Lecture 5

Scalable PCA

Dimensionality Reduction & Factor Analysis

Haiping Lu

https://github.com/haipinglu/ScalableML

COM6012 Scalable Machine Learning

Spring 2019

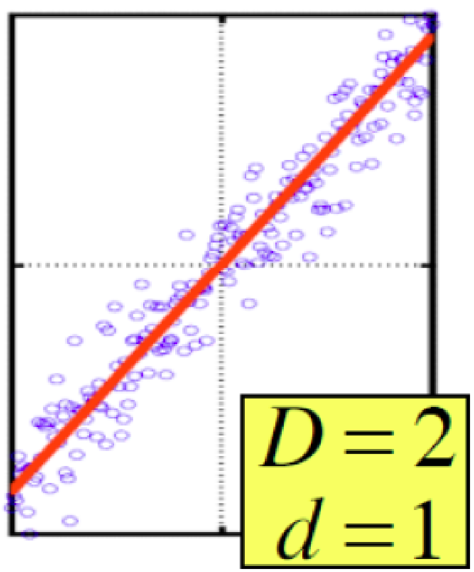
Week 5 Contents / Objectives

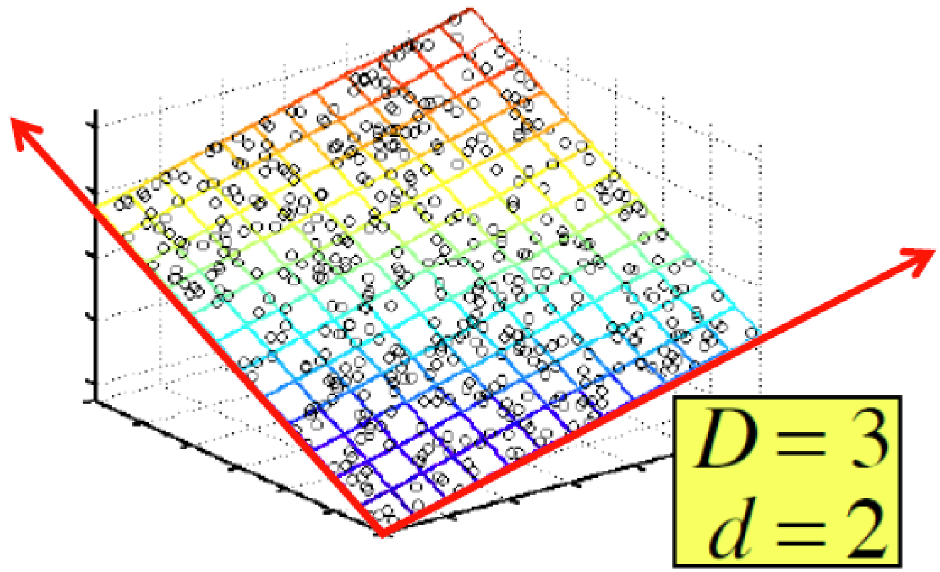
- PCA - Dimensionality Reduction
- SVD – Factor Analysis
- Scalable PCA in Spark
- More on Scala

Week 5 Contents / Objectives

- **PCA - Dimensionality Reduction**
- SVD – Factor Analysis
- Scalable PCA in Spark
- More on Scala

Dimensionality Reduction





- **Assumption:** Data lies on or near a low d-dimensional subspace
- Axes of this subspace are effective representation of the data

Why Reduce Dimensions?

Why reduce dimensions?

- Discover hidden correlations/topics
  - Words that occur commonly together
- Remove redundant and noisy features
  - Not all words are useful
- Interpretation and visualization
- Easier storage and processing of the data

Dimensionality Reduction

- Raw data is complex and high-dimensional
- Dimensionality reduction describes the data using a simpler, more compact representation
- This representation may make interesting patterns in the data clearer or easier to see

Dimensionality Reduction

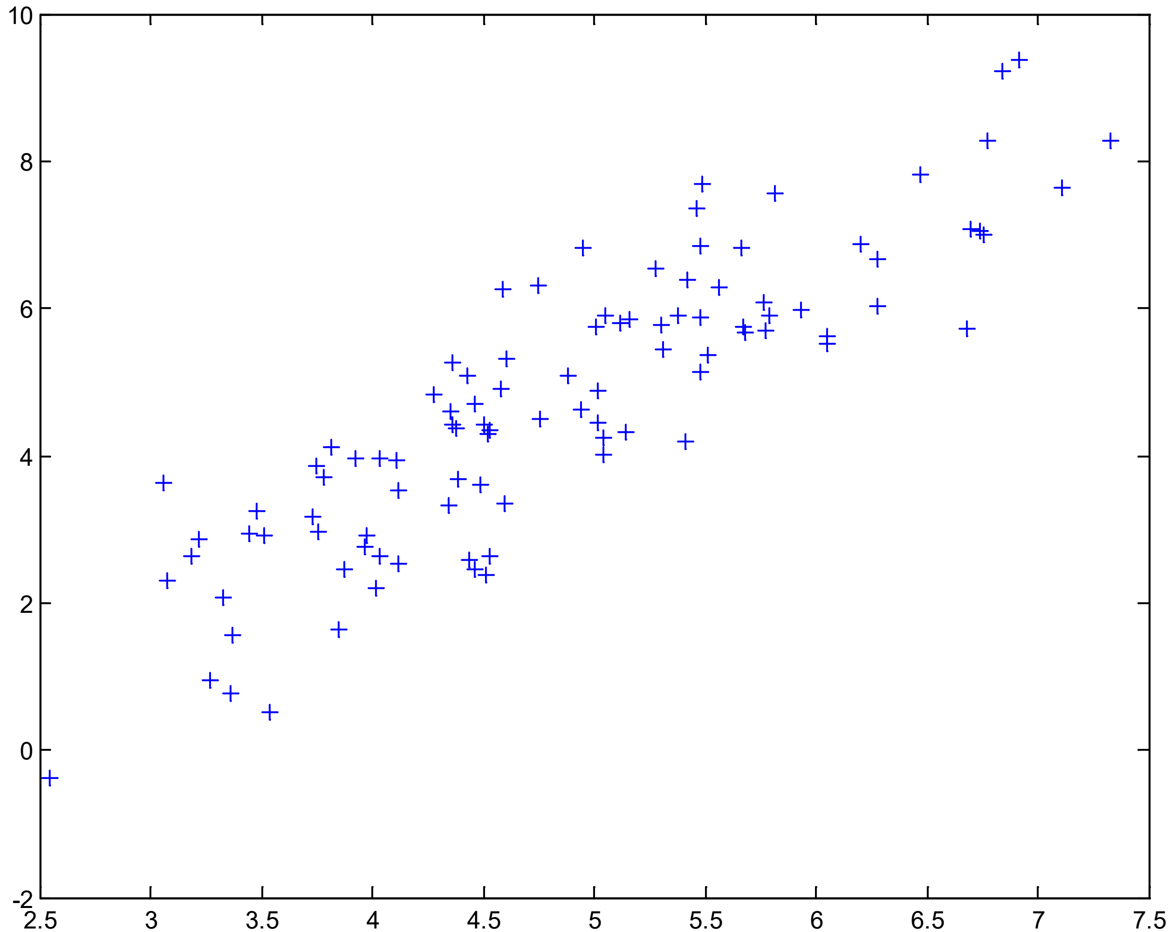
- Goal: Find a ‘better’ representation for data

- How do we define ‘better’?
- For example
  - Minimise reconstruction error
  - Maximise variance
  - **They give the same solution → PCA!**

PCA Algorithm

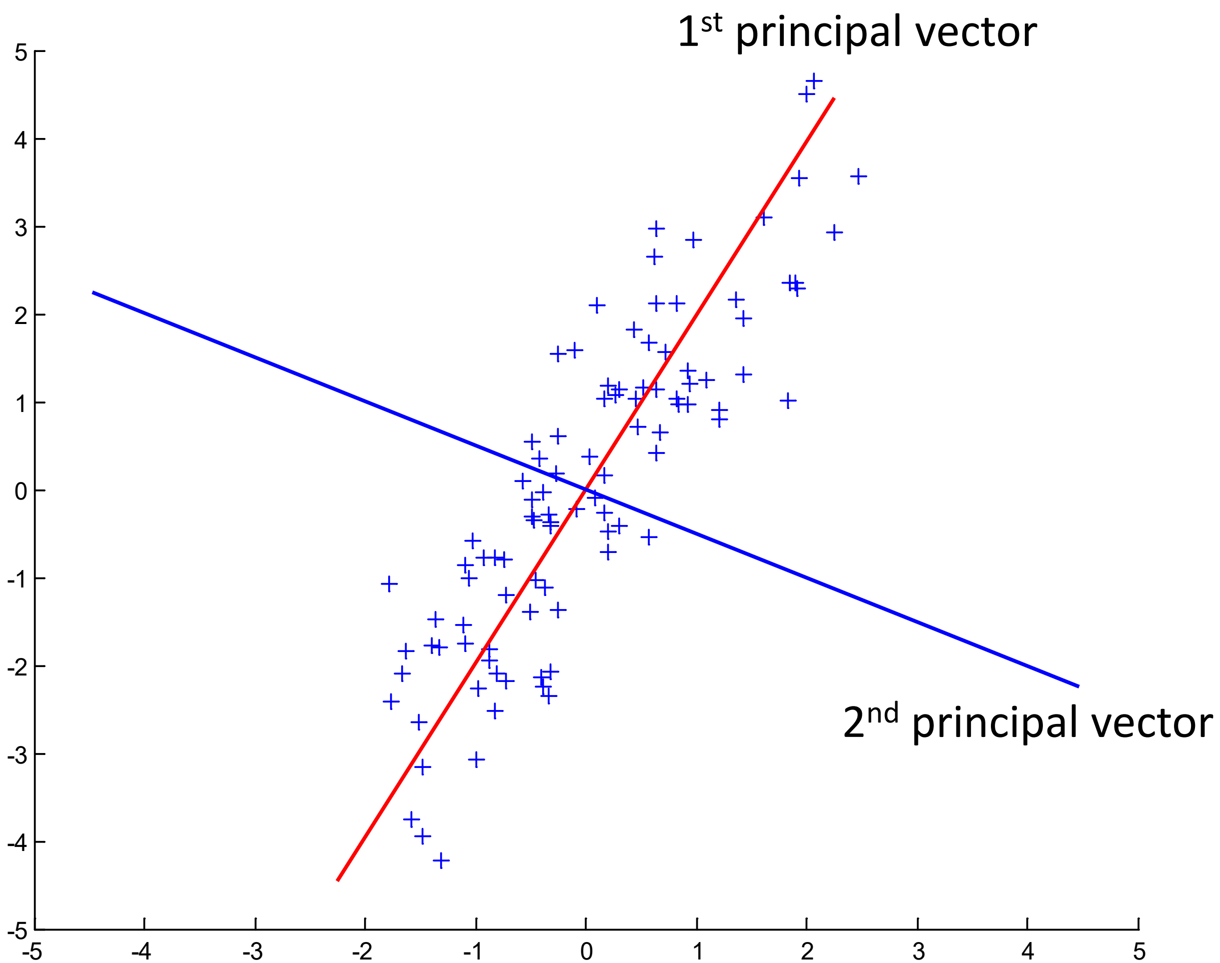
- Input: N data points, each → D-dimensional vector
- PCA algorithm
  - 1. $\mathbf{X_0}$ ← Form $N \times D$ data matrix, with one row vector $\mathbf{x}_n$ per data point
  - 2. $\mathbf{X}$: subtract mean $\mathbf{x}$ from each row vector $\mathbf{x}_n$ in $\mathbf{X_0}$
  - 3. $\mathbf{\Sigma} \leftarrow \mathbf{X}^T\mathbf{X}$ Gramian (scatter) matrix for $\mathbf{X}$
  - Find eigenvectors and eigenvalues of $\mathbf{\Sigma}$
  - PCs $\mathbf{U}$ ($D \times d$) ← the d eigenvectors with largest eigenvalues
- PCA feature for $\mathbf{y}$ D-dim: $\mathbf{U}^T\mathbf{y}$ (d-dimensional)
  - Zero correlations, ordered by variance

2D Data



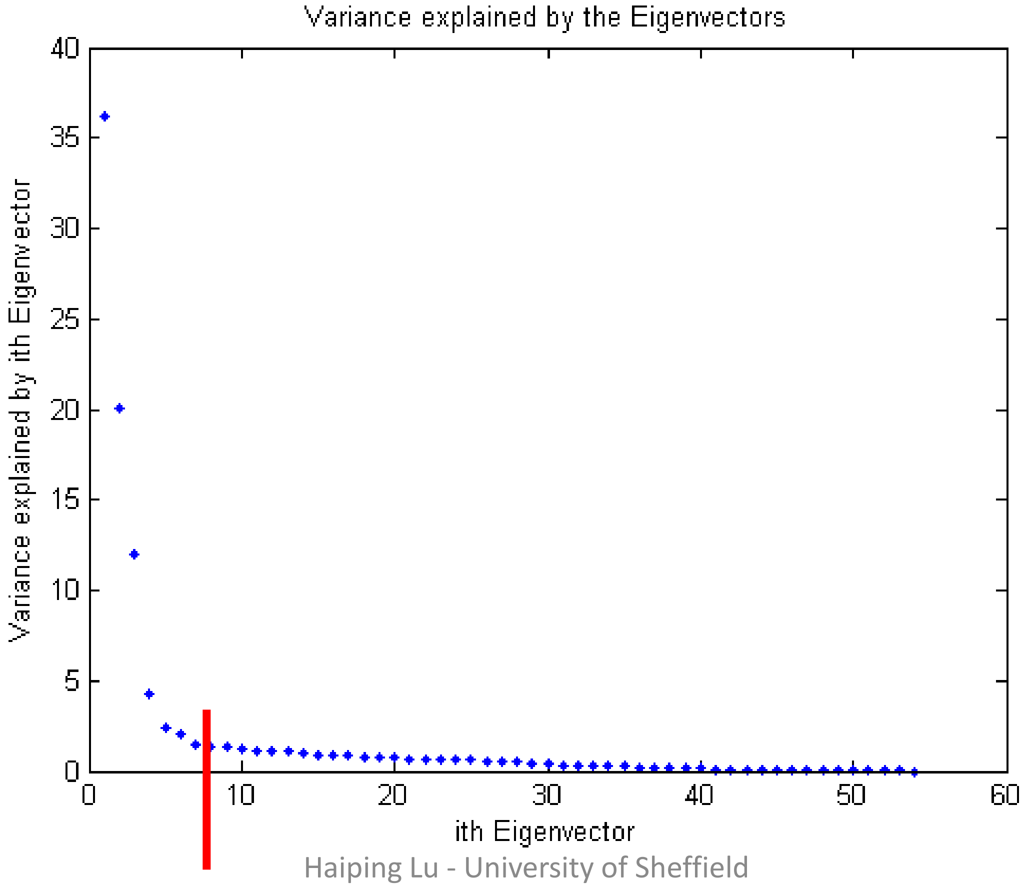
Principal Components

- The best axis to project
- Minimum RMS error
- Principal vectors are orthogonal



How Many Components?

- Check the distribution of eigen-values
- Take enough many eigen-vectors to cover 80-90% of the variance



Other Practical Tips

- PCA assumptions (linearity, orthogonality) not always appropriate
- Various extensions to PCA with different underlying assumptions, e.g., manifold learning, Kernel PCA, ICA
- Centring is crucial, i.e., we must preprocess data so that all features have zero mean before applying PCA
- PCA results dependent on scaling of data
- Data is sometimes rescaled in practice before applying PCA

Problems and Limitations

- What if very large dimensional data?
  - e.g., Images ($D \geq 10^4 = 100 \times 100$)
- Problem:
  - Gramian matrix Σ is size (D^2)
  - $D=10^4 \rightarrow |\Sigma| = 10^8$
- Singular Value Decomposition (SVD)!
  - Efficient algorithms available
  - Some implementations find just top *d* eigenvectors

Week 5 Contents / Objectives

- PCA - Dimensionality Reduction
- **SVD – Factor Analysis**
- Scalable PCA in Spark
- More on Scala

Singular Value Decomposition

- Factorization (decomposition) problem
  - #1: Find concepts/topics/genres → Factor Analysis
  - #2: Reduce dimensionality

term / document	data	information	retrieval	brain	lung
CS-TR1	1	1	1	0	0
CS-TR2	2	2	2	0	0
CS-TR3	1	1	1	0	0
CS-TR4	5	5	5	0	0
MED-TR1	0	0	0	2	2
MED-TR2	0	0	0	3	3
MED-TR3	0	0	0	1	1

The above matrix is actually "2-dimensional." All rows can be reconstructed by scaling [1 1 1 0 0] or [0 0 0 1 1]: D=5➔d=2

SVD - Definition

$$\mathbf{A}_{[\mathbf{n} \times \mathbf{m}]} = \mathbf{U}_{[\mathbf{n} \times \mathbf{r}]} \mathbf{\Lambda}_{[\mathbf{r} \times \mathbf{r}]} (\mathbf{V}_{[\mathbf{m} \times \mathbf{r}]})^{T}$$

- **A**: $n \times m$ matrix (e.g., n documents, m terms)
- **U**: $n \times r$ matrix (n documents, r concepts)
- **Λ**: $r \times r$ diagonal matrix (strength of each 'concept') (r: rank of the matrix)
- **V**: $m \times r$ matrix (m terms, r concepts)

SVD - Properties

Always possible to decompose matrix $\mathbf{A}$ into $\mathbf{A} = \mathbf{U}\,\mathbf{\Lambda}\,\mathbf{V}^T$, where

- **U, Λ, V**: unique (*)
- **U**, **V**: column orthonormal (i.e., columns are unit vectors, orthogonal to each other)
  - $\mathbf{U}^T\mathbf{U} = \mathbf{I}$; $\mathbf{V}^T\mathbf{V} = \mathbf{I}$ (**I:** identity matrix)
- **Λ:** singular value are positive, and sorted in decreasing order

SVD ←→Eigen-decomposition

- SVD gives us:
  - $\mathbf{A} = \mathbf{U}\ \mathbf{\Lambda}\ \mathbf{V}^T$
- Eigen-decomposition:
  - $\mathbf{B} = \mathbf{W}\ \mathbf{\Sigma}\ \mathbf{W}^T$
    - $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are orthonormal ($\mathbf{U}^T\mathbf{U}=\mathbf{I}$),
    - $\mathbf{\Lambda}, \mathbf{\Sigma}$ are diagonal
- Relationship:
  - $\mathbf{AA}^T = \mathbf{U}\ \Lambda\ \mathbf{V}^T(\mathbf{U}\ \Lambda\ \mathbf{V}^T)^T = \mathbf{U}\ \Lambda\ \mathbf{V}^T(\mathbf{V}\ \Lambda^T\mathbf{U}^T) = \mathbf{U}\ \Lambda\ \Lambda^T\ \mathbf{U}^T$
  - $\mathbf{A}^T\mathbf{A} = \mathbf{V}\ \Lambda^T\ \mathbf{U}^T\ (\mathbf{U}\ \Lambda\ \mathbf{V}^T) = \mathbf{V}\ \Lambda\ \Lambda^T\ \mathbf{V}^T = \mathbf{V}\ \Lambda^2\ \mathbf{V}^T$
  - $\mathbf{B} = \mathbf{A}^T\mathbf{A} = \mathbf{W}\ \Sigma\ \mathbf{W}^T$

SVD for PCA

- PCA by SVD:
  - 1. $\mathbf{X}_0 \leftarrow$ Form $N \times d$ data matrix, with one row vector $\mathbf{x}_n$ per data point
  - 2. $\mathbf{X}$ subtract mean $\mathbf{x}$ from each row vector $\mathbf{x}_n$ in $\mathbf{X}_0$
  - 3. $\mathbf{U}\ \mathbf{\Lambda}\ \mathbf{V}^T \leftarrow$ SVD of $\mathbf{X}$
  - The right singular vectors $\mathbf{V}$ of $\mathbf{X}$ are equivalent to the eigenvectors of $\mathbf{X}^T\mathbf{X} \rightarrow$ the PCs
  - The singular values in $\mathbf{\Lambda}$ are equal to the square roots of the eigenvalues of $\mathbf{X}^T\mathbf{X}$

SVD - Properties

'spectral decomposition' of the matrix:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} | & | \\ u_1 & u_2 \\ | & | \end{bmatrix} \times \begin{bmatrix} \lambda_1 & \varnothing \\ \varnothing & \lambda_2 \end{bmatrix} \times \begin{bmatrix} \text{---} & v_1 & \text{---} \\ \text{---} & v_2 & \text{---} \end{bmatrix}$$

SVD - Interpretation

‘documents’, ‘terms’ and ‘concepts’:

- **U**: document-to-concept similarity matrix
- **V**: term-to-concept similarity matrix
- **Λ**: its diagonal elements: ‘strength’ of each concept

Projection:

- Best axis to project on: (‘best’ = min sum of squares of projection errors)

SVD - Example

- $\mathbf{A} = \mathbf{U}\ \mathbf{\Lambda}\ \mathbf{V}^T$ - example:

term / document	data	information	retrieval	brain	lung
CS-TR1	1	1	1	0	0
CS-TR2	2	2	2	0	0
CS-TR3	1	1	1	0	0
CS-TR4	5	5	5	0	0
MED-TR1	0	0	0	2	2
MED-TR2	0	0	0	3	3
MED-TR3	0	0	0	1	1

$$
\begin{array}{cc}
 & \begin{array}{ccccc} \text{data} & \text{inf.} & \text{retrieval} & \text{brain} & \text{lung} \end{array} \\
\begin{array}{c} \\ \text{CS} \\ \\ \\ \\ \text{MD} \\ \\ \end{array} &
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 2 & 2 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
5 & 5 & 5 & 0 & 0 \\
0 & 0 & 0 & 2 & 2 \\
0 & 0 & 0 & 3 & 3 \\
0 & 0 & 0 & 1 & 1
\end{bmatrix}
\end{array}
=
\begin{bmatrix}
0.18 & 0 \\
0.36 & 0 \\
0.18 & 0 \\
0.90 & 0 \\
0 & 0.53 \\
0 & 0.80 \\
0 & 0.27
\end{bmatrix}
\text{x}
\begin{bmatrix}
9.64 & 0 \\
0 & 5.29
\end{bmatrix}
\text{x}
\begin{bmatrix}
0.58 & 0.58 & 0.58 & 0 & 0 \\
0 & 0 & 0 & 0.71 & 0.71
\end{bmatrix}
$$

SVD - Example

- $\mathbf{A} = \mathbf{U}\, \boldsymbol{\Lambda}\, \mathbf{V}^T$ - example:

doc-to-concept similarity matrix

CS-concept

MD-concept

$$
\begin{array}{c c}
 & \begin{array}{ccccc} \text{data} & \text{inf.} & \text{retrieval} & \text{brain} & \text{lung} \end{array} \\
\begin{array}{c} \text{CS} \\ \\ \\ \\ \text{MD} \\ \\ \\ \end{array} &
\left[\begin{array}{ccccc}
1 & 1 & 1 & 0 & 0 \\
2 & 2 & 2 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
5 & 5 & 5 & 0 & 0 \\
0 & 0 & 0 & 2 & 2 \\
0 & 0 & 0 & 3 & 3 \\
0 & 0 & 0 & 1 & 1
\end{array}\right]
\end{array}
=
\left[\begin{array}{cc}
0.18 & 0 \\
0.36 & 0 \\
0.18 & 0 \\
0.90 & 0 \\
0 & 0.53 \\
0 & 0.80 \\
0 & 0.27
\end{array}\right]
\times
\left[\begin{array}{cc}
9.64 & 0 \\
0 & 5.29
\end{array}\right]
\times
\left[\begin{array}{ccccc}
0.58 & 0.58 & 0.58 & 0 & 0 \\
0 & 0 & 0 & 0.71 & 0.71
\end{array}\right]
$$

SVD - Example

- $\mathbf{A} = \mathbf{U}\,\boldsymbol{\Lambda}\,\mathbf{V}^T$ - example:

'strength' of CS-concept

$$
\begin{array}{c|ccccc}
 & \text{data} & \text{inf.} & \text{retrieval} & \text{brain} & \text{lung} \\
\hline
\text{CS} & & & & & \\
\end{array}
$$

$$
\begin{array}{c}
\uparrow \\ \text{CS} \\ \downarrow \\ \uparrow \\ \text{MD} \\ \downarrow
\end{array}
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 2 & 2 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
5 & 5 & 5 & 0 & 0 \\
0 & 0 & 0 & 2 & 2 \\
0 & 0 & 0 & 3 & 3 \\
0 & 0 & 0 & 1 & 1
\end{bmatrix}
=
\begin{bmatrix}
0.18 & 0 \\
0.36 & 0 \\
0.18 & 0 \\
0.90 & 0 \\
0 & 0.53 \\
0 & 0.80 \\
0 & 0.27
\end{bmatrix}
\times
\begin{bmatrix}
9.64 & 0 \\
0 & 5.29
\end{bmatrix}
\times
\begin{bmatrix}
0.58 & 0.58 & 0.58 & 0 & 0 \\
0 & 0 & 0 & 0.71 & 0.71
\end{bmatrix}
$$

SVD - Example

- $\mathbf{A} = \mathbf{U}\,\mathbf{\Lambda}\,\mathbf{V}^T$ - example:

term-to-concept similarity matrix

CS-concept

$$
\begin{array}{cc}
 & \begin{array}{ccccc} \text{data} & \text{inf.} & \text{retrieval} & \text{brain} & \text{lung} \end{array} \\
\begin{array}{c} \text{CS} \\ \\ \\ \\ \text{MD} \\ \\ \\ \end{array} &
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 2 & 2 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
5 & 5 & 5 & 0 & 0 \\
0 & 0 & 0 & 2 & 2 \\
0 & 0 & 0 & 3 & 3 \\
0 & 0 & 0 & 1 & 1
\end{bmatrix}
\end{array}
=
\begin{bmatrix}
0.18 & 0 \\
0.36 & 0 \\
0.18 & 0 \\
0.90 & 0 \\
0 & 0.53 \\
0 & 0.80 \\
0 & 0.27
\end{bmatrix}
\times
\begin{bmatrix}
9.64 & 0 \\
0 & 5.29
\end{bmatrix}
\times
\begin{bmatrix}
0.58 & 0.58 & 0.58 & 0 & 0 \\
0 & 0 & 0 & 0.71 & 0.71
\end{bmatrix}
$$

SVD – Dimensionality Reduction

- Q: how exactly is (**further**) dim. reduction done?
- A: set the smallest singular values to zero:
- Note: **3 zero singular values** already removed

$$
\begin{bmatrix}
1 & 1 & 1 & 0 & 0 \\
2 & 2 & 2 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 \\
5 & 5 & 5 & 0 & 0 \\
0 & 0 & 0 & 2 & 2 \\
0 & 0 & 0 & 3 & 3 \\
0 & 0 & 0 & 1 & 1
\end{bmatrix}
=
\begin{bmatrix}
0.18 & \cancel{0} \\
0.36 & \cancel{0} \\
0.18 & \cancel{0} \\
0.90 & \cancel{0} \\
0 & \cancel{0.53} \\
0 & \cancel{0.80} \\
0 & \cancel{0.27}
\end{bmatrix}
\text{ x }
\begin{bmatrix}
9.64 & 0 \\
0 & \cancel{5.29}
\end{bmatrix}
\text{ x }
\begin{bmatrix}
0.58 & 0.58 & 0.58 & 0 & 0 \\
\cancel{0} & \cancel{0} & \cancel{0} & \cancel{0.71} & \cancel{0.71}
\end{bmatrix}
$$

SVD - Dimensionality Reduction

$$
\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \sim \begin{bmatrix} 0.18 \\ 0.36 \\ 0.18 \\ 0.90 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ x } \begin{bmatrix} 9.64 \end{bmatrix} \text{ x } \begin{bmatrix} 0.58 & 0.58 & 0.58 & 0 & 0 \end{bmatrix}
$$

SVD - Dimensionality Reduction

- Best rank-1 approximation

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 \\ 0 & 0 & 0 & 3 & 3 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \sim \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 5 & 5 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Week 5 Contents / Objectives

- PCA - Dimensionality Reduction
- SVD – Factor Analysis
- **Scalable PCA in Spark**
- More on Scala

PCA & SVD in Spark MLlib

- Not scalable: computePrincipalComponents( ) from RowMatrix
- **Scalable**: computeSVD( ) from RowMatrix
- Code: https://github.com/apache/spark/blob/v2.3.2/mllib/src/main/scala/org/apache/spark/mllib/linalg/distributed/RowMatrix.scala
- Documentation: https://spark.apache.org/docs/2.3.2/api/scala/index.html#org.apache.spark.mllib.linalg.distributed.RowMatrix

PCA in Spark MLlib (RDD)

- https://spark.apache.org/docs/2.3.2/mllib-dimensionality-reduction.html

```
val mat: RowMatrix = new RowMatrix(dataRDD)

// Compute the top 4 principal components.
// Principal components are stored in a local dense matrix.
val pc: Matrix = mat.computePrincipalComponents(4)
```

- Not scalable, local computation

```
val brzSvd.SVD(u: BDM[Double], s: BDV[Double], _) = brzSvd(Cov)
```

PCA in Spark ML (DF)

- Now in
  https://spark.apache.org/docs/2.3.2/ml-features.html#pca
- Under features
- Not scalable

```
val pca = new PCA()
  .setInputCol("features")
  .setOutputCol("pcaFeatures")
  .setK(3)
  .fit(df)
```

SVD in Spark MLlib (RDD)

- https://spark.apache.org/docs/2.3.2/mllib-dimensionality-reduction.html
- With distributed implementations

```
val mat: RowMatrix = new RowMatrix(dataRDD)

// Compute the top 5 singular values and corresponding singular vectors.
val svd: SingularValueDecomposition[RowMatrix, Matrix] = mat.computeSVD(5, computeU = true)
val U: RowMatrix = svd.U  // The U factor is a RowMatrix.
val s: Vector = svd.s  // The singular values are stored in a local dense vector.
val V: Matrix = svd.V  // The V factor is a local dense matrix.
```

SVD in Spark MLlib (RDD)

- An $m \times n$ data matrix **A** with $m>n$ (note different notations)
- For large matrices, usually we don't need the complete factorization but only the top k singular values and its associated singular vectors.
- Save storage, de-noise and recover the low-rank structure of the matrix (dimensionality reduction)

SVD in Spark MLlib (RDD)

- An $m \times n$ data matrix $\mathbf{A}$
- Assume $m>n$, SVD $\mathbf{A} = \mathbf{U}\, \boldsymbol{\Lambda}\, \mathbf{V}^{\mathrm{T}}$
- The singular values and the right singular vectors are derived from the eigenvalues and the eigenvectors of $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ (which is smaller than $\mathbf{A}$)
- The left singular vectors are computed via matrix multiplication as $\mathbf{U}=\mathbf{A}\mathbf{V}\, \boldsymbol{\Lambda}^{-1}$, if requested by the user via the computeU parameter

Selection of SVD Computation

- Auto
- If n is small (n<100) or k is large compared with n ($k>n/2$)
  - Compute $\mathbf{A}^T\mathbf{A}$ first and then compute its top eigenvalues and eigenvectors **locally** on the driver
- Otherwise
  - Compute $\mathbf{A}^T\mathbf{A}\,\mathbf{v}$ in a distributive way and send it to ARPACK to compute the top eigenvalues and eigenvectors on the driver node

Selection of SVD Computation

- Auto (default)

```
if (n < 100 || (k > n / 2 && n <= 15000)) {
  // If n is small or k is large compared with n, we better compute the Gramian matrix first
  // and then compute its eigenvalues locally, instead of making multiple passes.
  if (k < n / 3) {
    SVDMode.LocalARPACK
  } else {
    SVDMode.LocalLAPACK
  }
} else {
  // If k is small compared with n, we use ARPACK with distributed multiplication.
  SVDMode.DistARPACK
}
```

Selection of SVD Computation

- Specify computeMode (private)

```
case "local-svd" => SVDMode.LocalLAPACK
case "local-eigs" => SVDMode.LocalARPACK
case "dist-eigs" => SVDMode.DistARPACK
```

Selection of SVD Computation

- computeMode (note brzSvd.SVD is local)

```
// Compute the eigen-decomposition of A' * A.
val (sigmaSquares: BDV[Double], u: BDM[Double]) = computeMode match {
  case SVDMode.LocalARPACK =>
    require(k < n, s"k must be smaller than n in local-eigs mode but got k=$k and n=$n.")
    val G = computeGramianMatrix().asBreeze.asInstanceOf[BDM[Double]]
    EigenValueDecomposition.symmetricEigs(v => G * v, n, k, tol, maxIter)
  case SVDMode.LocalLAPACK =>
    // breeze (v0.10) svd latent constraint, 7 * n * n + 4 * n < Int.MaxValue
    require(n < 17515, s"$n exceeds the breeze svd capability")
    val G = computeGramianMatrix().asBreeze.asInstanceOf[BDM[Double]]
    val brzSvd.SVD(uFull: BDM[Double], sigmaSquaresFull: BDV[Double], _) = brzSvd(G)
    (sigmaSquaresFull, uFull)
  case SVDMode.DistARPACK =>
    if (rows.getStorageLevel == StorageLevel.NONE) {
      logWarning("The input data is not directly cached, which may hurt performance if its"
        + " parent RDDs are also uncached.")
    }
    require(k < n, s"k must be smaller than n in dist-eigs mode but got k=$k and n=$n.")
    EigenValueDecomposition.symmetricEigs(multiplyGramianMatrixBy, n, k, tol, maxIter)
}
```

Week 5 Contents / Objectives

- PCA - Dimensionality Reduction
- SVD – Factor Analysis
- Scalable PCA in Spark
- **More on Scala**

Scala (Scalable language)

- A pure object-oriented language. Conceptually, every value is an object and every operation is a method-call.
- A functional language. Supports functions, immutable data structures and preference for immutability over mutation
- Seamlessly integrated with Java
  - Mixed Scala/Java projects
  - Use existing Java libraries
  - Use existing Java tools

Scala Basic Syntax

- When considering a Scala program, it can be defined as a collection of objects that communicate via invoking each other’s methods.
- **Object** − same as in Java
- **Class** − same as in Java
- **Methods** − same as in Java
- **Fields** − Each object has its unique set of instant variables, which are called fields. An object's state is created by the values assigned to these fields.
- **Traits** − Like Java Interface. A trait encapsulates method and field definitions, which can then be reused by mixing them into classes.
- **Closure** − A **closure** is a function, whose return value depends on the value of one or more variables declared outside this function.

closure = function + environment

Scala is Statically Typed

- You don't have to specify a type in most cases
- Type Inference

```scala
val sum = 1 + 2 + 3
val nums = List(1, 2, 3)
val map = Map("abc" -> List(1,2,3))
```

Explicit Types

```scala
val sum: Int = 1 + 2 + 3
val nums: List[Int] = List(1, 2, 3)
val map: Map[String, List[Int]] = ...
```

Scala is High level

```
// Java – Check if string has uppercase character
boolean hasUpperCase = false;
for(int i = 0; i < name.length(); i++) {
    if(Character.isUpperCase(name.charAt(i))) {
        hasUpperCase = true;
       break;
    }
}

// Scala
val hasUpperCase = name.exists(_.isUpperCase)
```

Scala is Concise

// Java

```java
public class Person {
  private String name;
  private int age;
  public Person(String name, Int age) {
    this.name = name;
   this.age = age;
  }
  public String getName() {               // name getter
    return name;
  }
  public int getAge() {                   // age getter
    return age;
  }
  public void setName(String name) {      // name setter
    this.name = name;
  }
  public void setAge(int age) {           // age setter
    this.age = age;
  }
}
```

```scala
// Scala
class Person(var name: String, private var _age: Int) {
  def age = _age             // Getter for age
  def age_=(newAge:Int) {  // Setter for age
    println("Changing age to: "+newAge)
    _age = newAge
  }
}
```

Variables and Values

- Variables: values stored can be changed

```
var foo = "foo"
foo = "bar"  // okay
```

- Values: immutable variable

```
val foo = "foo"
foo = "bar" // nope
```

Scala is Functional

- First Class Functions. Functions are treated like objects:
  - passing functions as arguments to other functions
  - returning functions as the values from other functions
  - assigning functions to variables or storing them in data structures

```
// Lightweight anonymous functions
(x:Int) => x + 1

// Calling the anonymous function
val plusOne = (x:Int) => x + 1
plusOne(5)  → 6
```

Scala is Functional

- Closures: a function whose return value depends on the value of one or more variables declared outside this function.

```
// plusFoo can reference any values/variables in scope
var foo = 1
val plusFoo = (x:Int) => x + foo

plusFoo(5)   → 6

// Changing foo changes the return value of plusFoo
foo = 5
plusFoo(5)   → 10
```

Scala is Functional

- Higher Order Functions
  - A function that does at least one of the following:
    - takes one or more functions as arguments
    - returns a function as its result

val plusOne = (x:Int) => x + 1

val nums = List(1,2,3)

```
// map takes a function: Int => T
nums.map(plusOne)        →  List(2,3,4)
// Inline Anonymous
nums.map(x => x + 1)     →  List(2,3,4)
// Short form
nums.map(_ + 1)          →  List(2,3,4)
```

More Examples on Higher Order Functions

```
val nums = List(1,2,3,4)
// A few more examples for List class
nums.exists(_ == 2)             →  true
nums.find(_ == 2)               →  Some(2)
nums.indexWhere(_ == 2)         →  1
// functions as parameters, apply f to the
  value “1”
def call(f: Int => Int) = f(1)
call(plusOne)        →  2
call(x => x + 1)     →  2
call(_ + 1)          →  2
```

The Usage of "_" in Scala

- In anonymous functions, the "_" acts as a placeholder for parameters

  nums.map(x => x + 1) is equivalent to:

  nums.map(_ + 1)

  List(1,2,3,4,5).foreach(print(_)) is equivalent to:

  List(1,2,3,4,5).foreach(a => print(a))

- You can use two or more underscores to refer different parameters.

  val sum = List(1,2,3,4,5).reduceLeft(_+_) is equivalent to:

  val sum = List(1,2,3,4,5).reduceLeft((a, b) => a + b)

  - The reduceLeft method works by applying the function/operation you give it, and applying it to successive elements in the collection

Acknowledgement & References

- Acknowledgement
  - Some slides are adapted from slides by Jure Leskovec et al. http://www.mmds.org
- References
  - http://infolab.stanford.edu/~ullman/mmds/ch11.pdf
  - http://www.mmds.org

Be Scalable

for both computing & living