Final Report:
Understanding, implementing and using *SVD*

Dylan Laurianti, Yilan Dong

Introduction

The singular value decomposition (*SVD*) of a matrix A is a way to split an $m \times n$ matrix into three interpretable components, two unitary matrices U, V and one diagonal matrix Σ such that $A = U\Sigma V^T$. The column vectors of U, V provide insight into patterns in the columns and rows of A, respectively. The diagonal matrix Σ has non-increasing singular values σ_i, which scale the importance of these patterns. Common applications of the *SVD* are to solve $Ax = b$, to approximate or compress a matrix, to recognize patterns in data, to deflate the size of a problem, and to find the pseudoinverse of a matrix. The *SVD* is a more generally applicable matrix decomposition than most others, because it can be used on both real and complex rectangular matrices with no caveats. The general algorithm for computing the *SVD* first converts A to upper bidiagonal form using Householder reflections, so that then A^TA is symmetric and in tridiagonal form. Then the symmetric*QR* algorithm is implicitly applied to A^TA by alternating row and column *Givens* rotations until A is reduced to a diagonal matrix, which is Σ. The rotation calculated from A^TA causes A to have a non-zero entry outside of its two diagonals, which is then bulge-chased down by the following rotations until the matrix is restored to bidiagonal form. The use of a *Wilkinson Shift*, which approximates an eigenvalue of the matrix, converges the superdiagonal entries to zero efficiently. Then U is recovered as the accumulation of the Householder reflections that pre-multiplied A, and the row rotations on A^TA, and V is recovered likewise from the reflections that post-multipied A as well as the column rotations on A^TA.

Reduced *SVD* is a more efficient choice, without much loss of information, compared with full *SVD*. It often involves calculating only the most significant singular values, and their corresponding column vectors of U and V. According to [12], three most common reduced *SVD* are thin, compact and truncated *SVD*. The thin SVD only computes the columns in U corresponding to rows in V. Compact SVD only computes the columns in U and V corresponding to a non-zero singular value in σ. Finally, truncated SVD calculates an arbitrary number of singular values and vectors depending on an cutoff input variable.

For this project, we are interested in understanding, implementing and using *SVD*, because we are fascinated by interpretable nature of the all three components, as well as its general application in fields related to Computer Science. We explored the use cases of, and implemented our own version of *SVD*. Our initial goal was to implement the standard *SVD*, as well as at least two reduced *SVD*. We achieved our goal, by implementing *SVD* algorithm, and all three common methods of reduced *SVD*: thin, compact, and truncated. All of our implementations passed a comprehensive set of test cases detailed later in the report, and all but one demonstrated an increase in performance, compared to built-in *SVD* algorithms.

Previous work

For our literature review, we focused on two aspects: how the *SVD* algorithm came into existence and has evolved, and how it is applied to various different fields.

Evolution of *SVD*

[10] provides an overview of the work of five influential mathematicians who worked on developing the original theory of the singular value decomposition. In 1873, Beltrami discovered the *SVD* for real, square, nonsingular matrices. Independently in 1874, Jordan produced the same results more elegantly, leading to further insight into problem size reduction and characterizing the largest singular value as a maximum of a function. More progress was made in 1889 when Sylvester demonstrated that a matrix can be iteratively diagonalized. The *SVD* was brought to the field of integrals by Schmidt in 1907, where he also created an approximation theorem. Finally, in 1912 Weyl found how to determine the rank of a matrix in the presence of error, expanding the usefulness of Schmidt's approximation theorem.

[13] presents the effects of the development of coupling relations on the Multiple Relatively Robust Representations (MRRR) algorithm. These improvements were then adapted and implemented by the research team in a bidiagonal *SVD* (bSVD) algorithm as part of the LAPACK library. To apply the MRRR, the bSVD can be reduced to the tridiagonal symmetric eigenproblem, which must be done using the normal equations. The new algorithm was shown to be an improvement over existing methods, with a cost of $O(nk)$ for k singular values and vectors.

[11] begins from the idea of how a truncated *SVD* can save runtime compared to the full *SVD*, and develops an even more cost-cutting algorithm for matrices of low rank. This new algorithm's usefulness is not restricted just to the tall and skinny matrices that the truncated method is. Using simultaneous row and column splits to reduce the size of computations needed, the researchers created a merge and truncate algorithm. They found that their algorithm is both faster on single-core processors, and more parallelizable. The algorithm provided even better improvements when working on tall and skinny matrices, and had typical error between 1-2%.

Motivated by the problem of the *SVD* overlooking the importance of outliers and smaller sub-patterns in their datasets, [6] looked to create a modified *SVD* algorithm that would use clustering techniques first to preserve the patterns that only appear in a subset of the data. They used a single-link algorithm to create clusters, then split the clusters into submatrices, ranked the submatrices in order of their size, and then repeatedly calculated a basis vector from the submatrix that is determined through residual ratios to have the most data not yet represented in the existing basis vectors. The application they were concerned with was summarizing the topics across a group of multiple text documents. They found that the unmodified *SVD* algorithm would heavily favor the documents that only focused on one topic, where documents with multiple ideas would have all of their data relatively ignored. In their testing of the algorithm, they found that it was not only more precise for the type of data they were working with, but also produced a much lower dimension solution than the original *SVD*.

[3] saw that none the existing dimensionality reduction algorithms were specialized for sparse matrices, and sought to find a new algorithm tailored to that common kind of data. They used the idea of a coreset, a weighting of a subset of the rows in the matrix such that the sum of squared distances from a point to the coreset approximates the sum of squared distances from the point to the original data. The researchers showed that for every matrix A there exists a (k, ϵ)-coreset of cardinality $|C| = O(k/\epsilon^2)$, where k is the dimensionality reduction, and ϵ is the error ratio. These coresets can then be used to approximate the matrix, while having much lower dimensionality.

[7] presents randomized algorithms for *PCA* and *SVD* calculation. They found that the deterministic *SVD* algorithms can end up returning left singular vectors that are far from orthonormal, and use randomized algorithms to generate the correct orthonormal singular vectors. They focused on algorithms for two specific applications of the *SVD*, the reduced *SVD* and low-rank approximation. They found that their algorithms at least matched the performance and accuracy of the deterministic algorithms, and in the types of cases that caused the deterministic algorithms trouble, the randomized algorithms far outperformed them.

Applications of *SVD*

The generalizablity of *SVD* on non-square and complex matrices makes it an ideal candidate in solving real-world problems. In fact, the use of *SVD* is prevalent in various fields related to Computer Science, such as Network Science, Natural Language Processing, Information Retrieval, Computer Vision and Bioinformatics. In particular, *SVD* is often used during data prepossessing to reduce the size and dimension of the data, as well as to minimize the effects of noisy data: since the singular values, Σ, and their corresponding vectors in U and V, are arranged by the order of importance, we can easily reduce the size and dimension of data using *SVD*, by keep only left-most t columns of U, the first t singular values, and the first t rows of V.

Text categorization is a fundamental component of natural language processing. [5] proposes a new text classifier in the form of Artificial Neural Network (ANN) for Arabic text. *SVD* is used in data preprocessing to reduce the size and the dimension of the data. The authors found the use of *SVD* decreases the time and space complexity, and increases the performance. In particular, *SVD*-supported ANN classifier outperform their counterparts without the use of *SVD*, when evaluated using an Arabic corpus. This research is especially important as it uses an Arabic corpus to measure the performance of various tools, which is something relatively scant yet invaluable in the status quo of the Euro-centric academia.

Efficient and accurate identification of recurring patterns in protein sequences can provide invaluable information on protein interactions, a phenomena that affects our quotidian life. [1] looks at how to extract recurring motif patterns from protein sequences in the Protein Sequence Culling Server (PISCES) dataset through clustering algorithms. The PISCES dataset contains more than 660,000 180-dimensional protein segments, rendering the use of any clustering algorithm that has an undesirable time-complexity impossible, without further data processing. *SVD* is therefore chosen among a pool of similar algorithms to reduce segments, making it possible to generate meaning clusters using rough K-means algorithm within reasonable amount of time. The authors further proposes that *SVD* be used in other bioinformatics research.

[4] proposes a new method of watermarking RBG image. *SVD* is used as an image compression tool and data embedding tool. The researchers find that the proposed method works especially well with JPEG compressions and GIF color reductions. They observe no visible difference before and after watermark applied using this method, and state that the watermarks created are robust in facing conventional attacks.

Lexical matching is an important data-mining tool. Polysemy and homonym, both of which inherent to the interpretation of humans, can be hard for algorithms. The use of *SVD* on clusters is proposed by [14] as a new indexing method in lexical matching. Their method is motivated by the observation that, although *SVD* based Latent Semantic Indexing (LSI) demonstrates good representative quality, and seems to overcome the problems of polysemy and homonym in traditional lexical matching, it shows a limited inter-document discriminitive power. Through the evaluation of an English and a Chinese corpus, the authors claim that the proposed method, compared to other *SVD*-based LSIs, demonstrates improved precision in inter-document similarity measurement. In this case, the incorporation of *SVD* reconcils the difficulties posed by polysemy and homonym, and the lack of discriminatory abilities of other LSIs.

Effective denoising techniques aid in the discovery of seismic frequencies, which can provide important information on geological structures. [8] presents a novel random noise reduction method using *SVD*, based on the observation that there is a high-degree of trace-to-trace correlation in the frequency spectrum of effective signals. More specifically, *SVD* is incorporated to decompose the frequency spectra into eigenimages. The authors then compare the result of filtering through the proposed method with that of traditional filter methods on both synthetic and real-life seismic data, and state that their method demonstrates better performance both in the removal of background noise, as well as in the preservation of the effective signals.

This section is, by no means, all-encompassing: after all, *SVD* is of fundamental importance in countless discoveries. However, it still provides a rather comprehensive view of the capabilities of *SVD*, regarding various fields in which it is applicable. In summary, we found that models incorporating *SVD* often demonstrate better performance than their counterparts who don't.

Methods

Implementation

Our work on implementing the *SVD* algorithms began very ambitiously, with goals to create the whole procedure with minimal dependencies on *Julia*'s built-in *LinearAlgebra.jl* library and on the standard LAPACK functions. As we began to fully understand the depth of the *SVD* algorithm, we continually found it necessary to scale back the layers of abstraction that we chose to implement ourselves. Our understanding of the algorithm evolved as follows. First, we gathered an overview of the necessary steps to the algorithm, containing many terms and ideas that we had yet to comprehend. Then, we looked into the meaning of each of the definitions found in the description of the algorithm, and realized that these were basically algorithms in themselves, and sometimes quite multilayered and complex ones, especially the *implicit-QR* algorithm and bidiagonalization algorithm. Instead of implementing a single algorithm, we found that our initial ambitions would have led us to a rabbit hole deeper than we were prepared for.

Our first attempts at the *SVD* algorithm were fairly clumsy and were worked out through trial and error. The typing of the built-in *LinearAlgebra.jl* package works great within itself, but for an outsider looking to override functionality, the use of many different types to represent matrices of different forms can cause added difficulty. Eventually, we got to a point where everything compiled successfully and we were getting the right form for everything, but we could not get the superdiagonal entries to go to zero. In our debugging work, we saw that the bulge-chasing method was applied properly and that the matrix the algorithm was working on never left bidiagonal form at the end of an iteration, but it was not converging to diagonalized form as we would expect.

After searching for more detailed resources on the algorithm, we made further attempts using the*implicit-QR* method, this time with a much more streamlined element-wise scheme of computations. This implementation showed more promise to be efficient, but still failed to converge to the diagonalized singular value matrix. After much effort to make our *SVD* algorithm from scratch, we decided that it would be more beneficial to develop a working set of algorithms to get some final performance results. Following instructions from [9], these implementations no longer rely on the far superior *implicit-QR* method of *SVD* computation, but instead explicitly calculate the eigenvalues and right eigenvectors of the input matrix, and then reconstruct the left eigenvectors from them. This loss of information causes a significant performance decrease.

We then implemented three reduced *SVD*, all of which apply only to square and tall matrices, based on our implementation of the general *SVD*. Thin *SVD*, corresponding to the function *myThinSVD*, is a specialized version of *mySVD* on tall matrices; compact *SVD*, implemented through function *myCompactSVD*, keeps only the non-zero singular values. Finally, truncated *SVD* accepts a parameter t, in addition to matrix A, and keeps only the t most significant singular values, and their corresponding t most important column vectors of U and V. In the last two algorithms, since we do not want to compute all of the singular values, we must first convert A^TA to tridiagonal symmetric form by Hessenberg decomposition, so that the *eigen* function can only compute a specific subset of the singular values.

Performance Analysis

We look at the performance of our *SVD* implementations through two metrics: correctness and time efficiency. Both metrics are measured by comparing our implementation to existing *SVD* algorithms offered by the *Julia* package *LinearAlgebra.jl*, and to the truncated *SVD* algorithm offered by the *Julia* package *TSVD.jl*.

Correctness

We measured the correctness of all of our implementations by a set of comprehensive test suites. For our general *SVD* algorithm, *myGeneralSVD*, we measured its correctness on square matrices of size $n \times n$, as well as on general rectangle matrices of size $n \times m$, where $n \in [3, 500] \cap \{3x|x \in \mathbb{N}^+\}$, and $m \in [1, 500]$ is chosen randomly for each test case. We chose 500 to be the upper bound of matrix size due to the limitation of available computation power. Each matrix entry is a random value between 0, inclusive, and 1, exclusive. For reduced *SVD* algorithms, *myThinSVD*, *myCompactSVD* and *myTruncatedSVD*, we used square matrices of size $n \times n$, as well as on tall matrices of size $n \times m$, where $n \in [3, 500] \cap \{3x|x \in \mathbb{N}^+\}$, and $m \in [n, 500]$ is chosen randomly for each test case. For each implementation, we compare the each component of the factorization output with that yielded by existing algorithms. For the test on each matrix A of size $n \times m$, the output, $(\textit{my}U, \textit{my}\Sigma, \textit{my}V)$, is considered correct, when compared with the output, (U, Σ, V), from existing algorithms if all of the following satisfies 1) all the singular values, i.e. the diagonal of Σ and that of $\textit{my}\Sigma$ are identical; 2) for $i \in [1, n]$, column of $\textit{my}U$, denoted $\textit{my}U_i$ is equal to either U_i, or $-U_i$; 3) similarly, for $j \in [1, m]$, column j of $\textit{my}V$, denoted $\textit{my}V_i$, is equal to either V_j, or $-V_j$; and 4) for all but *myTruncatedSVD*, $\textit{my}U \cdot \textit{my}\Sigma \cdot \textit{my}V^T \approx A$. We chose to compare corresponding columns of U and $\textit{my}U$, V and $\textit{my}V$, as the output to *SVD* is not necessarily unique [2]; rather, every pair of corresponding column vectors of $U$and $\textit{my}U$, $V$and $\textit{my}V$ can be off by a minus sign, and the output is still correct.

Time Efficiency

To measure the time efficiency of our algorithms, we chose to look at only square matrices because we have not figured out a straight-forward visual representation, when graphing the run time in relation to matrix sizes of rectangular matrices. We looked at using a set of square matrices of size $n \times n$, where $n \in [3, 500] \cap \{3x | x \in \mathbb{N}^+\}$. Each matrix entry is a random value between 0, inclusive, and 1, exclusive. We run each algorithm on matrices of each size but different entries ten times, and take the average time elapsed.

We compared the time efficiency of *myGeneralSVD*, *myThinSVD* and *myCompactSVD* to taht of the *svd* algorithm provided by *LinearAlgebra.jl* package. We compare the time efficiency *myTruncatedSVD* to two algorithms: *truncatedSVD*, which is implemented using *svd* from *LinearAlgebra.jl*, by simplying throwing away the unwanted entries of the original factorization output; and to *tsvd* from package *TSVD.jl*. We then graphed the input size to time elapsed, for each of the algorithms we implemented. We grouped *myGeneralSVD*, *myThinSVD* and *myCompactSVD* together, to make the graph concise, and easier for comparison.

Results

We achieved all of our goals, although not with our initial approach, as mentioned in section **Methods**. This section details the analysis of our implementations, the difficulties we encountered, as well as the limitations to our approaches.

Performance Analysis

As mentioned in the **Methods** section, we analyzed the performance of our *SVD* implementations through two metrics: correctness and time efficiency.

Correctness

Under extensive randomized testing, we found that each of our algorithms always produced results identical to those given by the built-in *SVD*. All algorithms were first tested on square matrices, and then the general algorithm was tested on all shapes of matrices while the reduced algorithms were tested on tall matrices.

Time Efficiency

Graphing the input size against the time-to-compute for these algorithms shows the expected performance improvements. As shown in figure 1, our *myGeneralSVD* and *myThinSVD* match each other's slight performance gain over the built-in algorithm. Both of these algorithms save on computing unnecessary columns of the U unitary matrix. However, the compact *SVD* takes significantly more time. Due to the nature of our randomized data, *myCompactSVD* is a poor choice as these matrices are extremely unlikely to fulfill the condition where we would use a compact *SVD*: $r \ll n$. The negative performance is a result of the extra step of Hessenberg reduction, without any savings from not computing the vectors corresponding to the many zero singular values that we would want our matrix to have if it was optimal to use a compact SVD.

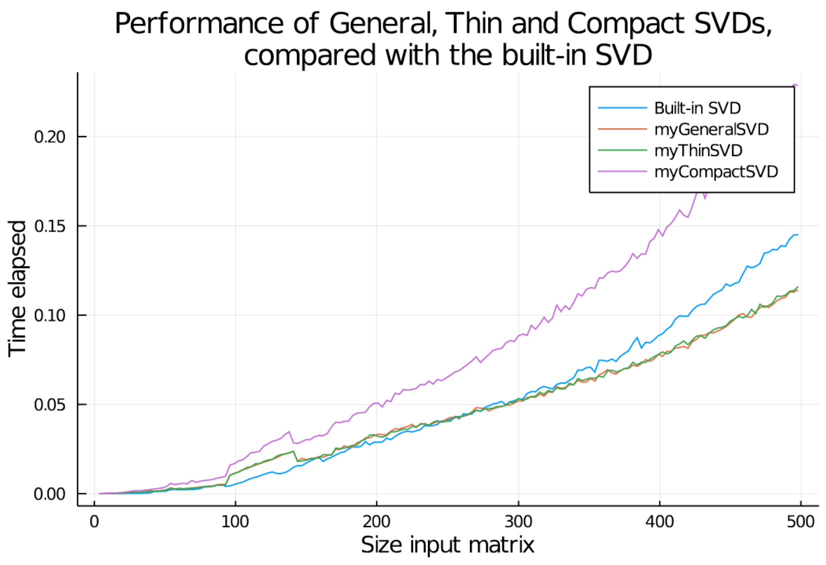

Figure 1: Relative time efficiency of *myGeneralSVD*, *myThinSVD* and *myCompactSVD*.

Our *myTruncatedSVD*, shown in figure 2, also surprisingly showed a performance improvement over both TSVD and a truncated version of the built-in *SVD*. The *truncatedSVD* is simply deleting data after computation, so it does not have any advantage over a full *SVD*. The fact that this TSVD algorithm performs worse than the built-in *SVD* makes this seem like an unfair comparison between truncated algorithms, but we do at least know that our *myTruncatedSVD* offers improvements over the built-in *SVD*.

It is important to note that our implementation work with only real matrices, while the built-in algorithms work with complex matrices, which likely involve much more complicated process. We also rely on optimized existing methods to compute the eigenvalues and vectors. This perhaps explains why our approach in general out-performs the built-in approaches.

Limitations and Difficulties

As mentioned in **Methods** section, we were unable to implement the *implicit-QR* version of *SVD*. Rather, we relied on the built-in eigenvalue and eigenvector computing algorithms provided by *LinearAlgebra.jl*. During our implementations of the *implicit-QR* algorithm, we were unable to

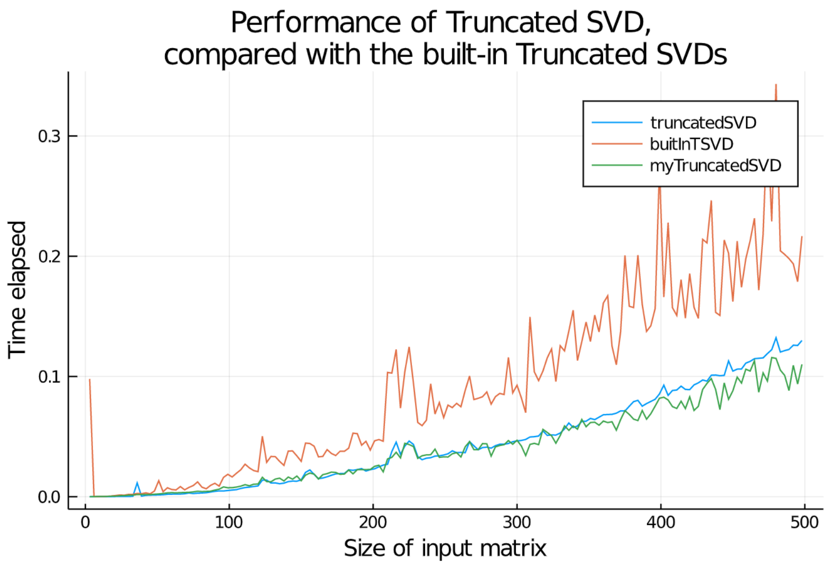

Figure 2: Relative time efficiency of *myTruncatedSVD*.

zero out the superdiagnal entries of what is supposed to be component Σ, as we would expect. This limitation theoretically results in a decrease in time efficiency, as information is lost when computing the eigenvalues and eigenvectors of $A^T \cdot A$ explicitly, and time must be spent later to reconstruct that lost information which is the left singular vectors.

Future Work

Our work on this project leads us to many paths of work in the future. The most pressing of which is to implement an *implicit-QR* version of our *SVD* algorithm, to see how the gains that that implementation yields compared to both the built-in *SVD* and our eigenvalue-based explicit algorithms. We could also look to implement other specialized version of the *SVD* and compare those to standard implementations.

Another avenue for expansion would be in our performance analysis. It would be interesting to generate matrices with different properties and see how the algorithms perform. Specifically, our *myCompactSVD* algorithm could be tested on matrices of different proportions of rank to number of rows, to see the benefit of not computing those singular vectors with a corresponding zero singular value. We could also test matrices with different weighting properties of the singular values, to see our *myTruncatedSVD* algorithm can truncate at different levels of data preservation, and see how that affects time efficiency performance.

References

[1] Elayaraja, E., Thangel, K., Chitralegha, M., and Chandrasekhar, T. Extraction of motif patterns from protein sequences using SVD with rough K-means algorithm. *International Journal of Computer Science Issues 9*, 2 (Nov. 2012), 350–356.

[2] Fedorov, G., and Kuznetsov, S. V. Checking correctness of lapack svd, eigenvalue and one-sided decomposition routines.

[3] Feldman, D., Volkov, M., and Rus, D. Dimensionality reduction of massive sparse datasets using coresets. *CoRR abs/1503.01663* (2015).

[4] Gorfte, Z. E., Cherkaoui, N., Bouzid, A., and Roukhe, A. Multi-data embedding in to RGB image with using SVD method. *International Journal of Computer Science Issues 10*, 2 (Sept. 2013), 190–193.

[5] Harrag, F., Hamdi-Cherif, A., and El-Qawasmeh, E. Performance of MLP and RBF neural networks on Arabic text categorization using SVD. *Neural Network World 20* (Jan. 2010), 441–459.

[6] Huang, F., and Wilks, Y. Clustered sub-matrix singular value decomposition. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Companion Volume, Short Papers*, NAACL-Short '07, Association for Computational Linguistics (USA, 2007), 69–72.

[7] Li, H., Kluger, Y., and Tygert, M. Randomized algorithms for distributed computation of principal component analysis and singular value decomposition. *CoRR abs/1612.08709* (2016).

[8] Liu, B., and Liu, Q. Random noise reduction using SVD in the frequency domain. *Journal of Petroleum Exploration and Production Technology 10*, 7 (Oct. 2020), 3081–3089.

[9] Sauer, T. *Numerical Analysis*. Pearson Addison Wesley, USA, 2006.

[10] Stewart, G. W. On the early history of the singular value decomposition. *SIAM Review 35*, 4 (1993), 551–566.

[11] Vasudevan, V., and Ramakrishna, M. A hierarchical singular value decomposition algorithm for low rank matrices. *CoRR abs/1710.02812* (2017).

[12] Wikipedia. Singular value decomposition.

[13] Willems, P. R., Lang, B., and Vömel, C. Computing the bidiagonal SVD using multiple relatively robust representations. *SIAM Journal on Matrix Analysis and Applications 28*, 4 (Dec. 2006), 907–926.

[14] Zhang, W., Xiao, F., Li, B., and Zhang, S. Using SVD on clusters to improve precision of interdocument similarity measure. *Computational Intelligence and Neuroscience* (Aug. 2016).