com.aliasi.matrix

Class SvdMatrix

```
java.lang.Object
  └com.aliasi.matrix.AbstractMatrix
      └com.aliasi.matrix.SvdMatrix
```

All Implemented Interfaces:
Matrix

```
public class SvdMatrix
extends AbstractMatrix
```

An `SvdMatrix` provides a means of storing a matrix that has been factored via a singular-value decomposition (SVD). This class also provides a static method for computing regularized singular-value decompositions of partial matrices.

Singular Value Decomposition

Singular value decomposition (SVD) factors an `m×n` matrix `A` into a product of three matrices:

$$A = U * S * V^T$$

where `U` is an `m×k` matrix, `V` is an `n×k` matrix, and `S` is a `k×k` matrix, where `k` is the rank of the matrix `A`. The multiplication (`*`) is matrix multiplication and the superscripted `T` indicates matrix transposition.

The `m`-dimensional vectors making up the columns of `U` are called left singular vectors, whereas the `n`-dimesnional vectors making up the rows of `V` are called right singular vectors. The values on the diagonal of `S` are called the singular values. The combination of the `q`-th left singular vector, right singular vector, and singular value is known as a factor.

The singular value matrix `S` is a diagonal matrix with positive, strictly non-increasing values, so that `S[i][i] >= S[i+1][i+1]`, for `i < k`. The set of left and set of right singular vectors are orthonormal. Normality means that each singular vector is of unit length (length `1`). Orthogonality means that any pair of left singular vectors is orthogonal and any pair of right singular vectors are orthogonal (meaning their dot product is `0`).

Matrices have unique singular-value decompositions up to the re-ordering of columns with equal singular values and up to cancelling sign changes in the singular vectors.

Construction and Value Computation

An `SvdMatrix` may be constructed out of the singular vectors and singular values, or out of the vectors with singular values scaled in.

Given that `S` is diagonal, the value of a particular entry in `A` works out to:

$$A[i][j] = \Sigma_k U[i][k] * S[k][k] * V[j][k]$$

To save time in the application and space in the class, we factor `S` into `U` and `V` to produce a new pair of matrices `U'` and `V'` defined by:

$$U' = U * sqrt(S)$$
$$V'^T = sqrt(S) * V^T$$

with the square-root performed component-wise:

$$sqrt(S)[i][j] = sqrt(S[i][j])$$

By the associativity of matrix multiplication, we have:

$$U * S * V^T$$
$$= U * sqrt(S) * sqrt(S) * V$$
$$= U' * V'^T$$

Thus the class implementation is able to store `U'` and `V'`, thus reducing the amount computation involved in returning a value (using column vectors as the default vector orientation):

$$A[i][j] = U'[i]^T * V'[j]$$

Square Error and the Frobenius Norm

Suppose `A` and `B` are `m×n` matrices. The square error between them is defined by the so-called Frobenius norm, which extends the standard vector L_2 or Euclidean norm to matrices:

$$squareError(A,B)$$

$$= frobeniusNorm(A-B)$$

$$= \Sigma_{i < m} \Sigma_{j < n} (A[i][j] - B[i][j])^2$$

The square error is sometimes referred to as the residual sum of squares (RSS), because `A[i][j] - B[i][j]` is the residual (difference between actual and predicted value).

Lower-order Approximations

Consider factoring a matrix `A` of dimension `m×n` into the product of two matrices $X * Y^T$, where `X` is of dimension `m×k` and `Y` is of dimension `n×k`. We may then measure the square error `squareError(A,X * Y)` to determine how well the factorization matches `A`. We know that if we take `U'`, `V'` from the singular value decomposition that:

$$\texttt{squareError(A, U' * V'}^{T}\texttt{) = 0.0}$$

Here `U'` and `V'` have a number of columns (called factors) equal to the rank of the matrix `A`. The singular value decomposition is such that the first `k` columns of `U'` and `V'` provide the best order `q` approximation of `A` of any `X` and `Y` of dimensions `m×q` and `n×q` In symbols:

$$U'_q, V'_q = \mathrm{argmin}_{X \text{ is } m\times q,\ Y \text{ is } n\times q}\ \texttt{squareError}(A, X * Y^T)$$

where U'_q is the restriction of `U'` to its first `q` columns.

Often errors are reported as means, where the mean square error (MSE) is defined by:

```
meanSquareError(A,B) = squareError(A,B)/(m×n)
```

To adjust to a linear scale, the square root of mean square error (RMSE) is often used:

```
rootMeanSquareError(A,B) = sqrt(meanSquareError(A,B))
```

Partial Matrices

A partial matrix is one in which some of the values are unknown. This is in contrast with a sparse matrix, in which most of the values are zero. A variant of singular value decomposition may be used to impute the unknown values by minimizing square error with respect to the known values only. Unknown values are then simply derived from the factorization $U' * V'^T$. Typically, the approximation is of lower order than the rank of the matrix.

Regularized SVD via Shrinkage

Linear regression techniques such as SVD often overfit their training data in order to derive exact answers. This problem is mitigated somewhat by choosing low-order approximations to the full-rank SVD. Another option is to penalize large values in the singular vectors, thus favoring smaller values. The most common way to do this because of its practicality is via parameter shrinkage.

Shrinkage is a general technique in least squares fitting that adds a penalty term proportional to the square of the size of the parameters. Thus the square error objective function is replaced with a regularized version:

```
regularizedError(A, U' * V')

= squareError(A, U' * V')

+ parameterCost(U') + parameterCost(V')
```

where the parameter costs for a vector `X` of dimensionality `q` is the sum of the squared parameters:

> parameterCost(X)
>
> $= \lambda * \Sigma_{i < q} X[i]^2$
>
> $= \lambda * \text{length}(X)^2$

Note that the hyperparameter λ scales the parameter cost term in the overall error.

Setting the regularization parameter to zero sets the parameter costs to zero, resulting in an unregularized SVD computation.

In Bayesian terms, regularization is equivalent to a normal prior on parameter values centered on zero with variance controlled by λ. The resulting minimizer of regularized error is the maximum a posteriori (MAP) solution. The Bayesian approach also allows covariances to be estimated (including simple parameter variance estimates), but these are not implemented in this class.

Regularized Stochastic Gradient Descent

Singular value decomposition may be computed "exactly" (modulo arithmetic precision and convergence) using an algorithm whose time complexity is $O(m^3 + n^3)$ for an $m \times n$ matrix (equal to $O(\max(m,n)^3)$). Arithmetic precision is especially vexing at singular values near zero and with highly correlated rows or columns in the input matrix.

For large partial matrices, we use a form of stochastic gradient descent which computes a single row and column singular vector (a single factor, that is) at a time. Each factor is estimated by iteratively visiting the data points and adjusting the unnormalized singular vectors making up the current factor. Each adjustment is a least-squares fitting step, where we simultaneously adjust the left singular vectors given the right singular vectors and vice-versa.

The least-squares adjustments take the following form. For each value, we compute the current error (using the order `k` approximation and the current order `k+1` values) and then move the vectors to reduce error. We use `U'` and `V'` for the incremental values of the singular vectors scaled by the singular values:

```
for (k = 0; k < maxOrder; ++k)
    for (i < m) U'[i][k] = random.nextGaussian()*initValue;
    for (j < n) V'[j][k] = random.nextGaussian()*initValue;
    for (epoch = 0; epoch < maxEpochs && not converged; ++epoch)
        for (i,j such that M[i][j] defined)
            error = M[i][j] - U'_k[i] * V'_k[j] // * is vector dot product
            uTemp = U'[i][k]
            vTemp = V'[j][k]
            U'[i][k] += learningRate[epoch] * ( error * vTemp - regularization * uTemp )
            V'[j][k] += learningRate[epoch] * ( error * uTemp - regularization * vTemp )
```

where `initValue`, `maxEpochs`, `learningRate` (see below), and `regularization` are algorithm hyperparameters. Note that the initial values of the singular vectors are set randomly to the result of a draw from a Gaussian (normal) distribution with mean 0.0 and variance 1.0.

Because we use the entire set of factors in the error computation, the current factor is guaranteed to have singular vectors orthogonal to the singular vectors already computed.

Note that in the actual implementation, the contribution to the error of the first `k-1` factors is cached to reduce time in the inner loop. This requires a double-length floating point value for each defined entry in the matrix.

Gradient Interpretation

Like most linear learners, this algorithm merely moves the parameter vectors `U'[i]` and `U'[j]` in the direction of the gradient of the error. The gradient is the multivariate derivative of the objective function being minimized. Because our object is squared error, the gradient is just its derivative, which is just (two times) the (linear) error itself. We roll the factor of 2 into the learning rate to derive the update in the algorithm pseudo-code above.

The term `(error * vTemp)` is the component of the error gradient due to the current value of `V'[i][k]` and the term `(regularization * uTemp)` is the component of the gradient to the size of the parameter `U'[i][k]`. The updates thus move the parameter vectors in the direction of the gradient.

Convergence Conditions

The convergence conditions for a given factor require either hitting the maximum number of allowable epochs, or finding the improvement from one epoch to the next is below some relative threshold:

$$\text{regError}^{(\text{epoch})} = \text{regError}(M, U'_k{}^{(\text{epoch})} * V'_k{}^{(\text{epoch})T})$$

$$\text{relativeImprovement}^{(\text{epoch}+1)} = \text{relativeDiff}(\text{regError}^{(\text{epoch}+1)}, \text{regError}^{(\text{epoch})})$$

```
relativeDiff(x,y) = abs(x-y)/(abs(x) + abs(y))
```

When the relative difference in square error is less than a hyperparameter threshold `minImprovement`, the epoch terminates and the algorithm moves on to the next factor `k+1`.

Note that a complete matrix is a degenerate kind of partial matrix. The gradient descent computation still works in this situation, but is not as efficient or as accurate as an algebraic SVD solver for small matrices.

Annealing Schedule

Learning rates that are too high are unstable, whereas learning rates that are too low never reach their targets. To get around this problem, the learning rate, `learningRate[epoch]`, is lowered as the number of epochs increase. This lowering of the learning rate has a thermodynamic interpretation in terms of free energy, hence the name "annealing". Larger moves are made in earlier epochs, then the temperature is gradually lowered so that the learner may settle into a stable fixed point. The function `learningRate[epoch]` is called the annealing schedule.

There are theoretical requirements on the annealing schedule that guarantee convergence (up to arithmetic precision and no upper bound on the number of epochs):

$$\Sigma_{epoch} \text{ learningRate[epoch] = infinity}$$

$$\Sigma_{epoch} \text{ learningRate[epoch]}^2 < \text{infinity}$$

The schedule we use is the one commonly chosen to meet the above requirements:

```
learningRate[epoch] = initialLearningRate / (1 + epoch/annealingRate)
```

where `initialLearningRate` is an initial learning rate and `annealingFactor` determines how quickly it shrinks. The learning rate moves from its initial size (`initialLearningRate`) to one half (`1/2`) of its original size after `annealingRate` epochs, and moves from its initial size to one tenth (`1/10`) of its initial size after `9 * annealingRate` epochs, and one hundredth of its initial size after `99 * annealingRate` epochs..

Parameter Choices

The previous discussion has introduced a daunting list of parameters required for gradient descent for singular value decomposition. Unfortunately, the stochastic gradient descent solver requires a fair amount of tuning to recover a low mean square error factorization. The optimal settings will also depend on the input matrix; for example, very sparse partial matrices are much easier to fit than dense matrices.

Maximum Order

Determining the order of the decomposition is mostly a matter of determining how many orders are needed for the amount of smoothing required. Low order reconstructions are useful for most applications. One way to determine maximum order is using held out data for an application. Another is to look for a point where the singular values become (relatively) insignificant.

Maximum Epochs and Early Stopping

For low mean square error factorizations, many epochs may be necessary. Lowering the maximum number of epochs leads to what is known as early stopping. Sometimes, early stopping leads to more accurate held-out predictions, but it will always raise the factorization error (training data predictions). In general, the maximum number of epochs needs to be set empirically. Initially, try fewer epochs and gradually raise the number of epochs until the desired accuracy is achieved.

Minimum Improvement

By setting the minimum improvement to 0.0, the algorithm is forced to use the maximum number of epochs. By setting it above this level, a form of early stopping is achieved when the benefit of another epoch of refitting falls below a given improvement threshold. This value may also be set on an application basis using held out data, or it may be fit to achieve a given level of mean square error on the training (input) data. The minimum improvement needs to be set fairly low (less than 0.000001) to achieve reasonably precise factorizations. Note that minimum improvement is defined relatively, as noted above.

Initial Parameter Values

Initial values of the singular vectors are not particularly sensitive, because we are using multiplicative updates. A good rule of thumb for starting values is the the inverse square root of the maximum order:

```
      initVal ~ 1 / sqrt(maxOrder)
```

Learning Rate, Maximum Epochs and Annealing

A good starting point for the learning rate is 0.01. The annealing parameter should be set so that the learning rate is cut by at least a factor of 10 in the final rounds. This calls for a value that's roughly one tenth of the maximum number of epochs. If the initial learning rate is set higher, then the annealing schedule should be more agressive so that it spends a fair amount of time in the 0.001 to 0.0001 learning rate range.

References

There are thorough Wikipedia articles on singular value decomposition and gradient descent, although the SVD entry focuses entirely on complete (non-partial) matrices:

- Wikipedia: Singular Value Decomposition
- Wikipedia: Gradient Descent

Both of the standard machine learning textbooks have good theoretical explanations of least squares, regularization, gradient descent, and singular value decomposition, but not all together:

- Chris Bishop. 2007. *Pattern Recognition and Machine Learning.* Springer.
- Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2001. *The Elements of Statistical Learning*. Springer.

The following is a particularly clear explanation of many of the issues involved in the context of neural networks:

- Genevieve Orr, Nici Schraudolph, and Fred Cummins. 1999. CS-449: Neural Networks. Willamette University course notes.

Our partial SVD solver is based on C code from Timely Development (see license below). Timely based their code on Simon Funk's blog entry. Simon Funk's approach was itself based on his and Genevieve Gorrell's 2005 EACL paper.

- Timely Development's Netflix Prize Page.
- Simon Funk (pseudonym for Brandyn Webb). 2007. Gradient Descent SVD Algorithm. *The Evolution of Cybernetics* blog.
- Genevieve Gorrell. 2006. Generalized Hebbian Algorithm for Incremental Singular Value Decomposition in Natural Language Processing. EACL 2006.
- Genevieve Gorrell. 2006. Generalized Hebbian Algorithm for Dimensionality Reduction in Natural Language Processing. Ph.D. Thesis. Linköping University. Sweden.

Acknowledgements

The singular value decomposition code is rather loosely based on a C program developed by Timely Development. Here is the copyright notice for the original code:

```
// SVD Sample Code
//
// Copyright (C) 2007 Timely Development (www.timelydevelopment.com)
//
// Special thanks to Simon Funk and others from the Netflix Prize contest
// for providing pseudo-code and tuning hints.
//
// Feel free to use this code as you wish as long as you include
// these notices and attribution.
//
// Also, if you have alternative types of algorithms for accomplishing
// the same goal and would like to contribute, please share them as well :)
//
// STANDARD DISCLAIMER:
//
// - THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY
// - OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT
// - LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR
// - FITNESS FOR A PARTICULAR PURPOSE.
```

Since:
LingPipe3.2

Version:
4.0.0

Author:
Bob Carpenter

Constructor Summary

SvdMatrix(double[][] rowSingularVectors, double[][] columnSingularVectors, double[] singularValues) Construct an SVD matrix using the specified singular vectors and singular values.
SvdMatrix(double[][] rowVectors, double[][] columnVectors, int order) Construct a factored matrix using the specified row and column vectors at the specified order.

Method Summary

`double[][]`	`leftSingularVectors()` Returns a matrix in which the left singular vectors make up the columns.
`int`	`numColumns()` Returns the number of columns in this matrix.
`int`	`numRows()` Returns the number of rows in this matrix.
`int`	`order()` Returns the order of this factorization.
`static SvdMatrix`	`partialSvd(int[][] columnIds, double[][] values, int maxOrder, double featureInit, double initialLearningRate, double annealingRate, double regularization, Reporter reporter, double minImprovement, int minEpochs, int maxEpochs)` Return the singular value decomposition of the specified partial matrix, using the specified search parameters.
`double[][]`	`rightSingularVectors()` Returns a matrix in which the right singular vectors make up the columns.
`double`	`singularValue(int order)` Returns the singular value for the specified order.
`double[]`	`singularValues()` Returns the array of singular values for this decomposition.
`static SvdMatrix`	`svd(double[][] values, int maxOrder, double featureInit, double initialLearningRate, double annealingRate, double regularization, Reporter reporter, double minImprovement, int minEpochs, int maxEpochs)` Returns the signular value decomposition of the specified complete matrix of values.
`double`	`value(int row, int column)` Returns the value of this matrix at the specified row and column.

Methods inherited from class com.aliasi.matrix.AbstractMatrix

`columnVector, equals, hashCode, rowVector, setValue`

Methods inherited from class java.lang.Object

`clone, finalize, getClass, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

SvdMatrix

```
public SvdMatrix(double[][] rowVectors,
                 double[][] columnVectors,
                 int order)
```

Construct a factored matrix using the specified row and column vectors at the specified order. Each vector in the arrays of row and column vectors must be of the same dimension as the order.

See the class documentation above for more information on singular value decomposition.

Parameters:

`rowVectors` - Row vectors, indexed by row.
`columnVectors` - Column vectors, indexed by column.
`order` - Dimensionality of the row and column vectors.

SvdMatrix

```
public SvdMatrix(double[][] rowSingularVectors,
                 double[][] columnSingularVectors,
                 double[] singularValues)
```

Construct an SVD matrix using the specified singular vectors and singular values. The order of the factorization is equal to the length of the singular values. Each singular vector must be the same dimensionality as the array of singular values.

See the class documentation above for more information on singular value decomposition.

Parameters:

`rowSingularVectors` - Row singular vectors, indexed by row.
`columnSingularVectors` - Column singular vectors, indexed by column.
`singularValues` - Array of singular values, in decreasing order.

Method Detail

numRows

```
public int numRows()
```

Returns the number of rows in this matrix.

Specified by:

`numRows` in interface `Matrix`

Specified by:

`numRows` in class `AbstractMatrix`

Returns:

The number of rows in this matrix.

---

numColumns

```
public int numColumns()
```

Returns the number of columns in this matrix.

Specified by:
`numColumns` in interface `Matrix`

Specified by:
`numColumns` in class `AbstractMatrix`

Returns:
The number of columns in this matrix.

---

order

```
public int order()
```

Returns the order of this factorization. The order is the number of dimensions in the singular vectors and the singular values.

Returns:
The order of this decomposition.

---

value

```
public double value(int row,
                    int column)
```

Returns the value of this matrix at the specified row and column.

Specified by:
`value` in interface `Matrix`

Specified by:
`value` in class `AbstractMatrix`

Parameters:
`row` - Row index.
`column` - Column index.

Returns:
The value of this matrix at the specified row and column.

---

singularValues

```
public double[] singularValues()
```

Returns the array of singular values for this decomposition.

Returns:
The singular values for this decomposition.

---

singularValue

```
public double singularValue(int order)
```

Returns the singular value for the specified order.

Parameters:
`order` - Dimension of the singular value.
Returns:
The singular value at the specified order.

---

leftSingularVectors

```
public double[][] leftSingularVectors()
```

Returns a matrix in which the left singular vectors make up the columns. The first index is for the row of the original matrix and the second index is for the order of the singular vector. Thus the returned matrix is `m×k`, if the original input was an `m×n` matrix and SVD was performed at order `k`.

Returns:
The left singular vectors of this matrix.

---

rightSingularVectors

```
public double[][] rightSingularVectors()
```

Returns a matrix in which the right singular vectors make up the columns. The first index is for the column of the original matrix and the second index is for the order of the singular vector. Thus the returned matrix is `n×k`, if the original input was an `m×n` matrix and SVD was performed at order `k`.

Returns:
The right singular vectors.

---

svd

```
public static SvdMatrix svd(double[][] values,
                            int maxOrder,
                            double featureInit,
                            double initialLearningRate,
                            double annealingRate,
                            double regularization,
                            Reporter reporter,
                            double minImprovement,
                            int minEpochs,
                            int maxEpochs)
```

Returns the signular value decomposition of the specified complete matrix of values.

For a full description of the arguments and values, see the method documentation for `partialSvd(int[][],double[][],int,double,double,double,double,Reporter,double,int,int)` and the class documentation above.

The two-dimensional array of values must be an `m × n` matrix. In particular, each row must be of the same length. If this is not the case, an illegal argument exception will be raised.

This is now a utility method that calls `svd(double[][],int,double,double,double,double,Reporter,double,int,int)` with a reporter wrapping the specified print writer at the debug level, or a silent print writer.

Parameters:

`values` - Array of values.
`maxOrder` - Maximum order of the decomposition.
`featureInit` - Initial value for singular vectors.
`initialLearningRate` - Incremental multiplier of error determining how fast learning occurs.
`annealingRate` - Rate at which annealing occurs; higher values provide more gradual annealing.
`regularization` - A regularization constant to damp learning.
`reporter` - Reporter to which to send progress and error reports.
`minImprovement` - Minimum relative improvement in mean square error required to finish an epoch.
`minEpochs` - Minimum number of epochs for training.
`maxEpochs` - Maximum number of epochs for training. training, or `null` if no output is desired.

Returns:

Singular value decomposition for the specified partial matrix at the specified order.

Throws:

`IllegalArgumentException` - Under conditions listed in the method documentation above.

---

partialSvd

```
public static SvdMatrix partialSvd(int[][] columnIds,
                                   double[][] values,
                                   int maxOrder,
                                   double featureInit,
                                   double initialLearningRate,
                                   double annealingRate,
                                   double regularization,
                                   Reporter reporter,
                                   double minImprovement,
                                   int minEpochs,
                                   int maxEpochs)
```

Return the singular value decomposition of the specified partial matrix, using the specified search parameters.

The writer parameter may be set to allow incremental progress reports to that writer during training. These report on RMSE per epoch.

See the class documentation above for a description of the algorithm.

There are a number of constraints on the input, any violation of which will raise an illegal argument exception. The conditions are:

- The maximum order must be greater than zero.
- The minimum relative improvement in mean square error must be non-negative and finite.
- The minimum number of epochs must be greater than zero and less than or equal to the maximum number of epochs.
- The feature initialization value must be non-zero and finite.
- The learning rate must be positive and finite.
- The regularization parameter must be non-negative and finite.
- The column identitifer and value arrays must be the same length.
- The elements of the column identifier array and the value array must all be of the same length.
- All column identifiers must be non-negative.
- Each row of the column identifier matrix must contain columns in strictly ascending order.

Parameters:

`columnIds` - Identifiers of column index for given row and entry.
`values` - Values at row and column index for given entry.
`maxOrder` - Maximum order of the decomposition.
`featureInit` - Initial value for singular vectors.
`initialLearningRate` - Incremental multiplier of error determining how fast learning occurs.
`annealingRate` - Rate at which annealing occurs; higher values provide more gradual annealing.
`regularization` - A regularization constant to damp learning.
`reporter` - Reporter to which progress reports are written, or `null` if no reporting is required.
`minImprovement` - Minimum relative improvement in mean square error required to finish an epoch.
`minEpochs` - Minimum number of epochs for training.
`maxEpochs` - Maximum number of epochs for training.

Returns:

Singular value decomposition for the specified partial matrix at the specified order.

Throws:

`IllegalArgumentException` - Under conditions listed in the method documentation above.