Solution of the Cholesky question: we have there $a_{11} = r_{11}^2 \implies r_{11} = \sqrt{a_{11}}$ then $a_{21} = r_{11}r_{21} \implies r_{21} = \frac{a_{21}}{r_{11}}$ (we use r_{11} directly as it is already solve in the first step.) Next, $a_{22} = r_{21}^2 + r_{22}^2 \implies r_{22} = \sqrt{a_{22} - r_{21}^2}$. Going like this, we get a general result

$$r_{j,j} = \sqrt{A_{j,j} - \sum_{k=1}^{j-1} r_{j,k}^2}, \quad r_{i,j} = \frac{1}{r_{j,j}} \left(a_{i,j} - \sum_{k=1}^{j-1} r_{i,k} r_{j,k} \right) \quad \text{for } i > j.$$

In the above stated general formulas, do you see where the default `chol()` function might throw an error?

Say we want to solve $\mathbf{Ax} = \mathbf{b}$. Let $\mathbf{A}$ is lower triangular, then what is $\mathbf{x}$? We can use 'Forward substitution'. Similarly, if $\mathbf{A}$ is uppper triangular, we use 'back substitution'.

Now, a little more general case where $\mathbf{A}$ is symmetric. We apply Cholesky and get $\mathbf{A} = \mathbf{LL}^T$. We can now solve $\mathbf{Ax} = \mathbf{b}$ in two steps:

First we solve $\mathbf{Ly} = \mathbf{b}$ for $\mathbf{y}$, using forward solve and then $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$ using back solve. This way, we never need to invert $\mathbf{A}$. Inversion is one of the most costly operations. R has `backsolve` and `forwardsolve` implementations.

If $\mathbf{A}$ is not symmetric, we can use LU decomposition for the same.

A linear regression model $y_i = \mathbf{x}_i\boldsymbol{\beta} + \epsilon_i$ has a matrix-vector representation $\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\epsilon}$, where $\mathbf{X}$ is a $n \times p$ matrix whose $i\text{-}th$ row is $\mathbf{x}_i$. We use this representation very often.

In the least square regression, we estimate $\boldsymbol{\beta}$ as the minimizer of $\sum_{i=1}^{n}(y_i - \mathbf{x}_i\boldsymbol{\beta})^2$ and in matrix notation, the objective function becomes $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})$ (verify that they are the same!).

Expanding this expression we have, $(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^T(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = \mathbf{y}^T\mathbf{y} + \boldsymbol{\beta}^T\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - 2\boldsymbol{\beta}^T\mathbf{X}^T\mathbf{y}$. Now to minimize, we take the derivative of this objective function with respect to $\boldsymbol{\beta}$ which gives $2(\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} - \mathbf{X}^T\mathbf{y})$ and equate it to zero.

Thus, we need to solve $\mathbf{X}^T\mathbf{X}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}$ for $\boldsymbol{\beta}$. You may all be familiar with `lm` in R to fit linear regression. R actually uses QR decomposition of $\mathbf{X}$ for the above purpose. Then if $\mathbf{X} = \mathbf{QR}$, then $\mathbf{R}^T\mathbf{R}\boldsymbol{\beta} = \mathbf{X}^T\mathbf{y}$. For RHS, we need to compute product (we can avoid `%*%` using some of the techniques from HW). But we do not need to invert $\mathbf{R}^T\mathbf{R}$. Instead, we apply forward and back solve. Note that R uses Householder transform to compute QR,

not Gram–Schmidt. Householder is faster. I will give a separate note on this as it requires some background on reflection using orthogonal matrices.

What is an orthogonal or orthonormal matrix? If for a matrix $\mathbf{Q}$, we have $\mathbf{Q}^T\mathbf{Q} = \mathbf{Q}\mathbf{Q}^T = \mathbf{I}$, it is called orthonormal. However, most places consider this as the definition of orthogonal as well. In general, orthogonal means $\mathbf{Q}^T\mathbf{Q}$ and $\mathbf{Q}\mathbf{Q}^T$ lead to diagonal matrices, and may not be identity.

The linear regression implementation in SAS uses the sweep operator to solve the above system of equations. [1] `https://www.jstor.org/stable/2683825` has an excellent demonstration of this method. This can also be used to compute the inverse of any matrix. This is also called Gauss-Jordan elimination.

Dimension reduction PCA, SVD, CCA and other Clustering methods

In today's class, we study 'unsupervised' dimension reduction methods. Hence, we have a multivariate response but no predictors. We aim to identify low dimensional dependencies.

There are several popular dimension reduction approaches, which could classified into two subclasses, linear and non-linear. Linear methods:

- Principal component analysis (PCA)
- Canonical correlation analysis (CCA)
- Matrix factorization (MF)
- Non-negative matrix factorization (NMF)
- Linear factor analysis

Non-Linear methods:

- Kernel principal component analysis
- Multi dimension scaling (MDS)
- T-Distributed Stochastic Neighbor Embedding (tSNE)
- Autoencoders
- Gaussian process latent variable model (GPLVM).

We will only study the basic ones in the class, and I will just mention the others.

Eigen decomposition

Eigen decomposition is sometimes also called 'diagonalization'. Eigen decomposition by construction is only to square matrices, i.e. $n \times n$ matrices. Whereas SVD is applicable to any $m \times n$ matrix. When an eigendecomposition is computed for a square matrix $\mathbf{A}$, we try to get a representation $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{U}^T$, where $\mathbf{U}$ is an orthonormal matrix i.e. $\mathbf{U}\mathbf{U}^T = \mathbf{I}$. And $\mathbf{D}$ is a diagonal matrix. When the matrix being factorized is a normal or real symmetric matrix, the decomposition is called "spectral decomposition", derived from the spectral theorem.

We can also rewrite the eigendecomposition as $\mathbf{U}\mathbf{D}\mathbf{U}^T = \sum_i d_i \mathbf{u}_i \mathbf{u}_i^T$, where $\mathbf{u}_i$ is the *i-th* column of $\mathbf{U}$ and d_i is *i-th* diagonal entry in $\mathbf{D}$. (In general, we can write $\mathbf{A}\mathbf{B} = \sum_i \mathbf{a}_i \mathbf{b}_i^T$, where $\mathbf{a}_i$ is *i-th* column of $\mathbf{A}$ and $\mathbf{b}_i$ is the *i-th* row of $\mathbf{B}$. This is a very useful result.)

A (nonzero) vector $\mathbf{v}$ of dimension n is an eigenvector of a square $n \times n$ matrix $\mathbf{A}$ if it satisfies a linear equation of the form $\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$. This equation is called the eigenvalue equation or the eigenvalue problem.

Characteristic polynomial is defined by $P(\lambda) = \det(\mathbf{A} - \lambda\mathbf{I})$ and eigenvalues are different roots of this polynomial i.e. $P(\lambda') = 0$ is λ' is an eigenvalue of $\mathbf{A}$. Thus, for each eigenvalue λ_i, we have $\det(\mathbf{A} - \lambda_i\mathbf{I})$. When determinant is a matrix is zero, it has a non-empty null-space i.e. there exist a vector $\mathbf{v} \neq 0$ such that $(\mathbf{A} - \lambda_i\mathbf{I})\mathbf{v} = 0 \implies \mathbf{A}\mathbf{v} = \lambda_i\mathbf{v}$. Here $\mathbf{v}$ becomes the eigenvector. Hence, solving the characteristic function above, we can obtain all the eigenvalues.

Properties of orthogonal matrices

Using the eigendecomposition, we can easily compute any power of a matrix easily as $\mathbf{A}^k = \mathbf{U}\mathbf{D}^k\mathbf{U}^T$, where $\mathbf{D}^k$ is essentially a diagonal matrix with all the entries raised to the power k. For any analytic function (whose polynomial expansion exists like $\sin, \cos, \log, \exp$), we can use eigendecomposition when they are applied on a matrix. Like $\exp(\mathbf{A}) = \mathbf{U}\exp(\mathbf{D})\mathbf{U}^T$ or $\log(\mathbf{A}) = \mathbf{U}\log(\mathbf{D})\mathbf{U}^T$. Here exp or log are NOT elementwise operations on the matrix. These are Matrix-exponential and Matrix-log, respectively. Other than using eigendecomposition, there is no other way to compute these easily.

However, for non-analytic function like $\exp(-x^{-1})$, when applied on $\mathbf{A}$, eigendecomposition will not be able to help.

Other usages of eigenvalues: Given any $n \times n$ real or complex matrix $\mathbf{A}$, we have $\text{tr}(\mathbf{A}) = \sum_{i=1}^n \lambda_i$ and $\text{determinant}(\mathbf{A}) = \prod_i \lambda_i$

Some characterizations of square symmetric matrices: If for a square symmetric matrix $\mathbf{A}$, we have $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$, the matrix is called Positive definite (pd) matrix. When $\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0$ for all $\mathbf{x} \neq \mathbf{0}$ the matrix is called Positive semi-definite (psd/nnd) matrix.

If for a square symmetric matrix $\mathbf{A}$, we have $\mathbf{x}^T\mathbf{A}\mathbf{x} < 0$ for all $\mathbf{x} \neq \mathbf{0}$, the matrix is called Negative definite (nd) matrix. When $\mathbf{x}^T\mathbf{A}\mathbf{x} \leq 0$ for all $\mathbf{x} \neq \mathbf{0}$ the matrix is called Negative semi-definite (nsd) matrix.

For a pd matrix, the eigenvalues are all positive. Covariance matrices are pd. Thus, the eigenvalues are all positive (i.e. $\lambda_i > 0$). The Cholesky decomposition only works for pd matrices. For psd matrices, the eigenvalues are all non-negative (i.e. $\lambda_i \geq 0$).

For symmetric matrices, we can consider a useful variation of $\mathbf{UDU}^T$ which is very popularly known as 'spectral decomposition'. That is $\mathbf{A} = \mathbf{UDU}^T = \sum_{i=1}^{n} d_{ii}\mathbf{u}_i\mathbf{u}_i^T$, where $\mathbf{u}_i$ is *i-th* column of $\mathbf{U}$. Hence, eigen decomposition gives us a basis expansion for $\mathbf{A}$. Let $d_{11} \geq d_{22} \geq \ldots \geq d_{nn} > 0$ (for pd matrices). Then $\mathbf{B} = \sum_{i=1}^{K} d_{ii}\mathbf{u}_i\mathbf{u}_i^T$ is an approximation of $\mathbf{A}$, containing $100\frac{\sum_{i=1}^{K} d_{ii}}{\sum_{i=1}^{n} d_{ii}}\%$ of information.

QR algorithm for eigenvalue computation

```
A <- matrix(rnorm(100), 10, 10)
A <- (A+t(A))/2

A0 <- A
U <- diag(10)
i <- 1
r <- max(range(abs(A[lower.tri(A)])))
while(r > 1e-10){
  o <- qr(A)
  Q <- qr.Q(o)
  R <- qr.R(o)

  A <- R %*% Q
  U <- U %*% Q
  r <- max(range(abs(A[lower.tri(A)])))
}
sort(diag(A))
sort(eigen(A0)$values)
```

$\mathbf{A}_0 \rightarrow \mathbf{A}_1 \rightarrow \mathbf{A}_2 \cdots \rightarrow \mathbf{A}_k \rightarrow$. We continue until $\mathbf{A}_k$ is upper triangular since for upper triangular matrix, QR solution is trivial with $\mathbf{R}_k = \mathbf{A}_k$. By construction, `R %*% Q` becomes upper triangular, if it is diagonal.

Explanation: $\mathbf{A}_k = \mathbf{R}_k\mathbf{Q}_k = \mathbf{Q}_k^T\mathbf{A}_{k-1}\mathbf{Q}_k$ since $\mathbf{A}_{k-1} = \mathbf{Q}_k\mathbf{R}_k$. Extending above equation we get $\mathbf{A}_k = \mathbf{Q}_k^T\mathbf{Q}_{k-1}^T \cdots \mathbf{Q}_1^T\mathbf{A}_0\mathbf{Q}_1 \cdots \mathbf{Q}_{k-1}\mathbf{Q}_k$. Hence, it approximates eigenvalues and eigenvectors.

Statistical usage of eigen-decomposition:

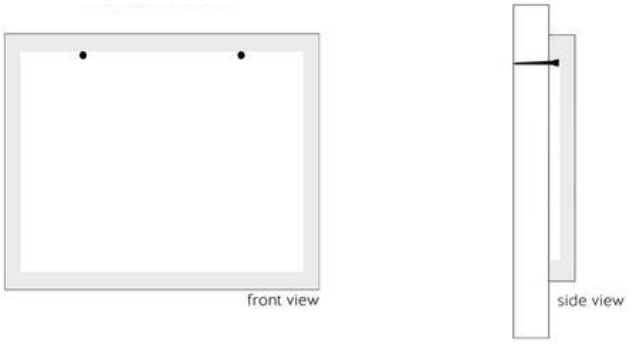
Check the following quick example for motivation:

```
Sigma <- matrix(c(1,0.7,0.7,1), 2, 2)

n <- 100

X <- mvtnorm::rmvnorm(n, sigma = Sigma)

plot(X[,1],X[,2])

U <- eigen(Sigma)$vectors

X1 <- X %*% U

plot(X1[,1],X1[,2])
```

Principal component analysis (PCA): The principal components of a collection of points in a real coordinate space are a sequence of p unit vectors, where the i-th vector is the direction of a line that best fits the data while being orthogonal to the first $i-1$ vectors. Here, a best-fitting line is defined as one that minimizes the average squared distance from the points to the line. These directions constitute an orthonormal basis in which different individual dimensions of the data are linearly uncorrelated. Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest.

Although the above description sounds very complex and cryptic, it is very easy to compute principal components using eigen-decomposition. PCA is a dimension reduction technique. It is thus important to understand how the variables of the input data set are varying from the mean with respect to each other, or in other words, to see if there is any relationship between them. Because sometimes, variables are highly correlated in such a way that they contain redundant information. So, in order to identify these correlations, we compute the covariance matrix.

Principal components are new variables that are constructed as linear combinations or mixtures of the initial variables. These combinations are done in such a way that the new variables (i.e., principal components) are uncorrelated and most of the information within the initial variables is squeezed or compressed into the first components. So, the idea is p-dimensional data gives you p principal components, but PCA tries to put maximum



possible information in the first component, then the maximum remaining information in the second and so on.

Geometrically speaking, principal components represent the directions of the data that explain a maximal amount of variance, that is to say, the lines that capture most information of the data. Take Figure **??**, for example which presents two ways of looking at a flat object such as whiteboard, TV, monitor, etc. If someone asks the size of such an object, the front view provides the most realistic assessment. The front view has some sort of a greatest variability. In the case of visualizing multivariate data too, the same thing holds. We should view the data from the directions of greatest variabilities as we can only plot up to 3-D.

The relationship between variance and information here, is that, the larger the variance carried by a line, the larger the dispersion of the data points along it, and the larger the dispersion along a line, the more information it has. To put all this simply, just think of principal components as new axes that provide the best angle to see and evaluate the data, so that the differences between the observations are better visible.

The covariance matrix of a p-dimensional multivariate data quantifies the overall variability of the dataset. Let $\mathbf{S}$ be the covariance matrix. Then eigendecomposition gives us the decomposition $\mathbf{S} = \sum_{i=1}^{p} \lambda_i \mathbf{u}_i \mathbf{u}_i^T$, where λ_i's are the eigenvalues and $\mathbf{u}_i$'s are the eigenvectors. Here, we let $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p > 0$. Hence, $\mathbf{u}_1$ is the first axis that explains the largest variability. Due to orthogonality, rest of the vectors $\{\mathbf{u}_2, \ldots, \mathbf{u}_p\}$ are orthogonal to $\mathbf{u}_1$. Hence, from the space orthogonal to $\mathbf{u}_1$, the largest variance is explained along the direction $\mathbf{u}_2$, and so on.

Original data is on $\mathbb{R}^p$, which has a standard basis $\{\mathbf{e}_1, \ldots, \mathbf{e}_p\}$, where $\mathbf{e}_k$ is the vector of length p with 1 at k-th place and all else are 0. The above eigenvectors $\{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_p\}$ give us an alternative (data-driven) basis set. Viewing the data on this new basis will give a new set of coordinates which are called the principal components.

Let $\mathbf{x}_1, \ldots, \mathbf{x}_n$ are p-dimensional multivariate data. We want to write $\mathbf{x}_k = \sum_{i=1}^{p} a_i \mathbf{u}_i$, where a_1 is the first principal component score, a_2 is the second principal component score, etc. We can choose a K such that $100\frac{\sum_{i=1}^{K}\lambda_i}{\sum_{i=1}^{p}\lambda_i}\% = 90\%$. Then the computed K will explain 90% of the original data. Why is it important? You may encounter $p = 500$, whereas for some data above may hold even for $K = 10$. Hence, we can only work with first 10 eigenvector directions to form 10 principal components. And we approximate original data as $\mathbf{x}_i \approx \mathbf{y}_i = \sum_{j=1}^{K} a_{i,j}\mathbf{u}_j$. Specifically, we can compute $a_{i,j}\mathbf{x}_i^T\mathbf{u}_j$. If we create a matrix of principal scores $\mathbf{A} = \{a_{i,j}\}$ which is of $n \times K$ dimension. We can compute $\mathbf{A} = \mathbf{X}\mathbf{U}_K$, where $\mathbf{U}_K$ is the submatrix with K eigenvectors (Verify this!).

Example of PCA: Project 150 observations of IRIS data with 4 features onto 2 dimensional space

```
head(iris)
dim(iris)

iris.data <- iris[,1:4]
iris.sca  <- scale(iris.data)
ir.pca <- prcomp(iris.sca,
                 center = TRUE,
                 scale = TRUE)
PC1 <- ir.pca$x[,"PC1"]
PC2 <- ir.pca$x[,"PC2"]
variance <- ir.pca$sdev^2 / sum(ir.pca$sdev^2)
v1 <- paste0("variance: ",signif(variance[1] * 100,3), "%")
v2 <- paste0("variance: ",signif(variance[2] * 100,3), "%")
plot(PC1, PC2, col=as.numeric(iris$Species),pch=19, xlab=v1, ylab=v2)
legend("topright", legend = levels(iris$Species), col =  unique(iris$Species), pch = 19)
```

See that PCA automatically produced well-separated principal components without any information on species type.

Kernel PCA Kernel PCA is a non-linear dimension reduction method where the sample covariance matrix is replaced by a covariance function, which is also called covariance kernel. The sample covariance matrix quantifies linear association among the variables. However, a covariance function can quantify non-linear associations too. Specifically, we compute the following matrix,

$$K_{i,j} = K(\mathbf{x}_i, \mathbf{x}_j).$$

Then we apply the same set of methods on $\mathbf{K}$ as before.

In traditional PCA, $K(\mathbf{x}_i, \mathbf{x}_j) = cov(\mathbf{x}_i, \mathbf{x}_j)$ (linear covariance). In Kernel PCA, we generalize it to any possible covariance kernel like, $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|_2^2}{\rho})$ which is called Gaussian kernel due to its similarity with Gaussian density function.

Here $\mathbf{x}_i$ and $\mathbf{x}_j$ are observations corresponding to *i-th* and *j-th* variable (like one could be sepal length and other one could petal length). Thus, each one will be of n, the total number of observations. And $\|\mathbf{x}_i - \mathbf{x}_j\|_2^2 = \sum_{k=1}^{n}(x_{i,k} - x_{j,k})^2$. There are several other choices of kernel and all of them depend on $\|\mathbf{x}_i - \mathbf{x}_j\|_2^2$. A good list of possible kernels can be found at `https://cran.r-project.org/web/packages/kernlab/vignettes/kernlab.pdf`.

Spectral clustering: This is a clustering technique, especially for multivariate data. We learn this technique in the HW in detail.

Singular value decomposition (SVD):

It generalizes the eigendecomposition of a square normal matrix with an orthonormal eigenbasis to any $m \times n$ matrix. It is related to the polar decomposition. Specifically, the singular value decomposition of an $m \times n$ complex matrix $\mathbf{M}$ is a factorization of the form $\mathbf{M} = \mathbf{U\Sigma V^T}$, where $\mathbf{U}$ is an $m \times m$ orthogonal matrix, $\mathbf{\Sigma}$ is an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal, and $\mathbf{V}$ is an $n \times n$ orthogonal matrix.

The diagonal entries in $\mathbf{\Sigma}$ are called the singular values. The number of non-zero singular values is equal to the rank of $\mathbf{M}$. The columns of $\mathbf{U}$ and the columns of $\mathbf{V}$ are called left-singular vectors and right-singular vectors of $\mathbf{M}$, respectively.

Statistical usage of SVD: There are several statistical usages of SVD. We often need to calculate the pseudoinverse of a singular matrix, $\mathbf{A}$ which is defined and unique for all matrices whose entries are real or complex numbers. It can be computed using the singular value decomposition. If $A = U\Sigma V^* A = U\Sigma V^*$ is the singular value decomposition of $\mathbf{A}$, then $A^+ = V\Sigma^+ U^*$. For a rectangular diagonal matrix such as Σ , we get the pseudoinverse by taking the reciprocal of each non-zero element on the diagonal, leaving the zeros in place, and then transposing the matrix. In numerical computation, only elements larger than some small tolerance are taken to be nonzero, and the others are replaced by zeros. For example, in the MATLAB or GNU Octave function pinv, the tolerance is taken to be $t = \epsilon \max(m, n) \max(\mathbf{\Sigma})$, where ϵ is the machine epsilon.

Canonical-correlation analysis: Given two column vectors $X = (x_1, \ldots, x_n)^T$ and $Y = (y_1, \ldots, y_m)^T$ of random variables with finite second moments, one may define the cross-covariance $\Sigma_{XY} = \mathrm{cov}(X, Y)$ to be the $n \times m$ matrix whose (i, j) entry is the covariance $\mathrm{cov}(x_i, y_j)$. In practice, we would estimate the covariance matrix based on sampled data from X and Y (i.e. from a pair of data matrices).

Canonical-correlation analysis seeks vectors $\mathbf{a}$ ($\mathbf{a} \in \mathbb{R}^n$) and $\mathbf{b}$($\mathbf{b} \in \mathbb{R}^m$) such that the random variables $a^T X$ and $\mathbf{b}^T\mathbf{y}$ maximize the correlation $\rho = \mathrm{corr}(\mathbf{a}^T\mathbf{x}, \mathbf{b}^T\mathbf{y})$. The (scalar) random variables $\mathbf{u} = \mathbf{a}^T\mathbf{x}$ and $\mathbf{v} = \mathbf{b}^T\mathbf{y}$ are the first pair of canonical variables. Then one

seeks vectors maximizing the same correlation subject to the constraint that they are to be uncorrelated with the first pair of canonical variables; this gives the second pair of canonical variables. This procedure may be continued up to $\min\{m, n\}$ times.

$(\mathbf{a}', \mathbf{b}') = \underset{\mathbf{a},\mathbf{b}}{\operatorname{argmax}} \operatorname{corr}(\mathbf{a}^T\mathbf{x}, \mathbf{b}^T\mathbf{y})$

To solve the above optimization problem, we can use SVD method. Specifically, we can compute the normalized cross-covariance matrix $\mathbf{\Sigma}_X^{-1/2}\mathbf{\Sigma}_{XY}\mathbf{\Sigma}_Y^{-1/2}$ and apply SVD to get $\mathbf{UDV}^T$. Ignore the smaller entries in $\mathbf{D}$. Similar to PCA, we can compute canonical components for the two types of data. Here, we need to compute canonical scores as $\mathbf{X\Sigma}_X^{-1/2}\mathbf{U}_K$ and $\mathbf{Y\Sigma}_Y^{-1/2}\mathbf{V}_K$ taking only K columns of each basis matrix. Diagonal entries in $\mathbf{D}$ are different canonical correlations.

```
library(CCA)
data(nutrimouse)
X=as.matrix(nutrimouse$gene[,1:10])
Y=as.matrix(nutrimouse$lipid)
res.cc=cc(X,Y)

cor(X%*%res.cc$xcoef[,1], Y%*%res.cc$ycoef[,1])
#matches with
res.cc$cor[1]

cor(X%*%res.cc$xcoef[,2], Y%*%res.cc$ycoef[,2])
#matches with
res.cc$cor[2]

#Canonical correlation components are independent which can be verified below
cor(X%*%res.cc$xcoef)

cor(Y%*%res.cc$ycoef)
```

There is another application of SVD related to alignment.

Matrix factorization First, I try to motivate matrix factorization from the above discussions. If m-dimensional $\mathbf{x}_i \sim \text{MVN}(0, \mathbf{\Sigma})$ for $i = 1, \ldots, n$ and $\mathbf{\Sigma} = \mathbf{UDU}$ is the eigen decomposition. Then, we can get $\mathbf{x}_i = \mathbf{UD}^{1/2}\mathbf{U}\mathbf{y}_i$, where $\mathbf{y}_i \sim \text{Normal}0, \mathbf{I}$. Hence, the data matrix $\mathbf{X} = \mathbf{UD}^{1/2}\mathbf{UY} = \mathbf{WH}$. Here, $\mathbf{W} = \mathbf{UD}^{1/4}$ and $\mathbf{H} = \mathbf{D}^{1/4}\mathbf{UY}$. If some of the eigenvalues are small, we can ignore those and only consider the large ones. We can then

translate $\mathbf{D}^{1/4}$ to a $m \times R$ dimensional $\mathbf{D}'$ when there are R many large eigenvalues. Then $\mathbf{W}$ becomes $m \times R$ and $\mathbf{H}$ becomes $R \times n$.

Now, $\mathbf{\Sigma}$ will often be unknown. Thus, it is more efficient to compute the matrix factorization of $\mathbf{X}$ directly. One way is by applying SVD which gives us $\mathbf{X} = \mathbf{UDV}^T$. Now many of the singular values in $\mathbf{D}$ many be zero or close to zero. Ignoring those, we can get $\mathbf{W}$ and $\mathbf{H}$ in exactly the same way.

However, for positive valued data, the above representation is not very optimal. See, $\mathbf{X} = ((x_{i,j}))$ and $x_{i,j} = \sum_k w_{i,k}h_{k,j}$. Now if $x_{i,j}$ is positive, restricting $\mathbf{W}$ and $\mathbf{H}$ to be positive would force many of these entries to be zero. This will encourage sparsity in $\mathbf{W}$ and $\mathbf{H}$.

Nonnegative matrix factorization: NMF is a new method for nonnegative data analysis. The method considers nonnegative constraint on matrix factorization and has some advantages over traditional PCA, especially in face image analysis. Now it has become a powerful technique for nonnegative data analysis in multiomics, genomics, etc. The primary reason is that there now exists a lot of multivariate data which are positive-valued.

Given an $m \times n$ data matrix $\mathbf{X}$ with nonnegative values for all entries, NMF is to find an approximate decomposition as $\mathbf{X} \approx \mathbf{WH}$, where $\mathbf{W}$ is $m \times R$ and $\mathbf{H}$ is $R \times n$ and $R < \min\{m, n\}$. Hence, NMF attempts to identify a low-dimensional structure in $\mathbf{X}$. Formally, we 'often' minimize the following objective function to get $\mathbf{W}$ and $\mathbf{H}$,

$$\frac{1}{2}\|\mathbf{X} - \mathbf{WH}\|_2^2.$$

I said 'often', as there exists variations where we replace the Frobeneous norm above with few other alternatives. One popular is the Kullback-Leibler divergence or I-divergence. The optimization exercises will come later in the course. Besides, we usually put sparsity inducing structure on $\mathbf{W}$ and $\mathbf{H}$ so that most of the entries in $\mathbf{W}$ and $\mathbf{H}$ will become zero.

Example The first example is due to Dorothy, a PhD graduate from our department. This is from her JSM, 2022 presentation. She applied NMF on her dog's (Hudson) original BW image. Here I show the quality of this approximation with different values of R in Figure 1. The original image is of dimension 947×789 (a huge matrix with 747183 entries). But matrix factorization helped us to represent this image with only $947 \times 80 + 789 \times 80 = 138880$ entries (18% of the original size). It is also thus evident that there are a lot of latent features. Thus, this algorithm is routinely used for problems related to matrix completion.

The second example is due to Netflix challenge where Netflix wanted a good recommender system to predict possible movie ratings by users based on the behavior of observed data which was presented as tuple (user, movie, rating). Netflix posted a huge matrix with a lot of missing entries. There were nearly a million movies. Each user can only provide rating on a handful of those. Thus, once we know the latent relations, it will be easier for

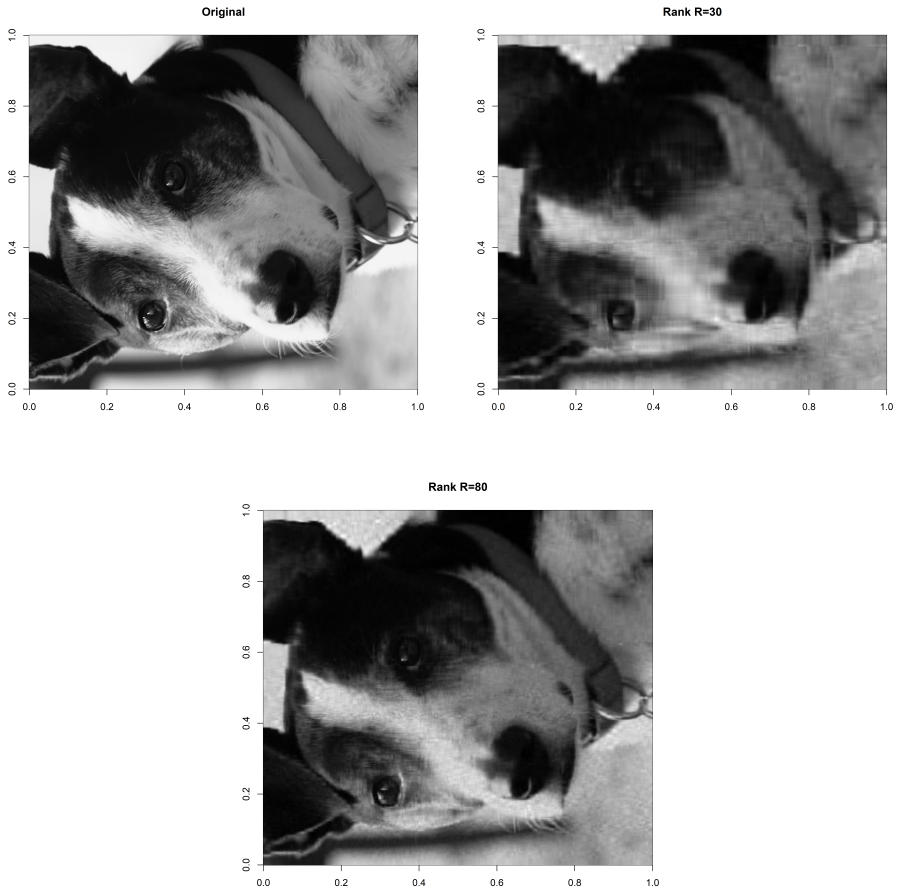


Figure 1: NMF example with a picture of Hudson for two values of R (ignore the axis coordinates).

us to predict the missing portions.

Linear factor model A typical factor model for p-variate data $\mathbf{y}_i = (y_{i1}, \ldots, y_{ip})^T$ from a single group:

$$\mathbf{y}_i = \mathbf{\Lambda}\boldsymbol{\eta}_i + \boldsymbol{\epsilon}_i, \quad \boldsymbol{\eta}_i \sim \mathrm{N}(0, \mathbf{I}_k), \quad \boldsymbol{\epsilon}_i \sim \mathrm{N}(0, \mathbf{\Sigma}), \tag{1}$$

where it is assumed that data are centered prior to analysis, $\boldsymbol{\eta}_i = (\eta_{i1}, \ldots, \eta_{ik})^T$ are latent factors, $\mathbf{\Lambda}$ is a $p \times k$ factor loading matrix, and $\mathbf{\Sigma} = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_p^2)$ is a diagonal matrix of residual variances. Under this model, marginalizing out the latent factors $\boldsymbol{\eta}_i$ induces the covariance $H = \mathrm{cov}(\mathbf{y}_i) = \mathbf{\Lambda}\mathbf{\Lambda}^T + \mathbf{\Sigma}$.

Non-linear latent variable model The linear dependence of the latent factors can be further generalized to a non-linear function like

$$\mathbf{y}_i = f(\boldsymbol{\eta}_i) + \boldsymbol{\epsilon}_i, \qquad \boldsymbol{\epsilon}_i \sim \mathrm{N}(0, \mathbf{\Sigma}). \tag{2}$$

The non-linear function f can be modeled using a Gaussian process, which will lead to Gaussian process latent variable model (GPLVM). Alternatively, neuralnet, B-splines, wavelet and other basis functions could be considered as well.

Distance based methods

Distance based dimension reduction methods are very popular. Spectral clustering, k-means clustering can be considered in this domain too. Here, similarity among the data points is quantified by the distances among them.

Given a set of observations $(x_1, x_2, \ldots, x_n)$, where each observation is a d-dimensional real vector, k-means clustering aims to partition the n observations into $k(\leq n)$ sets $S = S_1, S_2, \ldots, S_k$ to minimize the within-cluster sum of squares (WCSS) (i.e. variance). Formally, the objective is to find:

$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \arg\min_{\mathbf{S}} \sum_{i=1}^{k} |S_i| \operatorname{Var} S_i$ where μ_i is the mean of points in S_i.

This is equivalent to minimizing the pairwise squared deviations of points in the same cluster:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \frac{1}{|S_i|} \sum_{\mathbf{x},\mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2.$$

The equivalence can be deduced from identity $|S_i| \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \boldsymbol{\mu}_i\|^2 = \sum_{\mathbf{x} \neq \mathbf{y} \in S_i} \|\mathbf{x} - \mathbf{y}\|^2$.

Compare the following two outputs

```
fit <- kmeans(scale(iris[, 1:4]), centers=3)
fit$cluster
```

```
#To compute similarities among two possible class labels
aricode::ARI(fit$cluster, as.factor(iris[, 5]))
#Here we are comparing kmeans generated labels with true class label iris[, 5].
```

and

```
fit <- kmeans(iris[, 1:4], centers=3)
fit$cluster

#To compute similarities among two possible class labels
aricode::ARI(fit$cluster, as.factor(iris[, 5]))
#Here we are comparing kmeans generated labels with true class label iris[, 5].
```

For clustering, `https://scikit-learn.org/stable/modules/clustering.html#clustering` has many functions. Almost all of them are distance-based. R has different packages for these tasks. To my knowledge, there is no unified package.

What is MDS? MDS (multidimensional scaling) is an algorithm that transforms a dataset into another dataset, usually with lower dimensions, keeping the same Euclidean distances between the points.

Let distance function among the datapoints (which are p-dimensional) is defined as, $d_{i,j} :=$ distance between i-th and j-th objects. These distances are the entries of the dissimilarity matrix

$D := \begin{pmatrix} d_{1,1} & d_{1,2} & \cdots & d_{1,M} \\ d_{2,1} & d_{2,2} & \cdots & d_{2,M} \\ \vdots & \vdots & & \vdots \\ d_{M,1} & d_{M,2} & \cdots & d_{M,M} \end{pmatrix}$. The goal of MDS is, given $\mathbf{D}$, to find $\mathbf{M}$ vectors $\mathbf{x}_1, \ldots, \mathbf{x}_M \in \mathbb{R}^N$ and $N < p$ such that

$\|\mathbf{x}_i - \mathbf{x}_j\| \approx d_{i,j}$ for all $i, j \in 1, \ldots, M$, where $\|\cdot\|$ is a distance norm. It is usually taken as the Euclidean distance, but, in a broader sense, it may be a metric or arbitrary distance function.

Various approaches are available to determining the vectors $\mathbf{x}_i$. Usually, MDS is formulated as an optimization problem, where $(\mathbf{x}_1, \ldots, \mathbf{x}_M)$ is found as a minimizer of some cost function, for example, $\underset{x_1,\ldots,x_M}{\operatorname{argmin}} \sum_{i<j} (\|x_i - x_j\| - d_{i,j})^2$.

The above loss function can be dominated by large distances, thereby ignoring the local structure for pairs of short distances. A possible modification is to use $\underset{x_1,\ldots,x_M}{\operatorname{argmin}} \sum_{i<j} \left(\frac{\|x_i - x_j\| - d_{i,j}}{d_{i,j}} \right)^2$..

Another possibility is using Sammon's stress, which helps to balance effects of larger distances and identifying local structures. This is given by,

$$\underset{x_1,\ldots,x_M}{\operatorname{argmin}} \sum_{i<j} \left(\frac{\|x_i - x_j\| - d_{i,j}}{d_{i,j}} \right)^2 w_{i,j}.,$$

where $w_{i,j} = \frac{d_{i,j}}{\sum_{i<j} d_{i,j}}$. Details on this optimization can be found at `https://www.stat.pitt.edu/sungkyu/course/2221Fall13/lec8_mds_combined.pdf`. To fit classical MDS, one can use cmdscale() To fit classical MDS with Sammon's stress: MASS::sammon()

Principal coordinate analysis (PCoA) It is special case of traditional MDS and a generalization of PCA. PCA applies eigendecomposition on the covariance matrix to identify the principal components. Whereas PCoA applies the eigendecomposition to any choice for measure of association. Simple Euclidean distances among the data points can also be transformed into a measure of association by taking $-D$ (some people also apply centering/double-centering). Then you can apply the eigendecomposition on $-D$ and follow the same set of steps as in PCA. The PCoA can be applied either to the variables or on the observations, as it is based on any notion of association unlike PCA which only makes sense to be applied to the observations.

1 Packages to use

(Please let me know if you get to know some other packages too.) If you are Python user, numpy and scikit have everything. `https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html` `https://scikit-learn.org/stable/modules/clustering.html#clustering` for general linear algebra and clustering. For dimension reduction, `https://scikit-learn.org/stable/modules/decomposition.html#decompositions`.

In R, base package itself has all the linear algebra routines. For the clustering part, there are different packages to run different things.

References

[1] James H Goodnight. A tutorial on the sweep operator. *The American Statistician*, 33(3):149–158, 1979.