begin{pmatrix} M_{11} & \cdots & M_{1d} \\ \vdots & \ddots & \vdots \\ M_{n1} & \cdots & M_{nd} \end{pmatrix} = \begin{pmatrix} \frac{\boldsymbol{M v_1}}{\sigma 1} \ldots & \cdots & \frac{\boldsymbol{M v_r}}{\sigma r} \end{pmatrix} \begin{pmatrix} \sigma_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sigma_r \end{pmatrix} \begin{pmatrix} v_{11} & \cdots & v_{1d} \\ \vdots & \ddots & \vdots \\ v_{r1} & \cdots & v_{rd} \end{pmatrix}$$

$$[n * d] = \qquad [n * r] \qquad [r * r] \qquad [r * d]$$

Proof. By the definition of singular vector, **all** d-dimensional vector **u** can be written as a linear combination of singular vectors and a vector perpendicular to the singular vectors.

$u = w + \sum_{i=1}^{r} c_i \boldsymbol{v_i}$, where ci are constants and $\boldsymbol{v_i}$ are the singular values of M.

It is easy to show that Mw = 0, because if Mw ≠ 0, w would become a singular vector of M. Then by the definition of orthogonal basis we could derive that for all $\boldsymbol{u}$, $M\boldsymbol{u} = \sum_{i=1}^{r} c_i M\boldsymbol{v_i} = \sum_{i=1}^{r} M\boldsymbol{v_i}(\boldsymbol{v_i} \cdot \boldsymbol{u}) = \sum_{i=1}^{r} Mv_i v_i^T \boldsymbol{u} = USV^T\boldsymbol{u}$. Therefore, by using the property that two matrices are identical if and only if for all vector **v**, Av = Bv (Avrim Blum, 2018), we get $M = USV^T$.

After showing that all matrices can be decomposed into the product of U, S, V^T, I'm going to explain how to calculate U, S and V^T for a matrix[2].

Lemma 3: $U^TU = I$.

Proof (Avrim Blum, 2018). It is quite trivial that $U^TU = I$ is equivalent to $\frac{M\boldsymbol{v_i}}{\sigma i}$ are pairwise orthogonal. Let's proof by contradiction. Suppose that $\frac{M\boldsymbol{v_i}}{\sigma i}$ is not perpendicular to $\frac{M\boldsymbol{v_j}}{\sigma j}$, which i is the smallest index i that violates the orthogonal property. We assume that $\frac{M\boldsymbol{v_i}}{\sigma i} \cdot \frac{M\boldsymbol{v_j}}{\sigma j} = \epsilon > 0$. Then, let's consider a vector $\mathbf{v_i}$' such that $\boldsymbol{v'_i} = \frac{vi+\delta vj}{||vi+\delta vj||}$. Note that $Mv'_i = M\frac{vi+\delta vj}{||vi+\delta vj||} = \frac{\sigma i\frac{M\boldsymbol{v_i}}{\sigma i}+\sigma j\delta\frac{M\boldsymbol{v_j}}{\sigma j}}{||vi+\delta vj||} = \frac{\sigma i\frac{M\boldsymbol{v_i}}{\sigma i}+\sigma j\delta\frac{M\boldsymbol{v_j}}{\sigma j}}{\sqrt{1+\delta^2}}$.

By using the fact that $(1+x)^{-1/2} \geq 1 - \frac{1}{2}x$ for all x, and if δ is small we have

[2] Note that, even if we know that S, V^T are matrices of the singular values and singular vectors, we haven't introduced any method of calculating singular values of a matrix at this point.

$$|Mv'_i| > \frac{Mv_i}{\sigma i} \cdot \frac{\sigma i \frac{Mv_i}{\sigma i} + \sigma j \delta \frac{Mv_j}{\sigma j}}{\sqrt{1+\delta^2}} > \frac{Mv_i}{\sigma i} \cdot \left(\sigma i \frac{Mv_i}{\sigma i} + \sigma j \delta \frac{Mv_j}{\sigma j}\right)\left(1 - \frac{1}{2}\delta^2\right) > (\sigma i + \sigma j \delta \epsilon)\left(1 - \frac{1}{2}\delta^2\right) > \sigma i.$$

Hence, we conclude that we found a v'_i that is perpendicular to all previous singular vectors and also has a larger singular value than $\mathbf{v_i}$, this contradicts with the definition of ith singular vector. Therefore, we conclude that $U^TU = I$.

Lemma 4: $VV^T = I$.

Proof. Since the singular vectors are mutually orthogonal unit vectors, it is easy to show that

$$[VV^T]_{i,j} = \boldsymbol{v_i} \cdot \boldsymbol{v_j} = \begin{cases} \mathbf{1} & when\ i = j \\ \mathbf{0} & otherwise \end{cases}$$

We now have 3 equations.

$$M = USV^T$$

$$U^TU = I$$

$$VV^T = I$$

Consider $M^TM = (USV^T)^T(USV^T) = ((V^T)^TS^TU^T)USV^T = VS^TU^TUSV^T = VS^TSV^T$

Since S is a diagonal matrix, we have $S = S^T$. Hence, $M^TM = VS^2V^T$, which is equivalent to $M^TMVV^{-1} = VS^2V^TVV^{-1} = VS^2V^{-1}$ (Zichen, 2019).

Note that if we rewrite the last expression as $M^TM = VS^2V^{-1}$, we get the exact same expression as the matrix diagonalization of M^TM. Since M^TM is always a real symmetric matrix, it is always diagonalizable. Thus, we conclude that, the singular value decomposition of a matrix M is

$$U = \begin{pmatrix} | & \dots & | \\ \frac{Mu_1}{\sqrt{\lambda 1}} & \dots & \frac{Mu_r}{\sqrt{\lambda r}} \\ | & \dots & | \end{pmatrix}, S = \begin{pmatrix} \sqrt{\lambda 1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \sqrt{\lambda r} \end{pmatrix}, V^T = \begin{pmatrix} ---\boldsymbol{u_1}^T--- \\ ---\boldsymbol{u_2}^T--- \\ \vdots \\ ---\boldsymbol{u_r}^T--- \end{pmatrix} \quad (1.3)$$

where λi refers to the ith **distinct** eigenvalue of $M^T M$, and $\mathbf{u_i}$ as the eigenvector corresponds to λi. And to make the singular value decomposition of a matrix unique, we always order the eigenvalues in such way that $\lambda 1 > \lambda 2 > \cdots > \lambda r$ (Avrim Blum, 2018).

1.4 Relationship between SVD and PCA

After knowing how to do singular value decomposition on a matrix, let's return to principle component analysis. In this section, let's figure out the relationship between singular value decomposition and principle component analysis.

Assume that we have an n*d **centred** matrix M which each row refers to a data point. In section 1.1, we have worked out under the maximum projection variance problem setting, the principle components are the eigenvectors of the covariance matrix of M. Let's now do singular value decomposition on matrix M.

$$M = USV^T$$

Let's use C to represents the covariance matrix of M, then

$$C = \frac{M^T M}{n-1} (lemma\ 1.1)$$

$$= \frac{(USV^T)^T USV^T}{n-1} = \frac{VS^T U^T USV^T}{n-1}$$

$$= \frac{VS^2V^T}{n-1}\ (lemma\ 3) = V\sum V^T \quad \text{(Zichen, 2019)} \qquad (1.4.1)$$

Thus, we could make three interpretation of the above result. Firstly, columns of V are not only the singular vectors of the matrix M, but also the **eigenvectors** of the covariance matrix **C**. This means that the singular vectors of the **centred** data matrix are exactly the same as the principle components of the **centred** data (i.e. the ith singular vector is the ith principle component). Secondly, the ith eigenvalue of the covariance matrix

$$\lambda i = \frac{{s_i}^2}{n-1} \qquad (1.4.2)$$

which s_i is the ith singular value of matrix M. Finally, we know that by definition the subspace formed by the principle components is the subspace that maximize the projection variance, and now we know that this subspace is also the subspace spanned by the singular vectors of the original data, hence it also minimize the total perpendicular square projection distance of the centred data points.

Because of the strong connection between SVD and PCA, **Singular value decomposition of the covariance matrix** is the most commonly used technique of finding the principle components of the data.

1.5 Fast computation of principle components

In section 1.4, we discussed that principle components could be calculated by finding the eigenvalues and eigenvectors of the covariance matrix and the method is usually called *"the covariance method"*. However, finding eigenvectors and eigenvalues of a square matrix precisely is usually done by solving the characteristic polynomial of the given matrix. This method can only work on small matrices, since when the matrix is large, the characteristic equation with high exponents cannot be solved efficiently. In practice, principle component

analysis or singular value decomposition of the covariance matrix is usually done by the *"power iteration method"*.

Similar to the previous sections, we are going to introduce how to compute the first principle component and then expand this result to k principle components.

Let's denote the centred data matrix as **A** (dimension n * d), and the covariance matrix as **M** (size d * d). Then by section 1.1, we know that if we want to find the first principle component (or singular vector) of matrix **A**, we can simply find the eigenvector corresponds to the largest eigenvalue of matrix M.

The heuristic behind the power method is based on the following geometric interpretation of the singular vectors. Suppose that there is a n-dimensional unit ball, the matrix **M** maps the unit sphere to an ellipsoid, then the first singular vector $\mathbf{v_1}$ points to the direction that the ball gets maximally stretched, the second singular vector $\mathbf{v_2}$ points to the direction that is perpendicular to $\mathbf{v_1}$ that the ball gets maximally stretched. And the ith singular vector $\mathbf{v_i}$ points to the direction that is perpendicular to the first i-1 singular vectors that the ball gets maximally stretched. We might expect that if we start from a random d-dimensional vector **u** and keeps applying the mapping M (i.e. matrix multiplication **u = Mu**) over and over again, the resulting vector should have been stretched so much on the direction of the first singular vector that it almost lies completely on this direction (Roughgarden, 2015).

We define the **power iteration algorithm** as follows (Roughgarden, 2015).

Input: covariance matrix M, random d-dimensional unit vector **u**.

Output: The approximate first singular vector M.

1. Select a random d-dimensional unit vector $\mathbf{u_0}$.
2. Let $\boldsymbol{u_t = Mu_{t-1}}$ for t = 1…
3. Repeat step 2 until $\left|\left|\boldsymbol{u_t - u_{t-1}}\right|\right| < \boldsymbol{\varepsilon}$.

Return $\frac{\boldsymbol{u_t}}{||\boldsymbol{u_t}||}$.

Let's proof the performance of such algorithm in finding the first singular vector of the covariance matrix. Note that during the ith iteration, $\boldsymbol{u_i = M^i u_0}$.Consider $\boldsymbol{u_0}$ could be decomposed into linear combination of the eigenvectors ($\boldsymbol{v_1, v_2}$ ...) of the matrix M. Then by definition we have

$$\boldsymbol{u_0} = \sum_{i=1}^{n} c_i \boldsymbol{v}_i \tag{1.5.1}$$

Which n refers to the rank of matrix M. Since the original vector $\mathbf{u_0}$ is selected randomly then with almost a probability of 1, we have c_1 is not equal to 0. We multiply both side of 1.5.1 by M^i and using the fact that singular values of M are the same as the eigenvectors of M, we get

$$\boldsymbol{M^i u_0} = \sum_{j=1}^{n} c_j \boldsymbol{M^i v}_j = \sum_{j=1}^{n} c_j {\sigma_j}^i \boldsymbol{v}_j \tag{1.5.2}$$

Hence, by using the fact that can obtain $\frac{\sigma_k}{\sigma_1} < 1$ for all k > 1 we obtain,

$$\boldsymbol{M^i u_0} = c_1 {\sigma_1}^i (\boldsymbol{v}_1 + \frac{c2}{c1}\left(\frac{\sigma_2}{\sigma_1}\right)^i \boldsymbol{v}2 + \cdots + \frac{cn}{c1}\left(\frac{\sigma_n}{\sigma_1}\right)^i \boldsymbol{v}n)$$

$$\rightarrow\ c_1 {\sigma_1}^i \boldsymbol{v}_1 \tag{1.5.3}$$

Note that after the iteration we return the normalized version of $\boldsymbol{M^i u_0}$, which is equivalent to

$\frac{c_1{\sigma_1}^i \boldsymbol{v_1}}{||c_1{\sigma_1}^i \boldsymbol{v_1}||} = \boldsymbol{v_1}$ (Roughgarden, 2015).

Thus, we conclude that the power method could obtain the first singular vector of the covariance matrix M, and by definition the first singular value is given by $\sigma_1 = ||\boldsymbol{M v_1}||$.

If we generalized the above result into finding the first k singular vector, we use the following approach[3] (Roughgarden, 2015).

1. Find the first singular vector $\mathbf{v_1}$ using power iteration.
2. Project the centred data orthogonally to the singular vector $\mathbf{v_1}$, by replacing each data point x with $\mathbf{x} - (\mathbf{x} \cdot \mathbf{v1})\mathbf{v1}$.
3. Recurse by finding the first k − 1 singular vectors of the new data (in which the direction $\mathbf{v_1}$ has been removed).

The power iteration method is much more efficient than the naïve covariance method when doing singular value decomposition of a matrix. Hence in real world application, for example python sklearn, power method is the algorithm used for the SVD and PCA packages.

2. Application of principle component analysis algorithm

2.1 PCA algorithm

After introducing the relationship between SVD and PCA, let's formulate the PCA algorithm.

Input: n d-dimensional data points.

Output: n k-dimensional data points (d >> k).

[3] The generalized algorithm is based on the definition of singular vectors, the correctness of such algorithm is exactly the same as the proof of the greedy algorithm in section 1.2

1. Mean normalized the data points. Let μ_i (1 <= i <= d) refers to the mean of each feature, $\mathbf{x}_i$ (1 <= i <= d) refers to each data points as a d dimensional vector. Then we do $x_{ij} = x_{ij} - \mu_j$.
2. Store the normalized data points into an n*d matrix M. Each row corresponds to the transpose of the data vector.
3. Find the covariance matrix $\sum$ of the matrix M.
4. Execute singular value decomposition on the covariance matrix $\sum$, $\sum = VSV^T$ by the power iteration method.
5. Finally, let U be the d * k submatrix formed the first k columns of V, then for each centred data point, its coordinate in the new coordinate system is given by $\boldsymbol{x_i' = U^T x_i}$.

Note that sometimes, we also need to do reconstruction from dimensionality reduced data. Then, for each data points its approximate coordinate that is reconstructed from the low dimensional subspace is given by $\boldsymbol{x_{iapprox} = U x_i'}$ (Ng, 2016).[4]

2.2 Select the number of principle components

A problem that one might be interested in is how many principle components shall I choose to preserve most of the variation of the data. To explain the answer to this question, we introduce the concept of **explained variance**.

In principle component analysis, explained variance or variance retained is defined by the variance of the projected data divided by the variance of the original data.

[4] This is also called the inverse transformation of PCA.

$$explained\ variance = \frac{Var(X_{approx})}{Var(X)} \quad (2.2.1)$$

Here X_{approx} is the approximated data point reconstructed from the low dimensional subspace. Since we have already **centred** the data matrix **M**, the above expression could be written as

$$explained\ variance = \frac{\frac{1}{n-1}\sum_{i=1}^{n} ||x_{i(approx)}||^2}{\frac{1}{n-1}\sum_{i=1}^{n} ||x_i||^2} \quad (2.2.2)$$

Using 1.2.2 and 1.2.3, we rewrite 2.2.2 as

$$explained\ variance = \frac{\frac{1}{n-1}\sum_{j=1}^{k} ||Mv_j||^2}{\frac{1}{n-1}\sum_{i=1}^{d} ||Mv_i||^2} = \frac{\sum_{j=1}^{k} ||Mv_j||^2}{\sum_{i=1}^{d} ||Mv_i||^2} = \frac{\sum_{j=1}^{k} {\sigma_j}^2}{\sum_{i=1}^{d} {\sigma_i}^2} \quad (2.2.3).$$

Where $\mathbf{v_i}$ refers to ith singular vector of M, and σ_i refers to the ith singular value of M.

Then by (1.4.2), 2.2.3 is equivalent to

$$explained\ variance = \frac{\sum_{i=1}^{k} s_i}{\sum_{i=1}^{d} s_i} \text{ (Ng, 2016)} \quad (2.2.4).$$

Where $\mathbf{s_i}$ refers to the ith singular value of the **covariance matrix of M**.

Since $\mathbf{s_i}$ are non-negative, the explained variance is monotonically increasing with respect to the number of selected principle components k. Then if we want to preserve p% of the data, the number of principle components selected should be the minimum k such that $\frac{\sum_{i=1}^{k} s_i}{\sum_{i=1}^{d} s_i} \geq p\%$.

2.3 Apply PCA on machine learning examples

Principle component analysis is a frequently used technique for **dimensionality reduction** in real world machine learning problems. In this section, let's firstly apply PCA on a 4 dimensional dataset Iris in order to visualize how PCA reduces the dimension of the data. Then let's go through a more practical example which use PCA in image compression.

2.3.1 Visualize PCA on Iris dataset

Let's firstly visualize the dimensionality reduction effect of PCA on the Iris dataset. Iris dataset is a build-in toy dataset of python's sklearn library. This dataset contains 150 samples, which each sample is described by a 4-dimensional vector that represents its sepal length, sepal length, petal length and petal width. The target of the dataset is a 150-dimension vector which each entry represents the type of the corresponding iris (0 represents Iris setosa, 1 represents Iris virginica and 2 represents Iris versicolor). The goal is to classify the type of an iris given the 4 features describing it.

We firstly import the dataset and all the modules we required for the visualization.

```
iris = datasets.load_iris()
data = pd.DataFrame(iris.data, columns=['sepal length (cm)','sepal width (cm)','pet
al length (cm)','petal width (cm)'])
data['target_names'] = iris.target
dataX = data.drop(columns=['target_names'])
dataY = data['target_names']
```

Next, we split the dataset into train set and test set which the radio of the training data and testing data is 4 to 1.

```
X_train, X_test, y_train, y_test = train_test_split(dataX, dataY, test_size=0.2,
random_state=42)
```

After that we fit the training data with the K-nearest neighbour classifier (we set k = 3) and use the classifier to predict our test data.

```
clf = KNeighborsClassifier(n_neighbors=3)
```

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

Finally, we print out the accuracy score of the prediction, and we found out the 3-nearest neighbour classifier has a classification success rate of 100%.

```
acc = accuracy_score(y_pred, y_test)
acc
```

We know that nearest neighbour classifier works best under the scenario where the same type of samples is grouped into clusters. Hence, we conclude for the iris dataset it is very likely that the same type of iris is adjacent to each other.

Let's now visualize how PCA preserves the clusters of the data points when we reduce the dimension of the data from 4 dimensions to 3 dimensions, 2 dimensions and 1 dimension.

We firstly calculate what percentage of variance each principle component retains.

```
pca = PCA()
pca.fit(dataX)
pca.explained_variance_ratio_
out: array([0.92461872, 0.05306648, 0.01710261, 0.00521218])
```

We could observe that for this dataset, the first principle components retain 92.5 % of the variance, the second principle to fourth principle components in total only retain 7.5% of the variance, so that we should expect that the clusters of the data points should be maintained even if we reduce our data points to 1 dimension by PCA algorithm.

Let's observe the projection effect of PCA algorithm when we compress our data from 4-dimensional to 3-dimensional. The code below (Galarnyk, 2017) plots the projection of the

original data points on the 3-dimensional subspace spanned by the first three principle components.

```
fig = plt.figure(1, figsize=(8, 6))
ax = Axes3D(fig, elev=-150, azim=110)
X_reduced = PCA(n_components=3).fit_transform(iris.data)
ax.scatter(X_reduced[:, 0], X_reduced[:, 1], X_reduced[:, 2], c=iris.target,
           cmap=plt.cm.Set1, edgecolor='k', s=40)
ax.set_title("First three PCA directions")
ax.set_xlabel("1st principle component")
ax.w_xaxis.set_ticklabels([])
ax.set_ylabel("2nd principle component")
ax.w_yaxis.set_ticklabels([])
ax.set_zlabel("3rd principle component")
ax.w_zaxis.set_ticklabels([])
plt.show()
```

We could get the following figure.

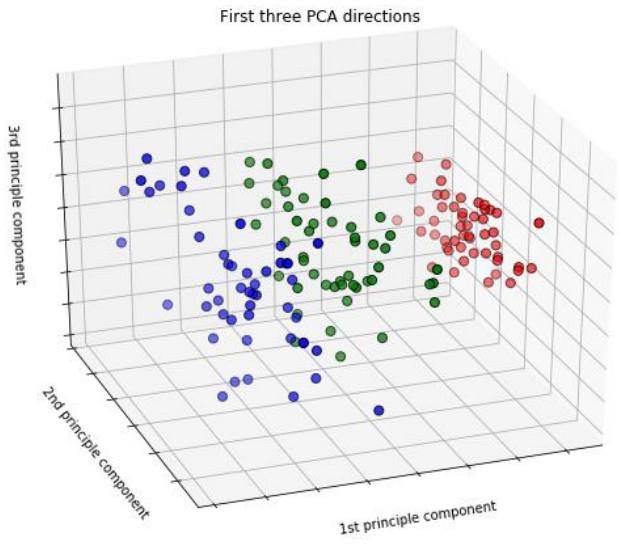


Figure 2.3.1

In this picture we can observed that the projection of the original data points on the subspace spanned by the first three principle components is grouped into three well separated clusters. Since we've already calculated that the first three principle components retain 99.5% of the variance of the data, the result that using PCA to reduce the dimension of the data from 4 to 3 would not vary the distribution of the data too much should be expected.

If we further reduce our data from 4-dimension to 2-dimention and 1-dimensional respectively by applying PCA algorithm, we could get the following result.

```
# plotting the projection of the data on the first two principle components
fig = plt.figure(1, figsize=(8, 6))
X_reduced = PCA(n_components=2).fit_transform(iris.data)
plt.scatter(X_reduced[:, 0], X_reduced[:, 1], c=iris.target,
            cmap=plt.cm.Set1, edgecolor='k', s=40)
plt.title("First two PCA directions")
plt.xlabel("1st principle component")
plt.ylabel("2nd principle component")
```

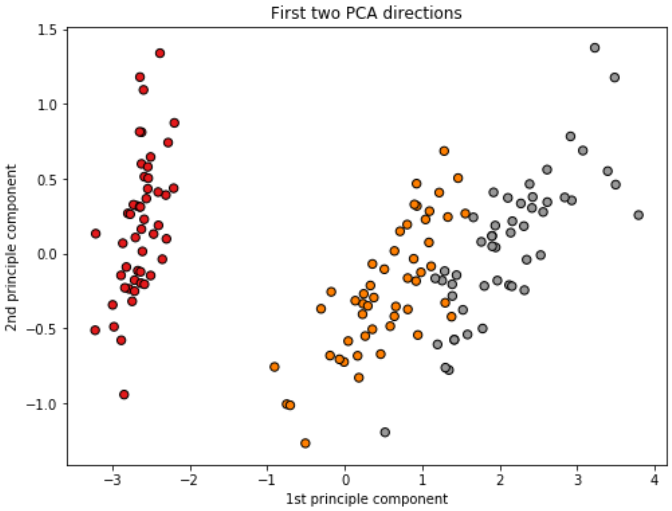


Figure 2.3.2

```
# code for plotting the projection of the data on the first principle component
fig = plt.figure(1, figsize=(8, 6))
X_reduced = PCA(n_components=1).fit_transform(iris.data)
plt.scatter(X_reduced[:, 0], y=np.zeros(150),c=iris.target,cmap=plt.cm.Set1, edgeco
lor='k', s=40)
plt.title("First PCA direction")
plt.xlabel("1st principle component")
```

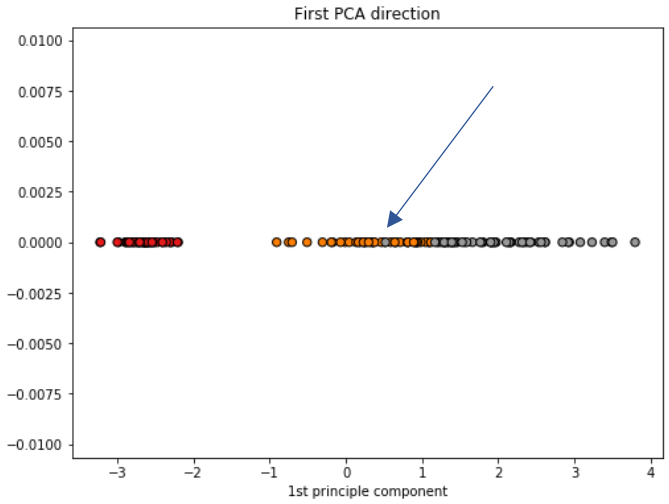


Figure 2.3.3

We could observe that in both cases the projected data still satisfy the property that samples of the same type are tended to group together. However, we can also notice that the distance between samples gets closer as the dimension of the data decreases. For example, in the 1-dimensional cases, the margin between the orange points and black points almost disappears and there exist items that might be misclassified by the 3-nearest neighbour algorithm (the one that is pointed by the arrow).

If we run the 3-nearest neighbour algorithm on the 2-dimensional and 1-dimensional data respectively, we get a prediction accuracy of 100% and 93.3% respectively.

Summary of results on the Iris dataset		
Dimension of the data	Variance retained	3-NN algorithm accuracy
4	100%	100%
3	99.5%	100%
2	97.8%	100%
1	92.5%	93.3%

To summarize, we could observe that for the iris dataset, if we only preserve 97.8% of the variance of the data which is project our 4-dimensional data to a 2-dimensional subspace spanned by the first two principle components, the performance of the classification

algorithm remains to be 100%. In other words, with the use of principle component analysis, on this particular dataset, we could halve our memory usage and reduce the runtime of our classification algorithm without decrease the prediction accuracy.

2.3.2 Apply PCA in faces recognition

From the above section, we visualize the effect of PCA algorithm on dimensionality reduction. However, the Iris dataset contains only 4 features and 150 data points, it is very hard to show the importance of principle component analysis when dealing with high dimensional data. In this section, we are going to show the application of principle component analysis in face recognition.

The problem setting of face recognition is given a database of existing faces and a new face, report the person's name of the new face. In this section, we are going to use the fetch_lfw_people dataset of python sklearn. We only import the pictures of people that have at least 60 different pictures.

```
faces = fetch_lfw_people(min_faces_per_person=60)
```

The imported eigenfaces dataset contains 1348 samples, each sample corresponds to a picture of one of the 8 people's face (['Ariel Sharon' 'Colin Powell' 'Donald Rumsfeld' 'George W Bush', 'Gerhard Schroeder' 'Hugo Chavez' 'Junichiro Koizumi' 'Tony Blair']). The labels (integer of 0 to 7) are contained in faces.target. The features are described by a 62*47 matrix where each entry contains a number between 0 to 255 that represents the greyness of the corresponding pixel (Lavrenko, 2014).

We start with the pre-processing step of the PCA algorithm, we **centred** the data to ensure that each feature has zero-mean.

```
# A is the vector that stores the mean of each feature
A = faces.data.mean(axis=0)
mnz = np.array(faces.data)
for i in range(faces.data.shape[0]):
        # subtract the mean of each feature
         mnz[i] = mnz[i] - A
```

Next, let's apply PCA on the normalized data and then we plot the reconstructed images formed by the first 40 principle components.

```
pca = PCA()
pca.fit(X)
fig, axes = plt.subplots(4, 10, figsize=(10, 7.5),subplot_kw={'xticks':[], 'yticks
':[]},gridspec_kw=dict(hspace=0.1, wspace=0.1))
for j, ax in enumerate(axes.flat):
        ax.imshow(pca.components_[j].reshape(62, 47), cmap='bone')
```

Figure 2.3.4

By the definition of principle components, these are the directions where the data deviates the most from the mean. We can observe that the first 5 to 10 principle components mainly capture the shape of the face while later principle components exaggerate the difference of certain features of the face such as nose, eyes or mouth.

Similarly, if we plot the reconstructed images formed by the last 40 principle components, we could observe that all of the pictures are completely grey, which shouldn't be very useful for the classification task, hence we could interpret them as noise in the data.

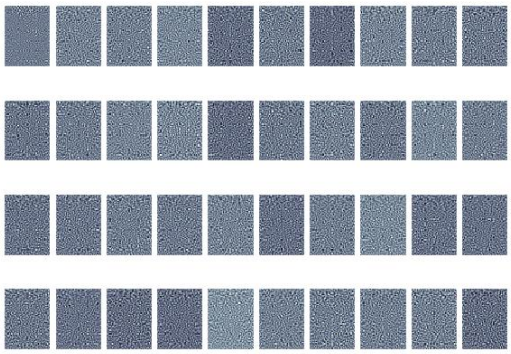


Figure 2.3.5

Since each sample image is described by 2914 features, it seems impossible to capture the distribution of the data by keeping only 2 to 3 of the principle components, let's plot the cumulative explained variance of the data with respect to the total number of principle components and work out what percentage of variance each principle component retain.

```
arr = np.cumsum(pca.explained_variance_ratio_)
fig, ax = plt.subplots( nrows=1, ncols=1 )
plt.plot(arr)
plt.xlabel('number of components')
plt.ylabel('cumulative explained variance')
```

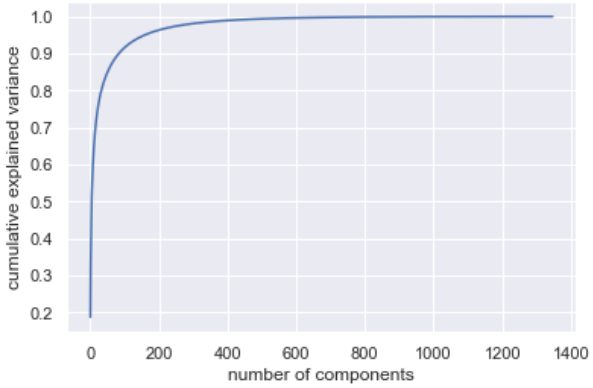


Figure 2.3.6

We observe that the first 400 principle components retain almost 100% of the variance, thus we might claim that using only 400 instead of 2914 features should be sufficient to describe the characteristic of the data.

Let's verify the above claim by measuring how the same machine learning algorithm performs on the dimensionality reduced dataset and the original dataset. We split the data into 7/8 training and 1/8 testing.

```
X = mnz
Y = faces.target
X_train, X_test, y_train, y_test = train_test_split(X, Y, test_size=0.125, random_s
tate=0)
```

Then, we fit a logistical regression CV classifier[5] to the original 2914-dimensional data and display the accuracy score and model training time.

```
clf = LogisticRegressionCV(cv=5, multi_class='multinomial')
start = time.time()
clf.fit(X_train, y_train)
pred = clf.predict(X_test)
end = time.time()
print(accuracy_score(pred, y_test))
```

After that we apply PCA algorithm to the training set. We set the retained variance percentage as 85%, 90%, 95% and 99% respectively. For each of the four settings, we fit the logistical regression CV classifier to the dimensionality-reduced data and record the accuracy score as well as the model training time.

```
l = [0.85, 0.90, 0.95, 0.99]
for i in l:
    print("retain ", i * 100, "% of the variance ")
    ncp = np.searchsorted(arr, i)
    print("component number= ", ncp)
```

[5] The logistical CV classifier is a modification version of the logistical regression classifier, which use cross-validation to select the best-fit hyperparameters for model.

```
    pca = PCA(n_components = ncp, svd_solver='randomized')
    pca.fit(X_train)
    X_train_reduce = pca.transform(X_train)
    X_test_reduce = pca.transform(X_test)
    start = time.time()
    clf.fit(X_train_reduce, y_train)
    pred = clf.predict(X_test_reduce)
    end = time.time()
    print("time consumed ", end - start, " accuracy score ", accuracy_score(pred, y
_test))
    print()
```

After all the above procedure has finished, we could get the following result.

Number of features/components	Variance retained (0 to 1)	Training time (s)	Accuracy score (0 to 1)
2914	1.00	39.69	0.811
412	0.99	8.03	0.817
159	0.95	5.41	0.734
83	0.90	6.21	0.751
51	0.85	5.03	0.740

We observed that by preserving less than 95% of the variance, the accuracy score of the logistical regression classifier significantly decrease. However, if we preserve 99% of the variance by projecting our 2914-dimensional data to a 412-dimensional subspace using PCA, the logistical regression classifier only uses 1/5 of the training time to achieve a classification score even slightly higher than fitting logistical regression on the original data.

We've already seen from figure 2.3.4 and figure 2.3.5 that on the face dataset the first few hundred principle components focus on exaggerating the main characteristic of faces (i.e. shape, mouse, nose, eyes) while the last few hundred principle components could be interpreted as noise in the data. Hence, we conclude that applying PCA on the face dataset could remove the noise without losing important characteristic of the data. This would avoid the logistical regression classifier fitting the noise and as a result the classification accuracy score increase slightly.

Finally, in order to make the above result even more concrete, let's reconstruct the image from dimensionality reduced data. Here, we need to do inverse transform on the dimensionality reduced data and most importantly we **add back the mean** of each feature that were subtracted before we applied PCA on the original data.

```
li = []
for i in range(faces.data.shape[0]):
    li.append(A)
mn_row = np.array(li)
mn_row
l = [0.85, 0.90, 0.95,0.99]
fig, ax = plt.subplots(len(l) + 1, 10, figsize=(10, 7.5),subplot_kw={'xticks':[], '
yticks':[]},gridspec_kw=dict(hspace=0.1, wspace=0.1))
for i in range(len(l)):
    ncp = np.searchsorted(arr, l[i])
    pca = PCA(n_components = ncp, svd_solver='randomized')
    pca.fit(X)
    components = pca.transform(X)
    projected = mn_row + pca.inverse_transform(components)
    for j in range(10):
        ax[i, j].imshow(projected[j].reshape(62, 47), cmap='binary_r')

ax[len(l), 0].set_ylabel('full-dim\ninput')
for i in range(0, len(l)):
    string = str(np.searchsorted(arr, l[i])) + '-dim\nrecon'
    ax[i, 0].set_ylabel(string)
for j in range(10):
        ax[len(l), j].imshow(faces.data[j].reshape(62, 47), cmap='binary_r')
```

We only plot the first 10 images in the dataset and each row corresponds to the images that are reconstructed from the low-dimensional space which preserves 85%, 90%, 95%, 99% variance respectively. We compare the reconstructed images with the original images that comprise 2914 features and get the following result.

Figure 2.3.7

We could observe that when preserving only 85% or 90% of the variance (51 or 83 principle components), the reconstructed image looks fuzzy, but the skeleton of the faces has already been outlined. And if we just increase the number of principle components to 159, which is equivalent to preserving 95% of the variance, most of the main components (shape, nose, eyes, mouth) of faces could be recognized in the reconstructed image. If we further increase the number of principle components to 412 which 99% of the variance is retained, we could not tell too much of the difference between the reconstructed image and the original image since the reconstructed image retains not only the main components of the faces but also the sharpness of the image. This result validates our early claim that the first several principle components exaggerate the shape and main feature of the faces while the rest of principle components focus on the sharpness and brightness of the image.

To summarize, in this section we applied PCA to the face recognition dataset and showed how PCA algorithm compress and reconstruct an image. We derived that on the fetch_lfw_people dataset, if we compressed the original data to the subspace spanned by the

first 412 principle components, we could achieve 0.6% higher classification accuracy with only 1/7 of memory usage and 1/5 model training time. Even if in the real word example, face recognition is much harder than recognizing eigenfaces, the advantage of PCA on reducing memory usage and runtime of a machine learning algorithm could be seen from the example in this section.

Conclusion

In conclusion, this essay firstly outlined the mathematical interpretation of principle component analysis which include its definition, its close relationship with singular value decomposition. Then, we visualized the dimensionality reduction effect of PCA on the Iris dataset. And finally, we applied PCA to a more practical face recognition example which demonstrates how PCA increases the efficiency and reduces memory usage of a machine learning algorithm.

References

Avrim Blum, J. H. (2018). *Foundations of Data Science.*

developers, s.-l. (n.d.). *The Iris Dataset.* Retrieved from scikit-learn: https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

Galarnyk, M. (2017, 12 5). *PCA using Python (scikit-learn).* Retrieved from Towards Data Science: https://towardsdatascience.com/pca-using-python-scikit-learn-e653f8989e60

Lavrenko, V. (2014). PCA 10: eigen-faces [Recorded by V. Lavrenko]. Retrieved from https://www.youtube.com/watch?v=_lY74pXWlS8

Ng, A. (2016). Machine Learning [Recorded by A. Ng]. Retrieved from https://www.youtube.com/watch?v=Zbr5hyJNGCs&list=PLLssT5z_DsK-h9vYZkQkYNWcItqhlRJLN&index=81

Quandt, R. E. (n.d.). Some Basic Matrix Theorems. Retrieved from http://www.quandt.com/papers/basicmatrixtheorems.pdf

Roughgarden, T. (2015, 4 15). *Lecture #8: PCA and the Power Iteration Method.* Retrieved from CS168: The Modern Algorithmic Toolbox: http://theory.stanford.edu/~tim/s15/l/l8.pdf

Shalizi, C. (2012). Retrieved from https://www.stat.cmu.edu/~cshalizi/uADA/12/lectures/ch18.pdf

Shalizi, C. R. (2019). *Advanced Data Analysis from an Elementary Point of View.*

Taboga, M. (n.d.). *Covariance matrix.* Retrieved from StatLect: https://www.statlect.com/fundamentals-of-probability/covariance-matrix

Williams, A. (2016, 3 27). *Everything you did and didn't know about PCA.* Retrieved from Its Neuronal: http://alexhwilliams.info/itsneuronalblog/2016/03/27/pca/

Zichen, W. (2019, 3 17). *PCA and SVD explained with numpy*. Retrieved from towardsdatascience: https://towardsdatascience.com/pca-and-svd-explained-with-numpy-5d13b0d2a4d8