Master Thesis

Recommendation System for Galaxy Tools and Workflows

(Find similar tools and predict next tools in workflows)

Anup Kumar

Examiners: Prof. Dr. Rolf Backofen
Prof. Dr. Wolfgang R. Hess
Adviser: Dr. Björn Grüning

University of Freiburg
Faculty of Engineering
Department of Computer Science
Bioinformatics Group Freiburg

July 2018

Thesis period

10. 01. 2018 – 09. 07. 2018

Examiners

Prof. Dr. Rolf Backofen and Prof. Dr. Wolfgang R. Hess

Adviser

Dr. Björn Grüning

Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Place, Date	Signature

Acknowledgement

I would like to extend my sincere gratitude to all the people who encouraged and supported me to accomplish this work. I am grateful to my mentor Dr. Björn Grüning who entrusted me with the task of building a recommendation system for Galaxy. He facilitated this work by providing me with all the indispensable means. Being precise, his pragmatic suggestions concerning the Galaxy tools and workflows helped me discern them better and improve the overall quality of the work. His advice to create a visualizer for showing the similar tools worked wonders as it enabled me to find and rectify a few bugs which were tough to establish. For the next task, creating a separate visualizer for looking through the next predicted tools was conducive in all merits. I offer thanks to Dr. Mehmet Tekman and Joachim Wolff for their expert feedback, insights and general advice. I appreciate and thank Eric Rasche who extracted the workflows for me from the Galaxy Freiburg server. At length, I wish to thank all the other members of Freiburg Galaxy team for their continuous support and help.

Abstract

The study explores two concepts to devise a recommendation system for Galaxy[1] which include finding similar tools[2] and predicting next tools in workflows[3]. The recommendation system can apprise a Galaxy user of the extent to which the tools are related to one another in terms of their functions and types. To collect data about tools, we extract information from the name, description, input and output types and help text attributes. We compute vectors [1, 2] for each tool using data from these attributes and apply similarity measures (jaccard index and cosine similarity) to estimate the distance between each pair of vectors and form similarity matrix for each tool attribute. Each row in the similarity matrix holds similarity scores of one tool with all other tools. To combine these similarity scores, one solution is to compute an average. We use optimization [3] to learn optimal importance weights for the corresponding rows of a tool in the similarity matrices. To define a loss function for optimization, we use a true similarity value based on the similarity measures. Exhibiting an array of next possible tools at each step of picking one while creating workflows can be a meaningful addition to the recommendation system. It would be convenient to leaf through a set of next possible tools as a guide while creating workflows. It can assist the less experienced users if they are indecisive about the next tools. In addition, it can curtail the time taken in creating a workflow. To achieve that, we figure out all the unique paths bridging the input and output tools in the workflows to make them accessible to the downstream machine learning algorithms. We learn the connections among tools to be able to predict the next possible ones based on the previous connections in a path. To learn these connections, we follow a classification approach and use LSTM (long short-term memory), a variant of recurrent neural networks. This network performs well for long-range and sequential (tools connections) data [4, 5]. We report the accuracy as precision.

[1] `https://usegalaxy.eu/`
[2] `https://galaxyproject.org/tools/`
[3] `https://galaxyproject.org/learn/advanced-workflow/`

Zusammenfassung

Contents

List of Figures

List of Tables

Part I.

Find similar Galaxy tools

1. Introduction

1.1. Galaxy

Galaxy[1] is an open-source biological data processing and research platform. It supports numerous types of extensively used biological data formats like FASTA, FASTAQ, GFF, PDB and many more. To process these datasets, it offers tools and workflows which transform these datasets (figure 1). Each tool and workflow furnish exclusive means to process datasets. A simple example of this processing is to merge two compatible datasets to make one. Another example can be to reverse complement a sequence of nucleotides[2].

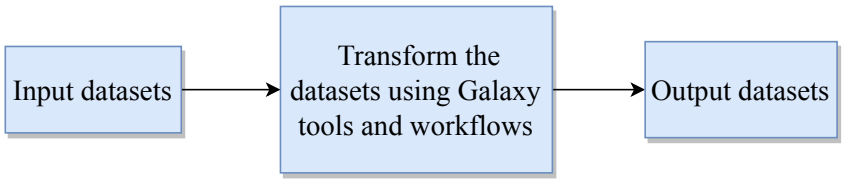

Figure 1.: Dataset transformation: The image shows a general flow of data transformation using Galaxy tools and workflows.

A tool is a data-transforming entity. The tools are classified into multiple categories based on their functions and types. For example, the tools which manipulate text like replacing texts and selecting first lines of a dataset are grouped together under "Text Manipulation" category. These tools form the building blocks of workflows. The workflows are data processing pipelines where a set of tools are joined one after another and protrude from a tool to make branches. The connected tools should to be compatible with each other. It means that the output file types of one tool should be present in the input file types of the next tool.

[1] https://usegalaxy.eu/

[2] https://usegalaxy.eu/?tool_id=MAF_Reverse_Complement_1&version=1.0.1&__identifer=zmk9dx9ivbk

1.2. Galaxy tools

A tool entails a specific function. It consumes a dataset, brings about some transformation and produces an output dataset which can be fed to other tools. It has multiple attributes which include its input and output file types, name, description, help text and so on. These attributes carry more information about a tool. When we look at the collective information about all these attributes for a set of tools, we recognize that some of them have comparable functionalities. There are tools which share affinities in their respective functions and the input and output file types they are glued to. For example, a tool "hicexplorer hicpca" [3] has an output type named "bigwig". If there is a tool which also has "bigwig" as its input and/or output type, we consider there can be some similarity between these tools as they do transformations on similar types of files. In addition, we can find similar functions of tools by analyzing their "name" and "description" attributes. Let's take an example of two tools (figure 2):

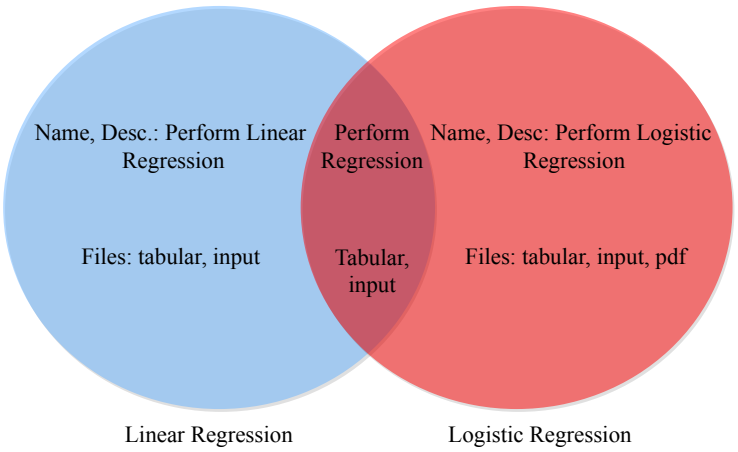

Figure 2.: Common features of two tools using venn diagram: The venn diagram shows the common features of two tools - linear regression and logistic regression. Based on these common features, we assess the extent of similarity between them.

In figure 2, we take two tools - "linear regression" and "logistic regression" and collect their respective information from their input and output file types, name and description attributes. We can see that these tools share some features. They share a similar function of doing regression and few file types are also common. Similarly, if

[3] `https://usegalaxy.eu/?tool_id=toolshed.g2.bx.psu.edu/repos/bgruening/hicexplorer_hicpca/hicexplorer_hicpca/2.1.0&version=2.1.0&__identifer=5kcqmvb71gx`

we extrapolate this notion of finding similar features among all the tools, we hope to find a set of similar tools for each tool. Also, it is possible that we end up with an empty set of similar tools for a tool.

1.3. Motivation

In figure 2, we see that there are tools which share characteristics. The Galaxy has thousands of tools having a diverse set of functions. Moreover, new tools keep getting added to the older set of tools. From a user's perspective, it is hard to keep knowledge about so many tools. In addition, it is important to make users aware of the presence of newly added tools if they are similar to some existing tool. If we can create a model which dispenses clues about a set of similar tools for each tool, it would give more options to users for their data processing using Galaxy.

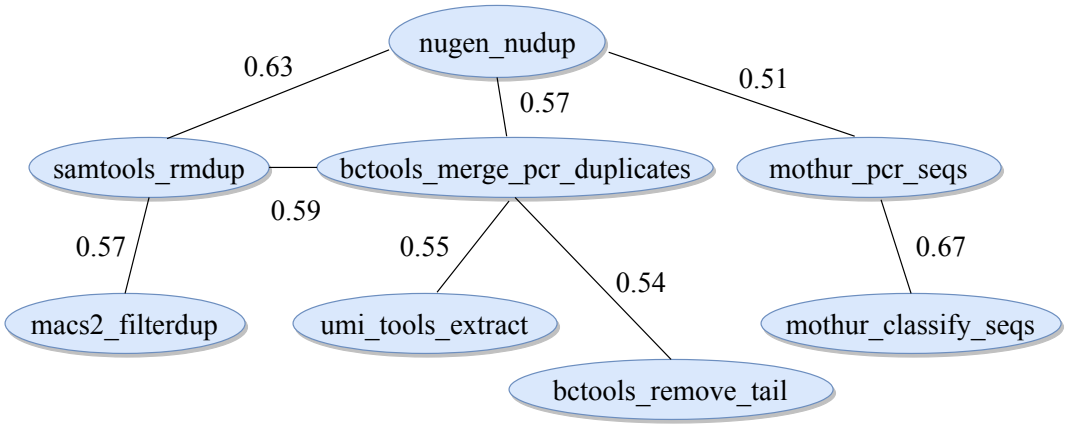

Figure 3.: Similarity knowledge graph consisting of tools as nodes: The graph defines a network exhibiting relationship among the tools using similarity scores between a pair of tools (nodes). The similarity score shows the strength of similarity between a pair of tools.

To elaborate it more, let's take an example of a tool "nugen nudup" [4] (figure 3). It is used to find and remove PCR duplicates. The similar tools for it can be "samtools rmdup" and "bctools merge pcr duplicates" which also work on related concepts. These similar tools have their respective sets of similar tools and thereby make a network of related tools. This knowledge network exhibits "connectedness" among tools and can help users find multiple ways to process their data. The strength of this relation may vary from being small to large and we learn a continuous representation

[4] `https://toolshed.g2.bx.psu.edu/repository?repository_id=4f614394b93677e3`

of the relation strength between a pair of tools and not a binary representation (which learns similar tool as 1 and dissimilar one as 0). Figure 3 shows how this similarity knowledge graph can evolve. First, we find similar tools for "nugen nudup" and connect them to their source tool specifying the similarity values as real numbers at the edges. These similar tools further have their own sets of similar tools and so on.

2. Approach

This section give a comprehensive description of our approach to estimate similarity among tools. It includes procedures to extract and clean tools data, learn vectors for each tool, find similarity (correlation) matrices using these vectors and optimize the combination of these matrices to estimate a final similarity matrix (figure 4).

2.1. Extract tools data

As Galaxy is an open-source project, the repositories of tools are stored at GitHub[1]. In these tool repositories, the xml files which start a "tool" tag belong to a tool. We read all of these xml files, extract information from a few of the tool attributes and collect them in a tabular file. This tabular file contains the information about all the tools.

2.1.1. Select tools attributes

A tool has multiple attributes like input and output file types, help text, name, description, citations and more. But all of these attributes are not important and do not identify a tool exclusively. To collect distinguishing information about a tool, we consider only these attributes:

- Input and output file types
- Name and description
- Help text

Moreover, we combine the input and output file types and name and description respectively as they are of similar nature. These combined attributes give complete information about a tool's file types (input and output types) and its functionality (name and description). Further, we take help text attribute as well which is larger in size compared to the previous two. It can also be empty for some tools. Apart

[1] `https://github.com/galaxyproject/tools-iuc/tree/master/tools`

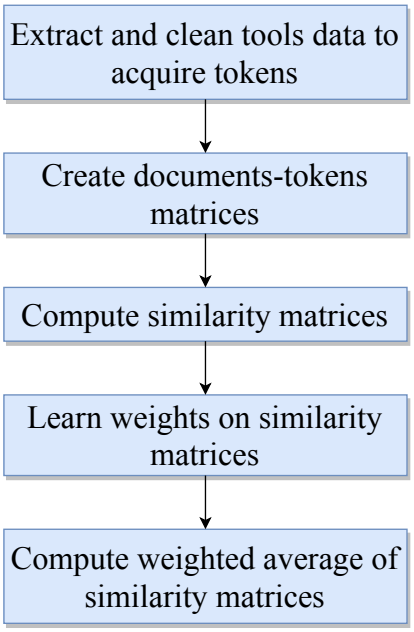

Figure 4.: Sequence of steps to find similar tools: The flowchart shows a series of steps used to establish similarity among tools using approaches from natural language processing to compute text similarity scores and optimization to combine optimally.

from being large in size, it is noisy. It provides more information about the usage of a tool. In the first few lines, it gives a detailed explanation of a tool's functions. Further, it explains how the input data should be supplied to a tool or how an input data looks like. Much of the information contained by this attribute is not important to clearly distinguish a tool. Hence, we decide to use only the first few lines (4 lines) of text present in help text attribute of a tool which illustrates its core functionality. The rest of the information in help text is discarded. The decision to select only the first 4 lines is empirical.

2.1.2. Clean tools data

Remove duplicates and stopwords

The collected data for tools is raw as it contains lots of commonplace and duplicate items which do not add value. These items should be removed to get *tokens* which are unique and useful. For example, a tool "bamleftalign" has input files as "bam" and "fasta" and output file as "bam". While combining these file types, we discard the repeated ones. In this case, we consider the file types as "bam" and "fasta". The

other attributes we deal with are different from the file types. The files types are already in the form of *tokens*. But, in the attributes like name and description and help text, the words come from English and the explanation contains complete or partially complete sentences. Hence, to process this information, we need strategies that are prevalent in natural language processing[2]. The sentences we write in English contain many words and has different parts. These parts include subject, object, preposition, interjection, verbs, adjectives, adverbs, articles and many others. For our processing, we need only those tokens (words) which categorize a tool uniquely and do away with multiple parts of speech present in the sentences. For example, a tool named *tophat* has a name and description as "TopHat for Illumina Find splice junctions using RNA-seq data". The words like "for", "using" and "data" do not give much value as they can be present for many other tools. These words are called as "stopwords"[3] and we selectively discard them. In addition, we remove numbers and convert all the tokens to lower case.

Use stemming

After removing duplicates and stopwords, our data is clean and contain tokens which uniquely identify corresponding tools. When we frame sentences, we follow grammar which constrains us to use different forms of the same word in varying contexts. For example, a word "regress" can be used in multiple forms as "regresses" or "regression" or "regressed". They share the same root and point towards the same concept. If many tools use this word in varying forms, it is beneficial to converge all the different forms of a word to one basic form. This process is called stemming[4]. We use NLTK[5] package for stemming. It enables us to reduce the size to tokens while keeping the meaning of these tokens same across all the tools.

Learn relevance for words

From now on, we use a term "token" for each word. For example, a tool's name contains "regress, perform" as a set of tokens (words). After discarding duplicate tokens, stopwords and using stemmed words, we have a set of meaningful tokens for all the three attributes - input and output file types, name and description and help text. We call these sets as "documents" (figure 6). The tokens present in these documents

[2] `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328/`
[3] `https://www.ranks.nl/stopwords`
[4] `https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html`
[5] `http://www.nltk.org/`

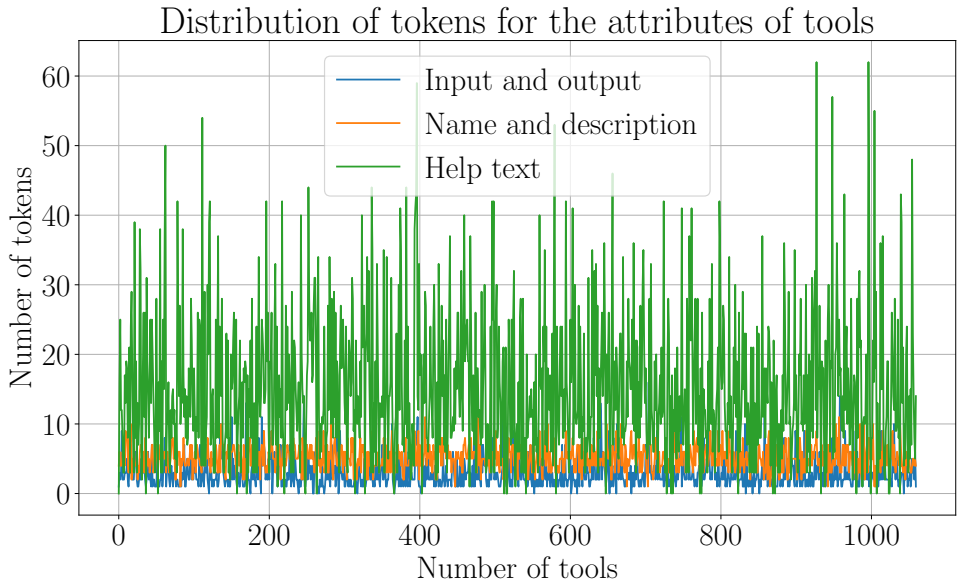

Figure 5.: Distribution of tokens (words) for all the attributes of tools: The plot shows a distribution of tokens for input and output file types, name and description and help text attributes of tools. The help text attribute contains more number of tokens compared to the other two. The input and output file types attribute contains a lower number of tokens compared to the other two attributes.

do not carry equal importance. Some tokens are more relevant to a document and some not so relevant. We need to find out importance factors for all tokens in a document. Using these factors, we can arrange them in big, sparse documents-tokens matrix. In these matrices, each row represents a document and each column belongs to one token. To compute these importance factors, we use BM25 (bestmatch25) [6]. Let's associate some variables to be used in explaining this algorithm.

- Token frequency[6] (tf)
- Inverted document frequency (idf)
- Average document length ($|D|_{avg}$)
- Number of documents (N)
- Size of a document ($|D|$)

[6] `https://nlp.stanford.edu/IR-book/pdf/06vect.pdf`

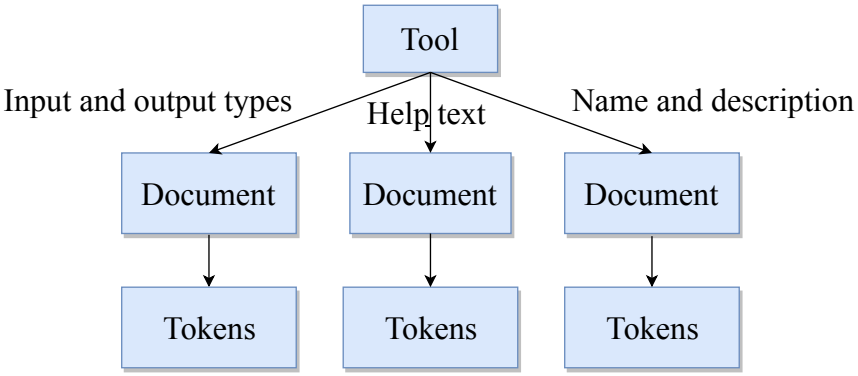

Figure 6.: Relationship between a tool, its documents and their tokens: The image shows that a tool has three documents corresponding to each attribute and each document contains tokens. For all the tools, we have documents equal to the number of tools for each attribute. The number of tokens in each document varies. Minimum number of tokens for any document can be 0.

Token frequency (tf) specifies the count of a token's occurrence in a document. If a token "regress" appears twice in a document, its tf is 2. This can also be understood as a weight given to this term. Inverted document frequency (idf) for a token is defined as:

$$idf = \log \frac{N}{df} \tag{1}$$

where df is the count of the documents in which this token is present and N is the total number of documents. If we randomly sample a document from a set of documents, then the probability of this token to be present in this document is $p_i = \frac{df}{N}$. From information theory, we can say that the information contained by this event is $-\log p_i$. The idf is higher when a token appears in a less number of documents which means that this token is a good candidate for representing that document and thereby, possesses a higher power to distinguish between documents. The tokens which appear in many documents are not good representatives as they tend to be commonplace and do not add much value. Average document length ($|D|_{avg}$) is the average number tokens for all the documents. Size of a document ($|D|$) is the count of all the tokens for that document.

$$\alpha = (1 - b) + \frac{b \cdot |D|}{|D|_{avg}} \tag{2}$$

$$tf^* = tf \cdot \frac{k+1}{k \cdot \alpha + tf} \tag{3}$$

$$BM25_{score} = tf^* \cdot idf \tag{4}$$

where k and b are hyperparameters. The standard value of k is 1.75 and b is 0.75. Using the equation 4, we compute the relevance score for each token in all the documents. Table 1 shows scores for a few documents where the tokens are present with their respective BM25 scores. In this way, we arrange document-tokens matrix for all the attributes of tools. For input and output file types, these matrix entries will have only two value, 1 if a token is present for a document and 0 if not. For other attributes, relevance scores are positive real numbers. This method of representing documents with their tokens is called vector space model as each document represents a vector of tokens.

Documents/tokens	regress	linear	gap	mapper	perform
LinearRegression	5.22	4.1	0.0	0.0	3.84
LogisticRegression	3.54	0.0	0.0	0.0	2.61
Tophat2	0.0	0.0	1.2	1.47	0.0
Hisat	0.0	0.0	0.0	0.0	0.0

Table 1.: A sparse documents-tokens matrix: This table shows a matrix of tools (documents) arranged along the rows and tokens along the columns. Each value in the matrix is a weight (relevance-factor) assigned to a token for a document. This matrix is sparse containing mostly zeros as the number of tokens is significantly large compared to the number of tokens present in a document. This table shows a sample of how actual documents-tokens matrix would look like.

Figure 7 shows a heatmap for documents-tokens matrices that belong to name and description and help text attributes. We can see that these plots are sparse. Each entry in these matrices contains BM25 score for each token in a document. This representation tells us which tokens are better representatives and which are not. But, they do not tell us anything about the co-occurrence of tokens in a document. It tells us that a token is important for a document if the BM25 score is higher but it does not tell us anything about its relation to other tokens. Due to this limitation, it does not acknowledge the presence of "concepts" or "context" hidden in a document. A concept in a document can be realised when we see the relation among a few words. To illustrate this idea, let's take an example of three words - "New York City". These

three words mean little or point to different things if we look at them separately. But, if we see them together, it points towards a concept. The BM25 model lacks the ability to find the correlation among tokens. To learn the hidden concepts within documents and find correlation among multiple tokens, we explore two ideas:

- Latent semantic analysis[7]
- Paragraph vectors

Using these approaches, we learn dense, multi-dimensional vector for each document instead of sparse vectors.

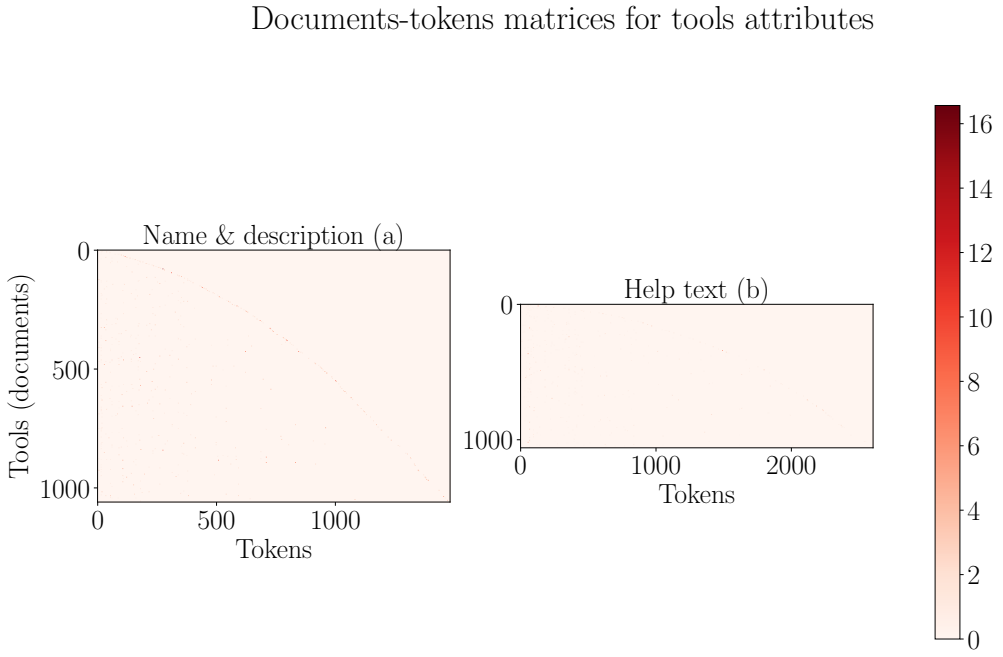

Figure 7.: Heatmap for documents-tokens matrices: The plot shows a heatmap of documents-tokens matrices for name and description and help text attributes. We see that the matrices are sparse containing only a few darker spots. The help text matrix is more sparse than the name and description matrix as the former contains more number of tokens. We exclude the documents-tokens matrix of input and output file types because we do not intend to find concepts among file types. For the other two matrices, we estimate their dense, lower-rank approximations.

[7] `http://lsa.colorado.edu/papers/dp1.LSAintro.pdf`

2.2. Learn dense vector for a document

2.2.1. Latent semantic analysis

It is a mathematical way to learn the hidden (latent) concepts in documents by computing a low-rank representation of a documents-tokens matrix [1, 7, 8]. This low-rank matrix is dense (figure 12). We use singular value decomposition (SVD) for this decomposition. The optimal rank to which a matrix needs to be decomposed to get the best approximation of the full-rank matrix is empirical in nature. We choose ranks from higher to lower for decomposition and consequently the sum of singular values also decreases with the rank. This decomposition follows the equation:

$$X_{n\times m} = U_{n\times n} \cdot S_{n\times m} \cdot V^T_{m\times m} \tag{5}$$

where n is the number of documents and m is the number of tokens. S is a diagonal matrix containing the singular values in descending order. It contains the weights of the concepts present in the documents-tokens matrix. The matrices U and V and orthogonal matrices and satisfy:

$$U^T \cdot U = I_{n\times n} \tag{6}$$

$$V^T \cdot V = I_{m\times m} \tag{7}$$

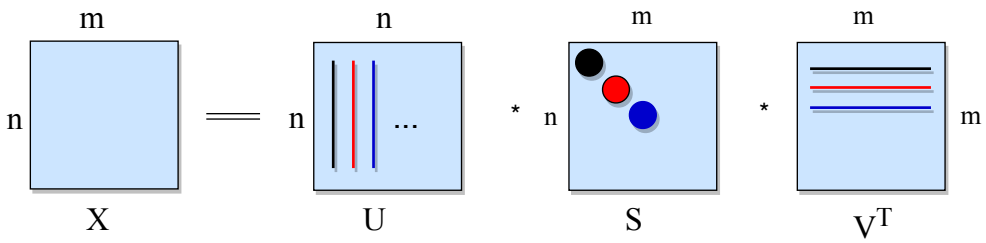

Figure 8.: Singular value decomposition: The image shows that how a matrix is decomposed using singular value decomposition. A matrix X is decomposed into three matrices, U, S and V. The k ($< m$) most important dimensions are kept and the rest are discarded. The value of k is empirical and can be estimated using optimization taking frobenius norm as an error function.

Figure 8 explains[8] how the SVD of a matrix is carried out. The matrix U contains information about how the tokens, arranged along the columns, are mapped to

[8] `http://theory.stanford.edu/~tim/s15/l/l9.pdf`

concepts. The matrix V stores information about how the concepts are mapped to documents arranged along the rows.

Low-rank approximation

The low-rank approximation of a matrix is important to discard the features which are non-repeating. These features with low scaling factors represent noise and by discarding these unimportant features, we can collect the latent relations present in the documents-tokens matrices. We have seen that our documents-tokens matrices suffer from sparsity and exhibit no relation among tokens. The low-rank approximation deals with these issues. The resulting matrices are dense and contain top singular values (which are higher in magnitude). The singular values which are small (the last entries of the S matrix along the diagonal) are discarded [9]. The low-rank approximation X_k ($k < m$) is computed using equation 8. The size of the reconstructed matrix X remains the same but the rank reduces to k.

$$X_{n \times m} = U_k \cdot S_k \cdot V_k^T \tag{8}$$

where U_k is the first k columns of U, V_k is the first k rows and S is the first k singular values. k is an empirical parameter. X_k is called as the rank-k approximation of the full-rank matrix X. Figure 10 shows how the rank varies with the sum of singular values for documents-tokens matrices of all the attributes. Figure 11 shows how the fraction of the sum of singular values alters with the fraction of ranks of documents-tokens matrices for the same attributes. The percentage rank is $k \div K$ where $1 <= k <= K$ and K is the original (full) rank of a matrix. By taking fractions, we bring all the three plots from figure 10 into one plot. From figure 11, we can say that if we reduce the ranks of matrices to 70% of the full-rank, we can still capture $\approx 90\%$ of the sum of singular values. The reduction to half of the full-rank achieves $\approx 80\%$ of the sum of singular values. We show the variation for input and output file types in figure 10 and 11 but we do not reduce its rank. That is for completeness.

We reduce the rank of the original documents-tokens matrices and compute the dense and low-rank approximations. Figure 12 shows the low-rank matrices for name and description and help text attributes. To compute this, we use only 5% of the full-rank. We can compare it with figure 7 and verify that it is denser than figure 7. In these low-rank matrices, we get dense vector representations for documents along the rows. In each matrix, each row contains a vector for one document. Using these documents vectors, we can compute the correlation or similarity using a similarity

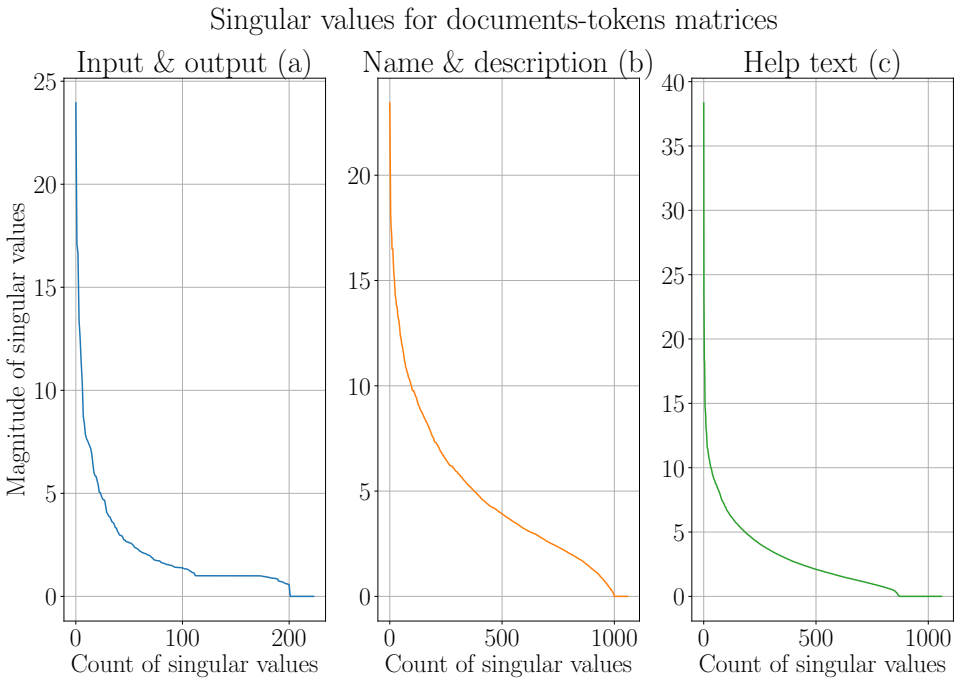

Figure 9.: Singular values of the documents-tokens matrices: The plot shows singular values computed using singular value decomposition (equation 5). The diagonal matrix S contains these singular values sorted in descending order. We can see that in (a), (b) and (c) that very few singular values have higher magnitude and most of the singular values are smaller.

measure. There are multiple similarity measures which can be used like euclidean distance, cosine similarity, manhattan distance, jaccard index and many more. In our case, we use cosine angle similarity for name and description and help text and jaccard index for input and output file types to compute the correlation between vectors. We get a positive real number between 0.0 and 1.0 as similarity score between a pair of vectors specifying how similar they are. The higher the score, higher is the similarity. Computing this similarity for all the documents gives us a similarity matrix $S_{n \times n}$ where n is the number of documents (tools). This square, symmetric matrix is called as similarity or correlation matrix. We compute three such matrices, each corresponding to one attribute (input and output file types, name and description and help text).

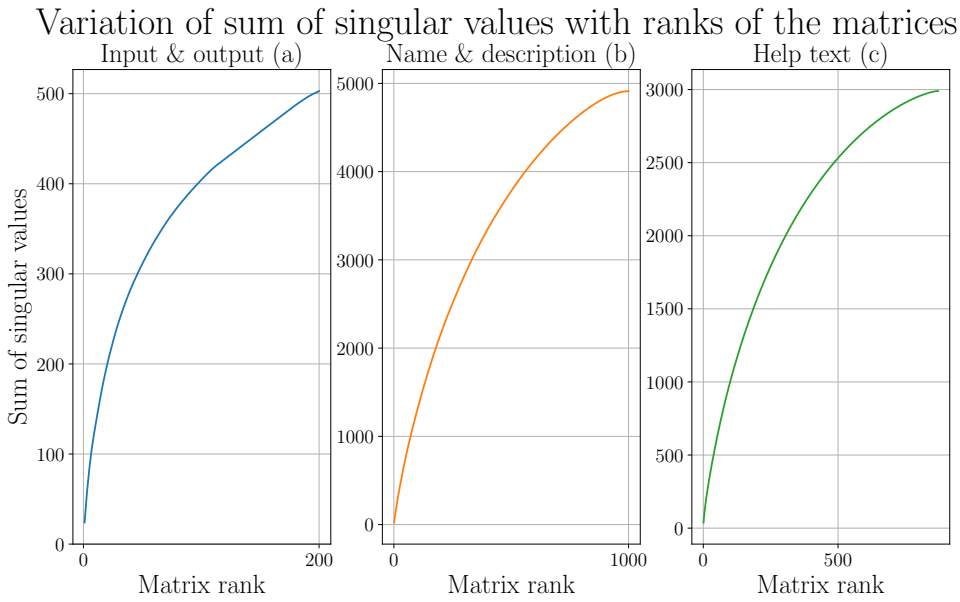

Figure 10.: Sum of singular values with matrix rank: The plot shows an easier way to see how the sum of singular values varies with a documents-tokens matrix rank for the three attributes. Here the (a), (b) and (c) show separately this variation as the ranks of these matrices and sum of singular values differ.

2.2.2. Paragraph vectors

Using latent semantic analysis, we learned dense vectors to represent each document. It learns better vector representations for documents compared to using full-rank documents-tokens matrices. One main limitation is to assess the quantity by which we need to lower the rank of a matrix in order to find the optimal results. There are ways to find the optimal reduced rank by optimizing the Frobenius norm but it is not simple. We would see in the analysis section that the similar tools for a tool are more dominated by the scores shared by the input and output file types. Due to this issue, the tools which are similar in their functions do not get pushed up in the ranking ladder. To avoid these limitations, we use an approach known as *doc2vec* (document to vector) [2]. It learns a dense, fixed-size vectors for each document using neural networks. These vectors are unique in a way that captures the semantics present in the documents. The documents which share similar context are represented by similar vectors. When we compute cosine distance between these vectors sharing similar context, we get a higher number (closer to 1.0). It allows the documents to have variable lengths.

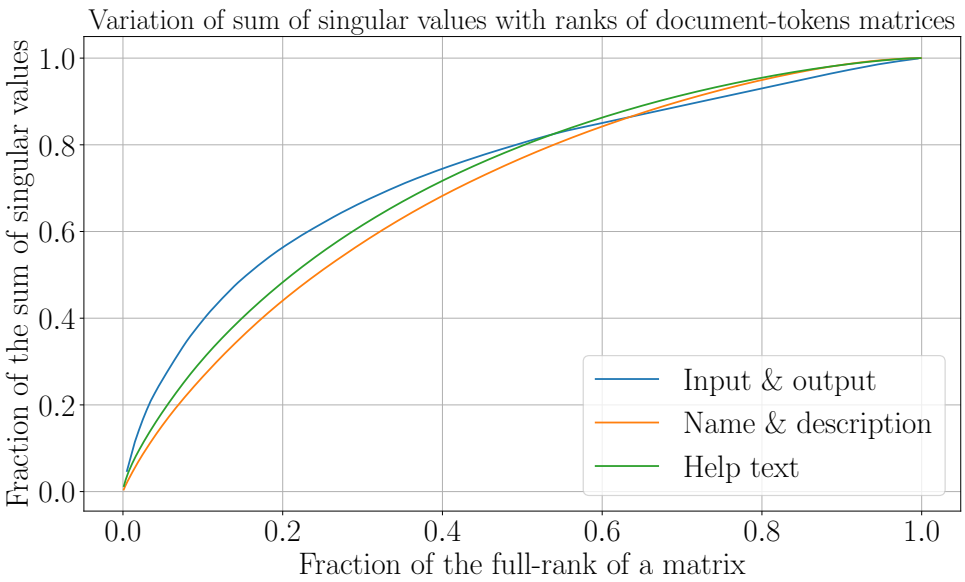

Figure 11.: The variation of the fraction of ranks of documents-tokens matrices with the fraction of the sum of singular values: This plot merges the results of the figure 10 into one plot. As the ranks of document-term matrices and sum of singular values vary, we convert them to respective percentages. $rank_{fraction} = \frac{k}{N}$ where k is the reduced rank and N is the full-rank of a matrix. For example 0.2 on the rank axis (x-axis) means 20% of the original rank of a matrix. Similarly, y-axis shows the fraction of the sum of all singular values $sum_{fraction} = \frac{\sum_{i=1}^{k}}{\sum_{i=1}^{K}}$ where K is the number of all singular values.

Approach

Paragraph vectors approach learns vector representations for all the words and documents (sets of words). The words which are used in a similar context have similar vectors. For example, words like "data" and "dataset" which are used, generally, in similar contexts have are represented by vectors that are close to each other. The vector representations of words in a document are learnt by maximizing the following equation:

$$\frac{1}{T} \cdot \sum_{t=k}^{T-k} \log p(w_t | w_{t-k,\ldots,t+k}) \tag{9}$$

where T is the total number of words in a document, k is the window size. We take a few words which make a context and using this context we try to predict each

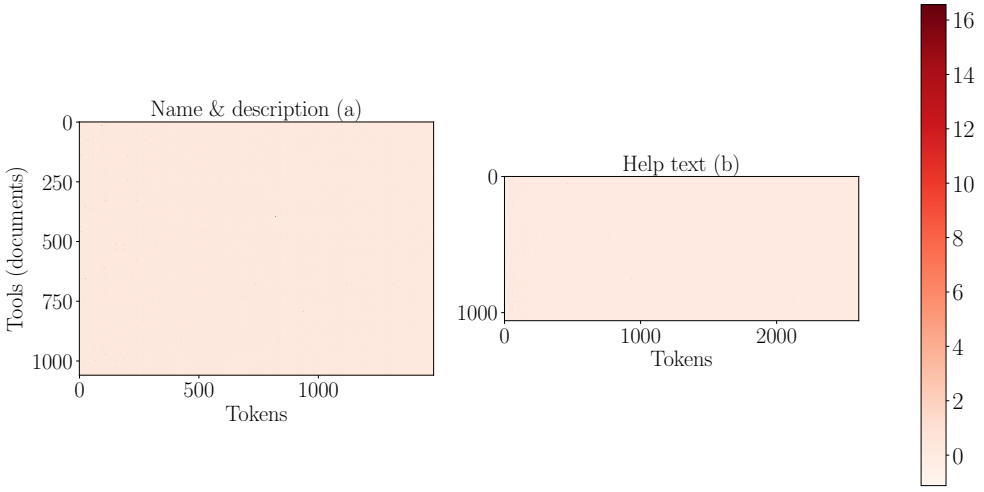

Figure 12.: Low-rank representations of documents-tokens matrices: The heatmap shows the low-rank (5% of full-rank) representations of documents-tokens matrices for name and description and help text. These matrices are denser compared to figure 7 which shows these matrices corresponding to the full-rank representations.

word. The probability p is computed using a softmax classifier and backpropagation is used to compute the gradient and the vectors are optimized using stochastic gradient descent. To learn paragraph vectors, in addition to using words vectors, paragraph vectors are also used to learn the probability of the next words in a context. The paragraph and word vectors are averaged or concatenated to make a classifier which predicts next words in a context. There are two ways how to choose a context.

- Distributed memory: In this approach of learning paragraph vectors, fixed length window of words are chosen and paragraph and word vectors are used to predict the words in this context. The words vectors are shared across all the paragraphs (documents) and paragraph vector is unique to each paragraph.
- Distributed bag of words: In this approach, words are randomly extracted from the paragraphs and from this set of words, a word is chosen randomly and predicted using the paragraph vectors. No order is followed in choosing words.

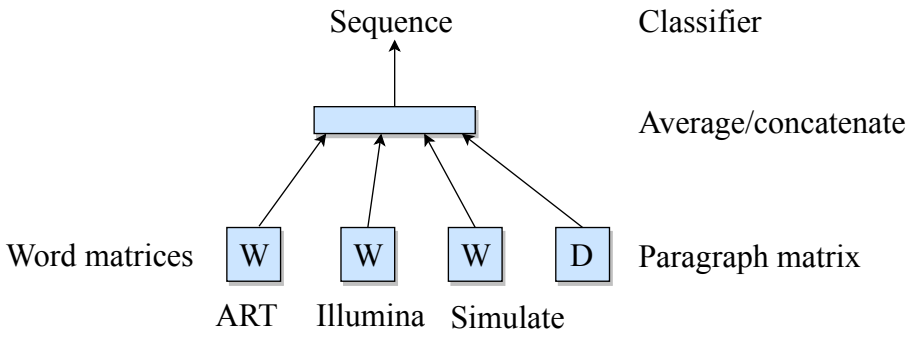

Figure 13.: Distributed memory approach for paragraph vectors: This image shows a mechanism for learning paragraph (document) vectors. W is a word matrix where each word is represented by a vector. D is a paragraph matrix where each paragraph (document) is represented by a vector. The word vectors are shared across all paragraphs (documents) but not the paragraph vectors. The three words "Art", "Illumina" and "Simulate" represent a context. The average or concatenated words and paragraph vectors are used to predict the "Sequence" word.

The figures 13 and 14 are inspired from the original work - Distributed Representations of Sentences and Document[9]. The second form of learning paragraph vectors is simple and we use it to learn documents (paragraphs) vectors for name and description and help text attributes. We learn only the paragraph vectors which makes it less computationally expensive [2] as the number of parameters is less compared to the distributed memory model which learns word vectors as well. The number of parameters in distributed bag-of-words is $\approx N \times z$ where N is the number of paragraphs (documents) and z is the dimensionality of each vector.

2.3. Similarity measures

From latent semantic analysis and paragraph vectors approaches, we generate vectors for the documents that belong to the three attributes. To find similarity between a pair of vectors, we can apply some similarity measures to get a similarity score. This score quantifies how much similar a pair of documents are. In this work, we use two similarity measures - cosine similarity and jaccard index. Both of these measures return a positive real number between 0 and 1 as a similarity score.

[9] `https://arxiv.org/abs/1405.4053`

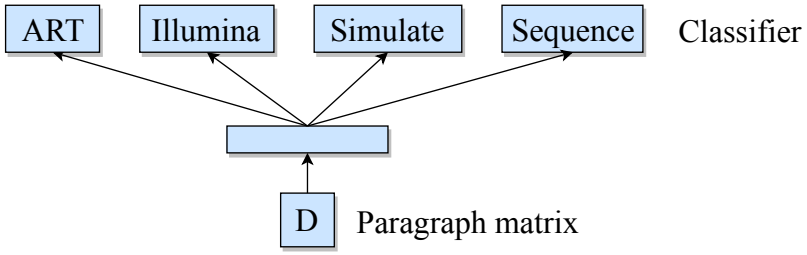

Figure 14.: Distributed bag-of-words approach for paragraph vectors: This image shows how the paragraph vectors are learned by predicting a random word chosen from a randomly selected set of words. D is a paragraph matrix where each paragraph (document) is represented by a vector. In this approach, the order of the words does not matter.

2.3.1. Cosine similarity

It calculates the cosine angle value between a pair of document vectors. Let's say we have two vectors, x and y. We can write:

$$x \cdot y = |x| \cdot |y| \cdot \cos\theta \tag{10}$$

$$\cos\theta = \frac{x \cdot y}{|x| \cdot |y|} \tag{11}$$

where $|x|$ is the norm of the vector x. If the norm is 0, we use 0 for the value of $\cos\theta$. $x \cdot y$ is the dot product of vectors x and y. The values emitted by this similarity follows a natural progression which means if the documents are dissimilar, then it is 0 and if completely similar, it is 1.0. Otherwise, it lies between 0.0 and 1.0. This score can also be understood as a kind of probability of similarity between a pair of documents[10].

2.3.2. Jaccard index

Jaccard index is a measure of similarity between two sets of entities and is given by the equation:

$$j = \frac{A \cap B}{A \cup B} \tag{12}$$

where A and B are two sets. $\cap$ is the number of entities present in both the sets and $\cup$ is the count of unique entities present in sets A and B [10]. We use this

[10] `https://nlp.stanford.edu/IR-book/html/htmledition/dot-products-1.html`

measure to compute the similarity between two tools based on their file types. For example, "LinearRegression" has 3 file types: *tabular*, *input* and *pdf*. Another tool "LDAAnalysis" has *tabular* and *txt* as file types. The jaccard index for this pair of tools would be:

$$j = \frac{Length[(tabular, input, pdf) \cap (tabular, txt)]}{Length[(tabular, input, pdf) \cup (tabular, txt)]} = \frac{1}{4} = 0.25 \tag{13}$$

2.4. Optimization

We get similarity matrices after applying similarity measures on the document vectors, one each for input and output file types, name and description and help text attributes. These matrices have the same dimensions ($N \times N$, N is the number of documents (tools)). To combine these matrices, one simple idea would be to take an average of the corresponding rows of similarity scores from the matrices. Doing this, we get combined similarity scores of one document (tool) with all other documents (tools). Iterating this process for all the documents would give us a similarity matrix which contains similarity scores of documents with all the other documents. The diagonal entries of this matrix would be 1.0 and all the other entries would be a positive real number between 0.0 and 1.0. Another way to find the combination is to learn the weights on the rows from three matrices and then combine them to obtain optimal similarity scores for a document (tool). The weights are positive real numbers between 0.0 and 1.0 and for each row, they sum up to 1.0. Instead of using fixed importance factors (weights) of 1/3 (3 is the number of matrices), we use an optimization technique to find these real numbers and then combine the matrices by multiplying with these optimal weights to obtain a weighted average optimal similarity matrix.

$$S_k^{optimal} = w_{io}^k \cdot S_{io}^k + w_{nd}^k \cdot S_{nd}^k + w_{ht}^k \cdot S_{ht}^k \tag{14}$$

where weight (w) is a positive, real number and satisfy $w_{io}^k + w_{nd}^k + w_{ht}^k = 1$. S_{io}^k, S_{nd}^k and S_{ht}^k are the similarity scores (corresponding matrix rows) for k^{th} tool corresponding to input and output file types, name and description and help text attributes respectively. Similarity scores S has a dimensionality of $1 \times N$ where N is the number of tools (documents). Each similarity score among a set of documents (tools) is independent of one another. We already have these similarity scores and need to learn the importance weights to compute the optimal similarity scores. We use gradient descent optimizer to learn the weights by minimizing an error function.

To define an error function, we need to set true values where we intend to reach and then we compute mean squared error. If we look at the similarity measure, we see that the maximum similarity between a pair of documents can be 1.0. Hence, the ideal similarity scores for a document with any other document can be at most 1.0.

$$S_{ideal} = [1.0, 1.0, ..., 1.0] \tag{15}$$

where S_{ideal} is the ideal similarity scores for one document against all the other documents. S_{ideal} has a dimensionality of $1 \times N$ where N is the number of documents. Using the ideal score, we can compute the mean squared error we accrue for all the similarity scores from the three attributes separately. After computing the error, we can verify which attribute is closer to the ideal score and which are not. The attributes which measure lower error get higher weights and those which score higher error get lower weights. The next section elaborates how we use gradient descent to do the optimization.

2.4.1. Gradient descent

Gradient descent is a popular algorithm for optimizing an objective function with respect to its parameters. The parameters are the entities which we want to learn. In our case, these are the weights. The algorithm minimizes an error function. The error function which we define is the mean squared error:

$$Error^k_{io}(w_{io}) = \frac{1}{N-1} \cdot \sum_{t=1}^{N-1} (w_{io} \cdot S^t_{io} - S_{ideal})^2 \tag{16}$$

$$Error^k_{nd}(w_{nd}) = \frac{1}{N-1} \cdot \sum_{t=1}^{N-1} (w_{nd} \cdot S^t_{nd} - S_{ideal})^2 \tag{17}$$

$$Error^k_{ht}(w_{ht}) = \frac{1}{N-1} \cdot \sum_{t=1}^{N-1} (w_{ht} \cdot S^t_{ht} - S_{ideal})^2 \tag{18}$$

$$Error^k(w) = \frac{1}{N-1} \cdot \sum_{t=1}^{N-1} (w \cdot S^t - S_{ideal})^2 \tag{19}$$

$$\arg\min_{w} Error^k(w) \tag{20}$$

$$\sum_{i=1}^{3} w_i = 1 \tag{21}$$

$Error$ is a vector - $< Error_{io}, Error_{nd}, Error_{ht}$, w is a weight vector - $< w_{io}, w_{nd}, w_{ht} >$ and S similarity scores vector - $< S_{io}, S_{nd}, S_{ht} >$. io, nd and ht refer to input and output file types, name and description and help text for $Error$, w and S. All these vectors are averaged over $N-1$ where N is the number of documents (tools). We take $N-1$ in order to remove the concerned tool's similarity score with itself. To minimize the equation 20 under the constraint given by equation 21 where $0 \leq w_i \leq 1$ and 3 is the number of attributes, we need to compute the gradient of the error function. The gradient specifies the rate of change of error with respect to the weights.

$$Gradient^k(w) = \frac{\partial Error^k}{\partial w} = \frac{2}{N-1} \cdot \sum_{t=1}^{N-1} (w \cdot S^t - S_{ideal}) \cdot S^t \tag{22}$$

The $Gradient$ is also a vector - $< Gradient_{io}, Gradient_{nd}, Gradient_{ht} >$. Using this gradient vector, we can update the weights. For this, we need to set the learning rate. Finding the right learning rate is important because a higher value can diverge the gradient descent and a lower value can slow down the learning and it could take a lot of time to converge to the minimum of the error function. For each iteration of gradient descent, we employ time-based decay of learning rate.

$$w^k = w^k - \eta \cdot Gradient(w^k) \tag{23}$$

where η is the learning rate.

2.4.2. Learning rate decay

Finding good learning rates forms an important part of the gradient descent optimization. If the learning rate is high, it poses a risk of optimizer divergence. On the other hand, if it is small, the optimizer can take a long time to converge. Both these situations are undesirable. But, we can avoid these drawbacks by starting out with a small learning rate and gradually decrease it over the iterations. It is called as a time-based decay of the learning rate [11].

$$lr^{t+1} = \frac{lr^t}{(1 + (decay * iteration))} \tag{24}$$

where lr^{t+1} and lr^{t} are the learning rates for $t+1$ and t iterations, *decay* controls how steep or flat the learning rate curve is and *iteration* is the gradient descent iteration number. A higher value of decay makes the learning rate curve steep as the learning rate drops quickly. A lower value can make the curve flat which can slow down the learning.

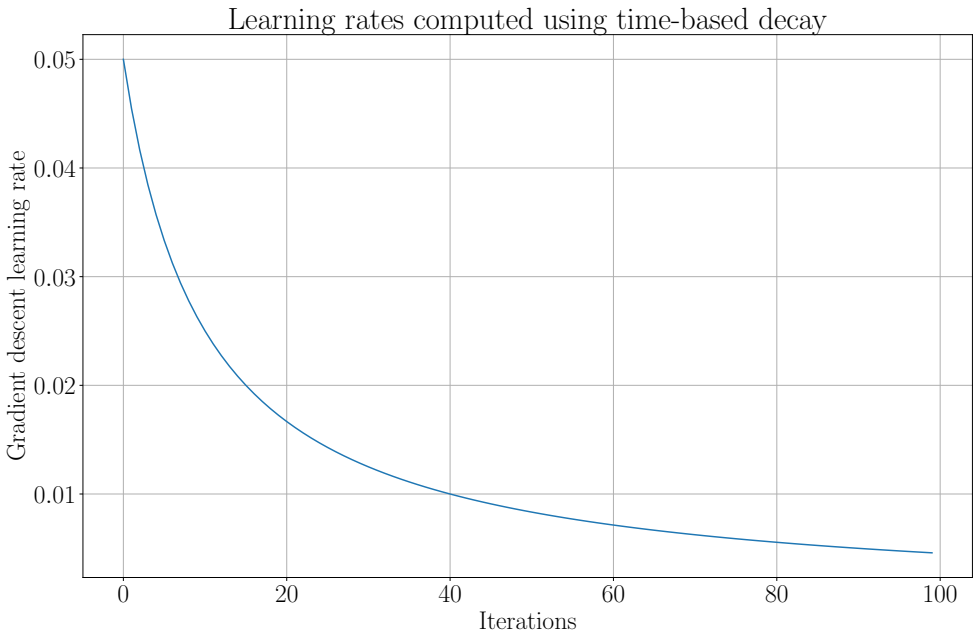

Figure 15.: Decay of learning rate for gradient descent optimizer: The plot shows how the learning rate for gradient descent evolves with iterations. It starts with a small value and decreases gradually over time. It is essential to have learning rates which neither drops too quickly or too slowly. Both of these ways can lead to divergence or slow convergence of the optimizer.

2.4.3. Weight update

Momentum

To reach the minimum point of our convex error function (equation 20), we need to go down continuously without being blocked at the saddle points. These saddle points are where the derivative of a function is zero. Adding a momentum term to the weight parameter, we expect to avoid the local minima and should be able to

converge to the lowest point quickly. It gives the necessary push to keep going down the convex error function by adding a fraction of the previous step update to the current update [11, 12]. We compute the weight parameter update for each iterations using:

$$update_{t+1} = \gamma \cdot update_t - \eta \cdot Gradient(w^t) \tag{25}$$

$$w_{t+1} = w_t + update_{t+1} \tag{26}$$

where $update_{t+1}$ is the update for changing the weight parameter for the current iteration $t+1$. $update_t$ is the previous iteration update. η is the learning rate and $Gradient$ is with respect to the weight parameter w_t.

Nesterov's accelerated gradient

The inclusion of momentum is useful to get necessary advance towards finding the minimum of the error function. However, the speed of going down the slope of the error function should become less if there is a possibility of change in gradient direction. In this situation, the speeding up can be avoided by estimating the forthcoming gradient (gradient for the next step) and then correcting it [13]. We update the weight parameter using the following equations:

$$update_{t+1} = \gamma \cdot update_t - \eta \cdot Gradient(w_t + \gamma \cdot update_t) \tag{27}$$

$$w_{t+1} = w_t + update_{t+1} \tag{28}$$

Gradient verification

We compute gradient using equation 21 and use it to update our weight parameters. To verify that the computed gradient is correct, we can approximate this gradient using the error function which we formulated in equation 19.

$$Gradient(w) = \frac{\partial Error}{\partial w} \approx \frac{Error(w+\epsilon) - Error(w-\epsilon)}{2 \cdot \epsilon} \tag{29}$$

where ϵ is a very small number ($\approx 10^{-4}$). Figure 16 shows the difference of the actual and approximated gradients for all the three attributes.

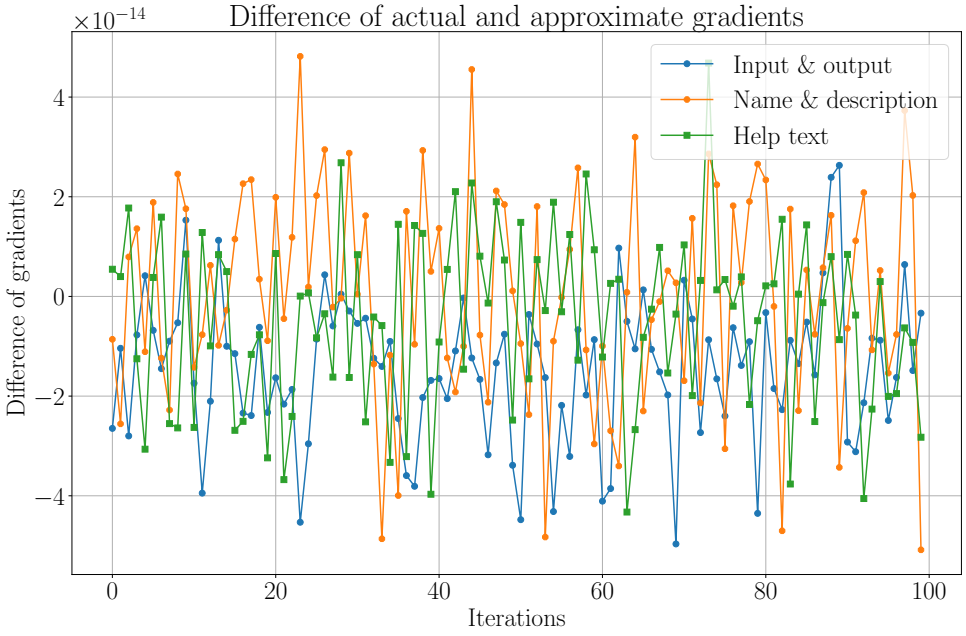

Figure 16.: Verification of gradient for the error function: The plot checks that the difference between the actual and approximated gradients for all the attributes across all tools computed over 100 iterations is close to 0. Gradient plays an important role in learning and it should be estimated correctly. The difference of gradients is consistently $\approx 10^{-14}$ which means that the actual and approximated gradients are same and the gradient we have computed using partial derivatives is correct.

3. Experiments

3.1. Amount of help text

We take a maximum of 4 lines of text from the help text attribute while reading the xml files of tools. This attribute is noisy and contains text which is not useful to be set up as a basis for finding similarity. We need to be careful of the amount of text which we extract from this attribute. Experimentally, we verify that using 4 lines of text from this source works better in most of the cases. The results become worse if we do not use this attribute at all.

3.2. Number of attributes

We consider three different attributes for compiling the collection of tokens. These attributes are different from each other. Using them together to make one set of tokens for each tool is not beneficial. Instead of combining the tokens from these attributes to make one set of tokens for a tool, we compute similarities using tokens from these different attributes and then combine them using an optimization technique and learn optimal weights on each attribute.

3.3. Similarity measures

For calculating similarity scores using input and output file types, we employ the jaccard index because we do not intend to learn any "concept" hidden in a set of file types for a tool. For similarities computed for name and description and help text, we use cosine similarity. Both these similarity measures give a real number between 0.0 and 1.0 as a similarity score for a pair of documents (tools).

3.4. Latent semantic analysis

Using latent semantic analysis, we learn dense vector representation for a document. It reduces the rank of the document-tokens matrix. We need to find out this reduction

factor which can improve the similarity scores and find relevant similar tools among tools compared to using full-rank document-tokens matrix. We follow an approach of lowering the rank by certain factors and look at how the values in these matrices are spread. We reduce the rank of documents-tokens matrices to 70%, 30% and 5% of the full-rank value. Moreover, we also verify the similarity matrices corresponding to these low-rank matrices. We expect the documents-tokens and similarity matrices to be denser compared to no rank reduction. To reduce the ranks, we consider documents-tokens matrices of name and description and help text and leave the input and output matrix in its full-rank state. We use singular value decomposition method from *numpy*[1] linear algebra package.

3.5. Paragraph vectors

In this approach, we learn a fixed-length dense vector for each document by setting the dimensions of these document vectors. When the size of a document is lower, we use a lower number of dimensions for the fixed-length vector. In figure 5, we see that the number of tokens is higher for help text attribute compared to the name and description attribute. To learn fixed-length vectors, we set the vector's length to be 100 for name and description and 200 for help text. Here as well, we learn paragraph vectors only for name and description and help text and not for input and output file types. We use *gensim* model to learn these paragraph vectors.

3.5.1. Distributed bag-of-words

Out of the two approaches to learn paragraph vectors, we apply distributed bag-of-words approach to learn vectors for each document. It does not compute word vectors and relies only on paragraph vectors to predict the randomly chosen words from a sampled text window. Learning only the paragraph vectors is faster and computationally less expensive.

3.6. Gradient descent

To optimize the similarity scores computed from multiple attributes, we execute gradient descent optimizer to learn weights on the scores of these attributes. Based on the similarity measures, we define an error function for the optimizer which iteratively minimizes it.

[1] `https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.linalg.svd.html`

3.6.1. Learning rates

We come up with a time-based decay strategy to calculate learning rate for each iteration. It starts off a higher value of 0.05 and gradually decreases over iterations. The drop in the learning rate is smooth, neither too steep not too flat. We set a maximum iteration of 100 within which the learning saturates comfortably. When we start with 0.1 or higher, there is a risk of optimizer divergence. If we start with 0.001, the learning does not saturate within 100 iterations (the learning becomes slow).

3.7. Code repositories

The codebase used for this study is at github. There are separate branches for the different ideas discussed here. For latent semantic analysis approach, we have two branches - one uses full-rank documents-tokens matrices[2] and another uses documents-tokens matrices reduced to 5% of the full-rank[3]. Both these branches differ only in their ranks of documents-tokens matrices. There is a separate branch for paragraph vectors approach[4]. All these code repositories are under MIT License[5].

[2] `https://github.com/anuprulez/similar_galaxy_tools/tree/lsi`
[3] `https://github.com/anuprulez/similar_galaxy_tools/tree/lsi_005`
[4] `https://github.com/anuprulez/similar_galaxy_tools/tree/doc2vec`
[5] `https://github.com/anuprulez/similar_galaxy_tools/blob/master/LICENSE.md`

4. Results and analysis

4.1. Latent semantic analysis

We experimented with multiple values of matrix rank reduction and we found that as we reduced the rank of documents-tokens matrices, they became denser and the distribution of the learned weights also changed. This observation could be attributed to the fact that document-tokens matrices became denser and as a result, the similarity matrices also became denser. As similarity matrices corresponding to name and description and help text became dense, optimizer learned higher weights on them which accounted for the change in the distribution of weights. We did not apply rank reduction on the documents-tokens matrix of input and output file types.

4.1.1. Full-rank matrices

Figure 17 shows the similarity matrices computed using full-rank documents-tokens matrices for input and output (17a), name and description (17b) and help text (17c) attributes. Using these similarity matrices, we learned each row's respective importance factor using gradient descent and then combined to get a weighted average similarity matrix (17d). Figure 18 shows the distribution of these importance factors (weights) for multiple tool attributes. We see that the magnitude of weights estimated for input and output file types is higher than that of the other two attributes. The higher magnitude of weights is associated with the higher values captured for the similarity matrix of input and output file types (17a) compared to the other two attributes.

4.1.2. 70% of full-rank

In figure 17, we can see that the similarity matrices for name and description and help text are sparse. To reduce the sparsity, we attempted to reduce the ranks of the respective documents-tokens matrices of these two attributes to 70% of full-rank. For example, if the rank of a matrix is 100, we reduce the rank to 70 using singular value decomposition (equation 8). Along with reducing the ranks, we consequently

reduced the singular values of these matrices as well (figure 11). Comparing figures 17 and 19, we see that the name and description and help text similarity matrices start becoming denser. The distribution of the weights also changes (figures 18 and 20). At this stage, it is hard to see the effect of rank reduction. Further, we reduced the ranks drastically to see the effect.

4.1.3. 30% of full-rank

To see the effect of rank reduction, we reduced the ranks of two documents-tokens matrices further to 30% of full-rank. This is a large reduction and would amount to keeping $\approx 60\%$ (removing $\approx 40\%$) (figure 11) of the sum of singular values for all the documents-tokens matrices. In figures 21 and 22, the effect of rank reduction is more visible compared to 70% (figures 19 and 20). We see that the magnitude of weights learned for input and output files starts to decrease and the magnitude of weights for name and description and help text start to increase (figure 22) because the corresponding similarity matrices score become denser (figure 21).

4.1.4. 5% of full-rank

Further, we reduced the ranks of two documents-tokens matrices to 5% of their full-ranks. By choosing this low value, we considered only top $\approx 20\%$ of the sum of singular values (figure 11). From figure 23, we can see that all the similarity matrices corresponding to the attributes are denser compared to figures 17, 91 and 21. Due to this, the weights distribution also changes (figure 24). We learn higher weights for name and description (24b) and help text (24c) compared to the weights for input and output file types (24a).

4.1.5. Improvement verification

To verify that the matrix rank reduction actually works and learns better similarity scores for tools (documents) which are similar in the actual case, we showcase two ways. One idea is to show quantitatively the reduction in mean squared error. Another is to verify the similar tools and the corresponding weights using a static visualizer.

Reduction in error

We observe a drop in mean squared error when we decrease the ranks of the matrices (figure 25). When we reduce the ranks, the similarity scores for the name and description and help text increase. This increase accounts for their larger weights

learned by the optimizer. The weights on an average become more balanced and together with higher similarity scores account for the decrease in the mean squared error. With decreasing mean squared error, the average similarity scores across all the tools using uniform and optimal weights increased. From figures 26 and 27, we find that the average similarity scores using uniform weights get better (from ≈ 0.05 to ≈ 0.12, mean of the orange lines in figures 26 and 27). The blue lines which measure the average similarity computed using optimal weights also record higher similarity (from ≈ 0.075 to ≈ 0.15, blue lines in figures 26 and 27). In the absence of true similarity values, it is hard to establish that we actually improve the performance. To be more certain that we improve the ranking of similar tools, we created a visualizer using javascript and html to look through the similar tools and their respective scores and weights for all tools. The next section explains it.

Static visualizer

We created static html pages (visualizers) to look at the results. For latent semantic analysis approach, we have two such websites. In addition to showing similar tools for the selected tool, they show a few plots for the error, gradient and learning rate drop and the selected tool's similarity scores with all the other tools.

- Use full-rank documents-tokens matrices[1]
- Use 5% of full-rank documents-tokens matrices[2]

4.2. Paragraph vectors

We used paragraph vectors approach to learn fixed-length, multi-dimensional vectors for documents from name and description and help text. The figure 28 shows the documents-tokens matrices for input and output file types (28a), name and description (100-dimensional) (28b) and help text (200-dimensional) (28c). The matrices in 28b and 28c are dense. Using these matrices, we computed corresponding similarity matrices for the attributes (figure 29). We see that the similarity matrices are symmetric and dense. Due to this, the weight distribution also changes (figure 30). We learned higher weights for name and description (30b) and help text (30c) compared to input and output file types (30a). Figure 31 shows that the average similarity scores across all the tools are also higher compared to that of rank reduction

[1] `https://rawgit.com/anuprulez/similar_galaxy_tools/lsi/viz/similarity_viz.html`
[2] `https://rawgit.com/anuprulez/similar_galaxy_tools/lsi_005/viz/similarity_viz.html`

approaches (figures 26 and 27). We increase the average similarity to ≈ 0.30 using optimal weights. For uniform weights as well, we record an average similarity of ≈ 0.25 (figure 31).

Static visualizer

For the paragraph vectors approach, the visualizer[3] shows similar tools by learning vectors for each document. In addition to showing similar tools for the selected tool, they show a few plots for the error, gradient and learning rate drop and the selected tool's similarity scores with all the other tools.

4.3. Comparison of latent semantic analysis and paragraph vectors approaches

We find the similar tools for "hisat", a popular mapping tool, at the different stages of rank reduction and verify if we actually fetch the similar tools. We find similar tools once with full-rank documents-tokens matrices (table 2) and then with 5% of full-rank documents-tokens matrices (table 3). At the end, we find similar tools for "hisat" using paragraph vectors approach (table 4) as well and compare them with the previous approaches. In the tables 2, 3 and 4, we look at the top-2 similar tools for "hisat" with their respective similarity scores for different attributes. The text below each table also gives the values of optimal weights. For example, the weighted similarity score in the first row of table 2 is calculated using the following equation (equation 30). The weights are given at the end of the table's description.

$$0.38 \cdot 0.77 + 0.10 \cdot 0.07 + 0.13 \cdot 1.0 = 0.42 \tag{30}$$

From the tables 2,3 and 4, we can say that the paragraph vectors approach works better than the latent semantic approach to find relevant similar tools by assigning reasonable similarity scores and learning optimal weights on them. In table 3, the similarity scores for name and description are not correct in the true sense as they measure 1.0 even though their descriptions are not exactly same (overfitting). For table 2, in name and description column, they are way too less (0.07 and 0.12). These incorrect interpretations can lead to wrong similarity assessment. Since, we do not compute low-rank estimation of input and output file types matrix, its score remains same in table 2 and 3. In table 4, "hisat2" has the same score in all the tables (2,3

[3] `https://rawgit.com/anuprulez/similar_galaxy_tools/doc2vec/viz/similarity_viz.html`

and 4). In table 4, the tool ranked at number two is more relevant than that from tables 2 and 3.

Similar tools	input & output	name & desc.	help text	weighted similarity
hisat2	0.38	0.07	1	0.42
srma_wrapper	0.5	0.12	0.01	0.4

Table 2.: Similar tools (top-2) for "hisat" extracted using full-rank documents-tokens matrices: The table shows top-2 similar tools selected for "hisat". This set of similar tools uses full-rank documents-tokens matrices. The weights learned by optimization are 0.77 (input and output file types), 0.10 (name and description) and 0.13 (help text).

Similar tools	input & output	name & desc.	help text	weighted similarity
hisat2	0.38	1	1	0.83
srma_wrapper	0.5	1	0.64	0.69

Table 3.: Similar tools (top-2) for "hisat" extracted using documents-tokens matrices reduced to 5% of full-rank: The table shows top-2 similar tools selected for "hisat". This set of similar tools uses documents-tokens matrices reduced to 5% of full-rank. The weights learned by optimization are 0.27 (input and output file types), 0.23 (name and description) and 0.5 (help text).

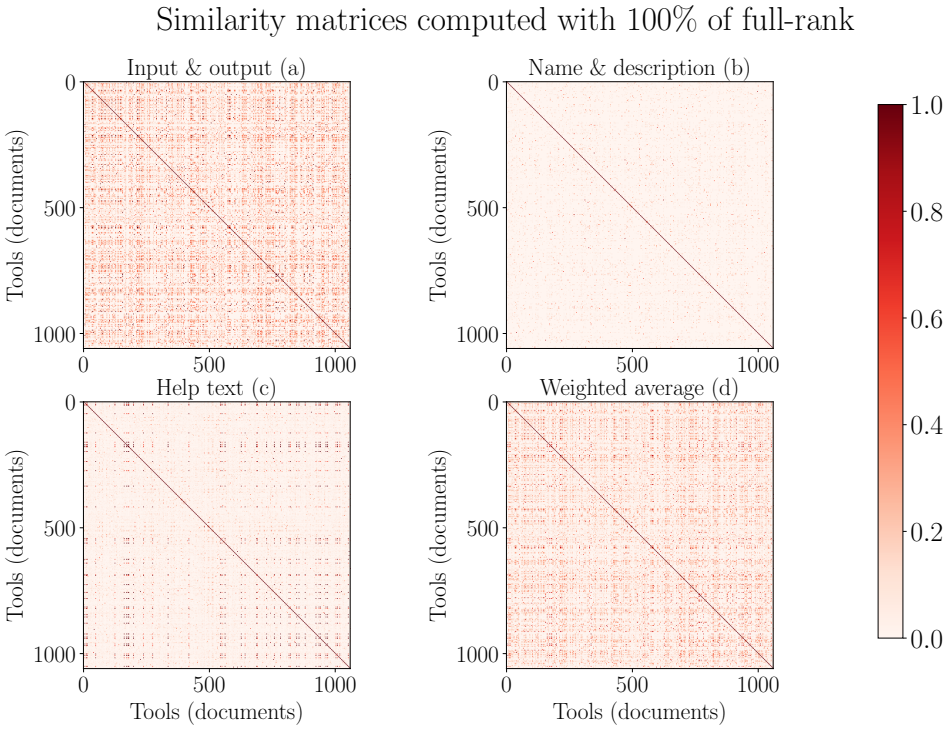

Figure 17.: Similarity matrices computed using full-rank documents-tokens matrices: The heatmap shows documents-documents (tools-tools) correlation matrices for input and output file types (a), name and description (b) and help text (c) attributes. Subplot (d) shows a documents-documents (tools-tools) similarity (correlation) matrix which is the weighted average of similarity matrices computed in (a), (b) and (c) and applying weights (figure 18) learned by the gradient descent optimizer with momentum and accelerated gradient (equation 27 and 28). The corresponding documents-tokens matrices contain their full-ranks.

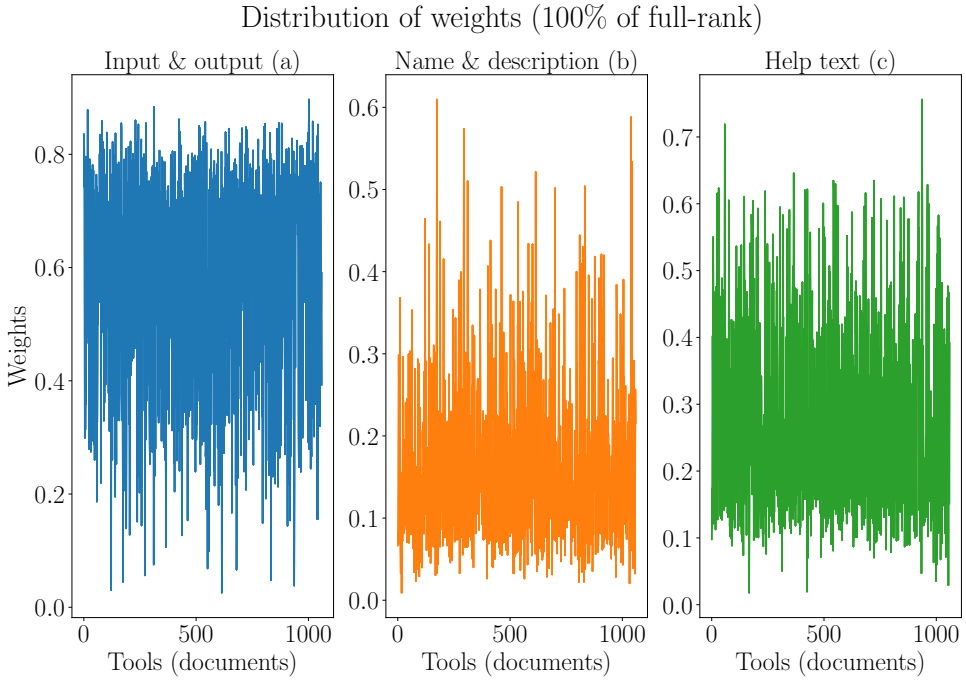

Figure 18.: Distribution of weights learned for similarity matrices computed using full-rank documents-tokens matrices: The plot shows the distribution of weights learned on the similarity matrices (17a, 17b and 17c) by the gradient descent optimizer for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding documents-tokens matrices contain their full-ranks.

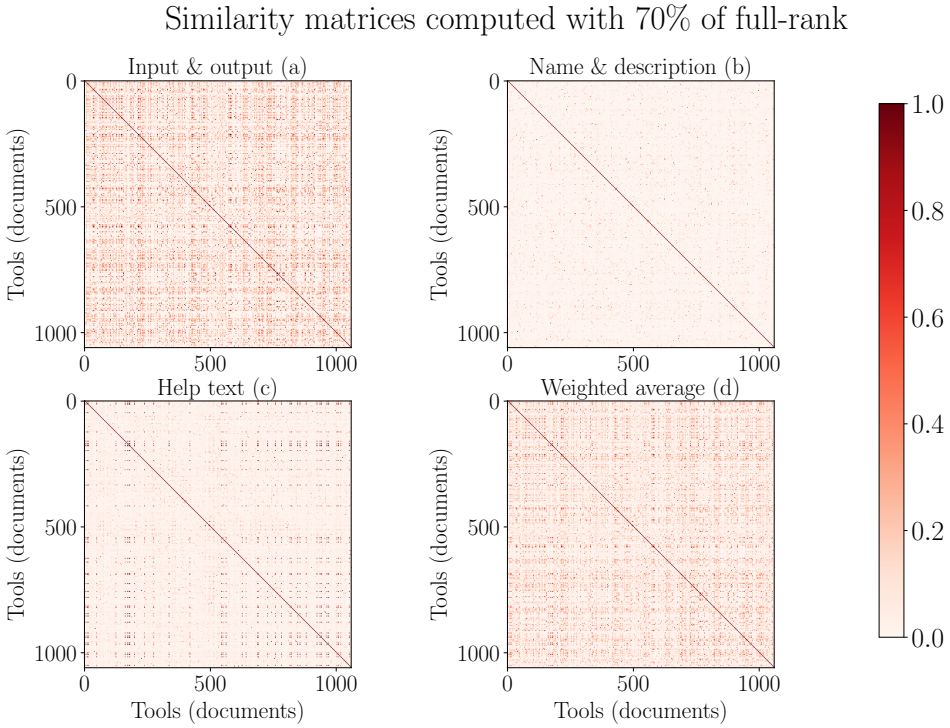

Figure 19.: Similarity matrices using 70% of full-rank: The heatmap shows documents-documents (tools-tools) correlation matrices for input and output (a), name and description (b) and help text (c) attributes. The (d) shows a documents-documents (tools-tools) correlation matrix which is the weighted average computed using (a), (b) and (c) and weights (figure 20) given by the gradient descent optimizer (equation 27). The corresponding documents-tokens matrices are reduced to 70% of their respective full-ranks.

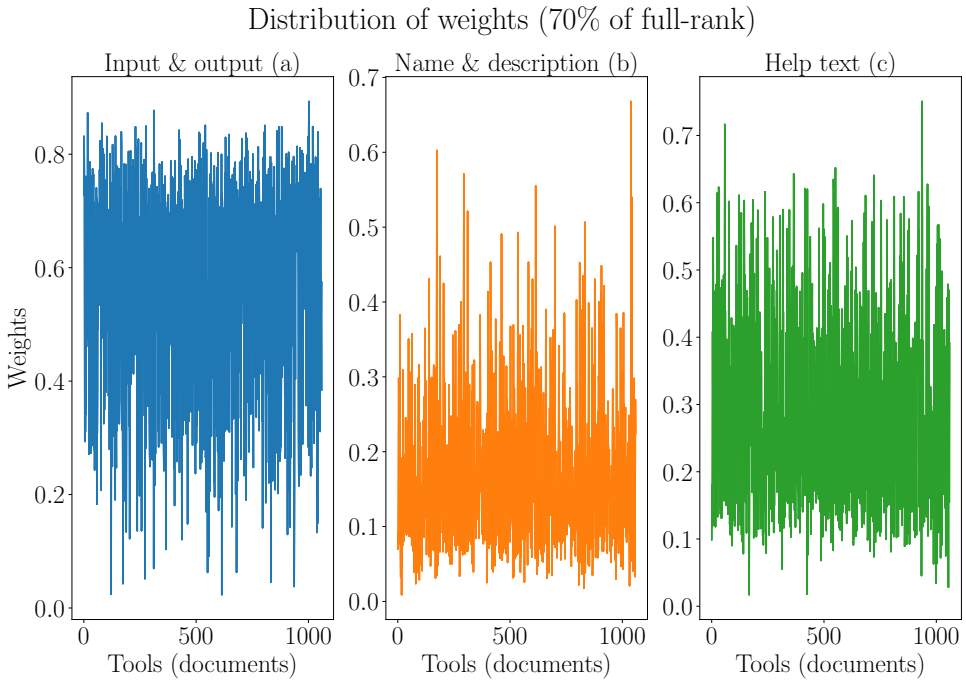

Figure 20.: Distribution of weights learned for similarity matrices computed using documents-tokens matrices reduced to 70% of their full-rank: The plot shows the distribution of weights learned on the similarity matrices (19a, 19b and 19c) by the gradient descent optimizer for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding documents-tokens matrices contain 70% of their full-ranks.

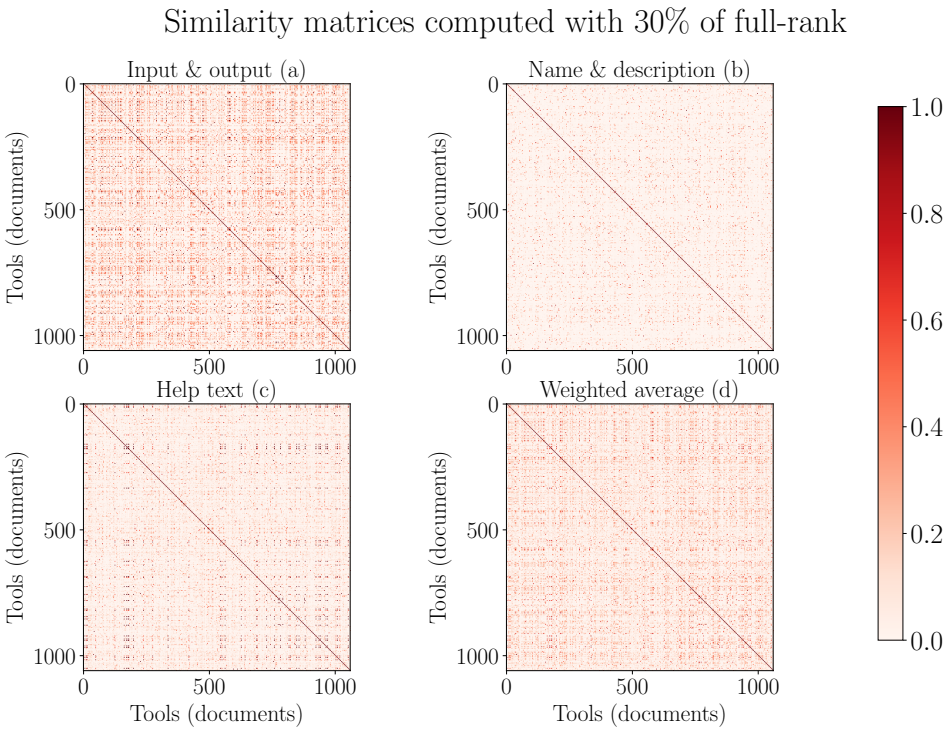

Figure 21.: Similarity matrices computed using document-tokens matrices reduced to 30% of their full-rank: The heatmap shows documents-documents (tools-tools) correlation matrices for input and output (a), name and description (b) and help text (c) attributes. The (d) shows a documents-documents (tools-tools) correlation matrix which is the weighted average computed using (a), (b) and (c) and weights (figure 22) given by the gradient descent optimizer (equation 15). The corresponding documents-tokens matrices are reduced to 30% of their respective full-ranks.

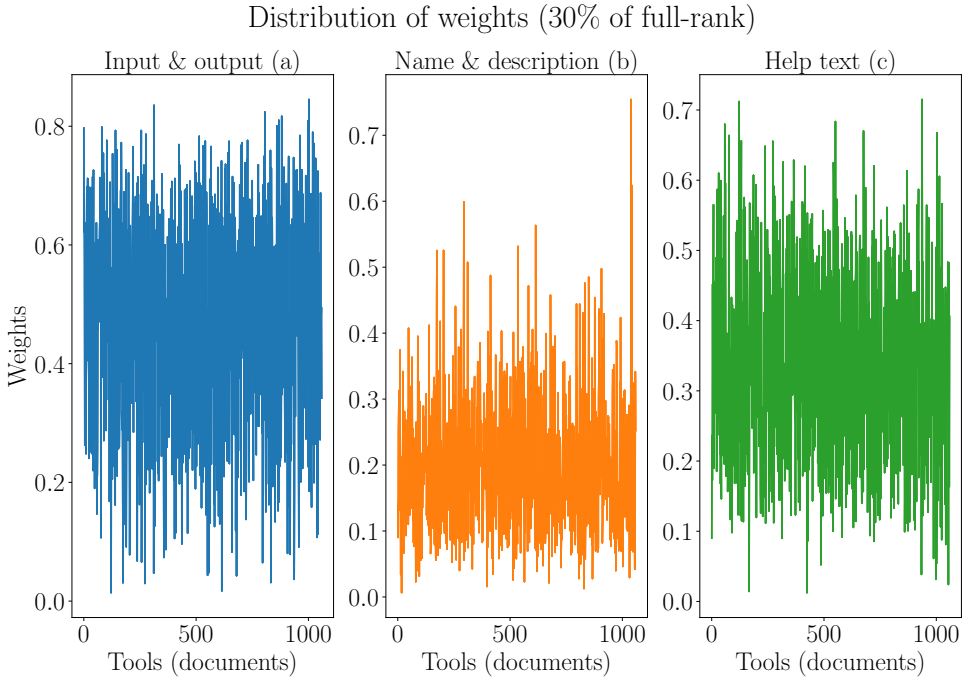

Figure 22.: Distribution of weights learned for similarity matrices computed using documents-tokens matrices reduced to 30% of their full-rank: The plot shows the distribution of weights learned on the similarity matrices (21a, 21b and 21c) by the gradient descent optimizer for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding documents-tokens matrices contain 30% of their full-ranks.

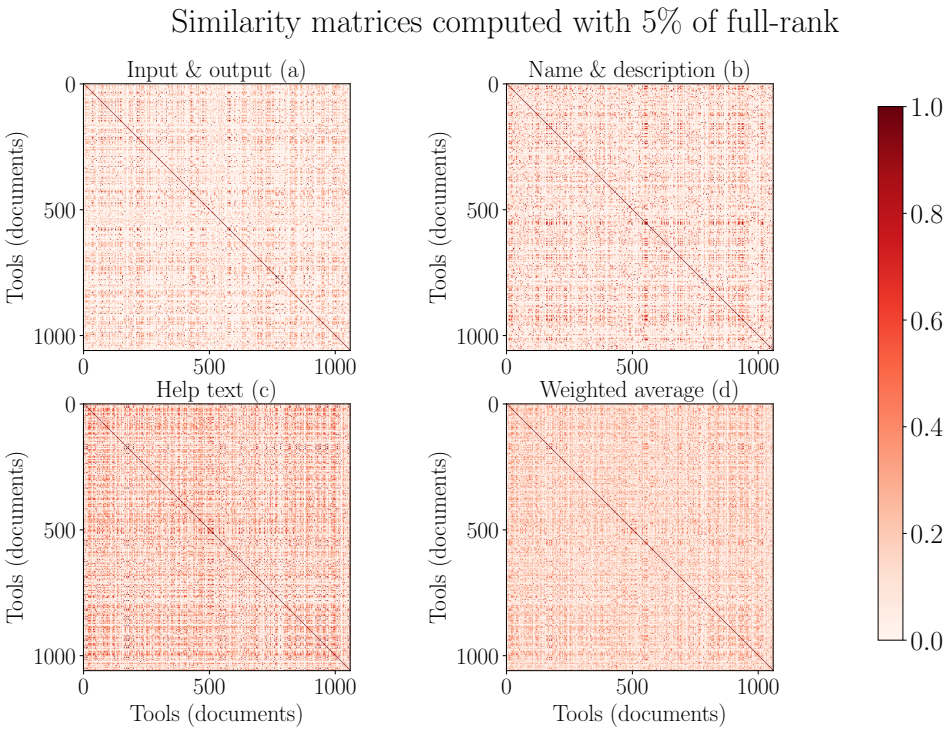

Figure 23.: Similarity matrices computed using document-tokens matrices reduced to 5% of their full-rank: The heatmap shows documents-documents (tools-tools) correlation matrices for input and output (a), name and description (b) and help text (c) attributes. The (d) shows a documents-documents (tools-tools) correlation matrix which is the weighted average computed using (a), (b) and (c) and weights (figure 24) given by the gradient descent optimizer (equation 15). The corresponding documents-tokens matrices are reduced to 5% of their respective full-ranks.

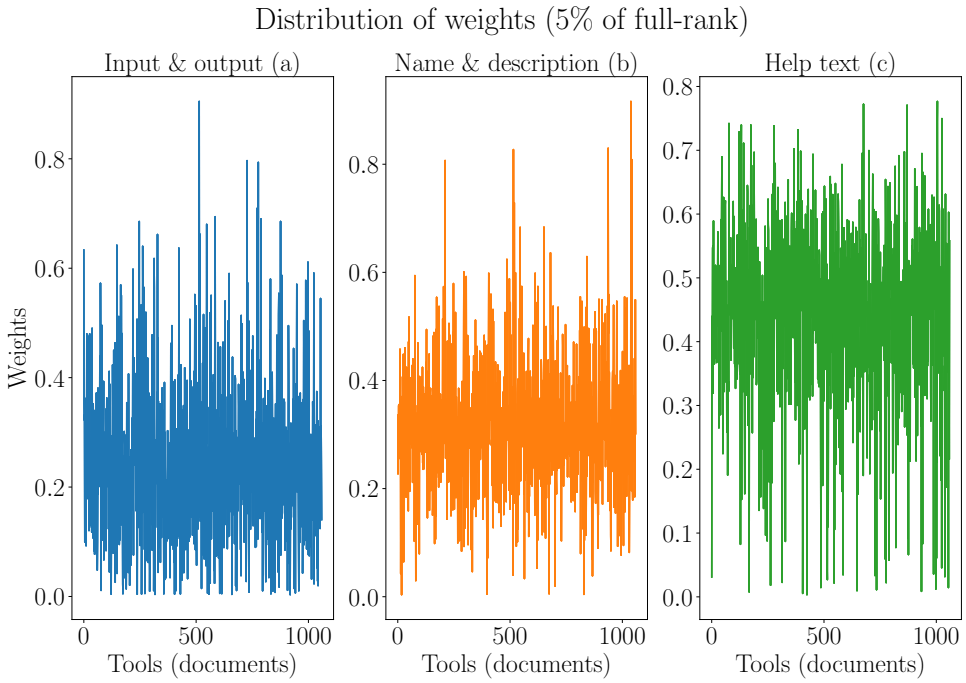

Figure 24.: Distribution of weights learned for similarity matrices computed using documents-tokens matrices reduced to 5% of their full-rank: The plot shows the distribution of weights learned on the similarity matrices (23a, 23b and 23c) by the gradient descent optimizer for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding documents-tokens matrices contain 5% of their full-ranks.

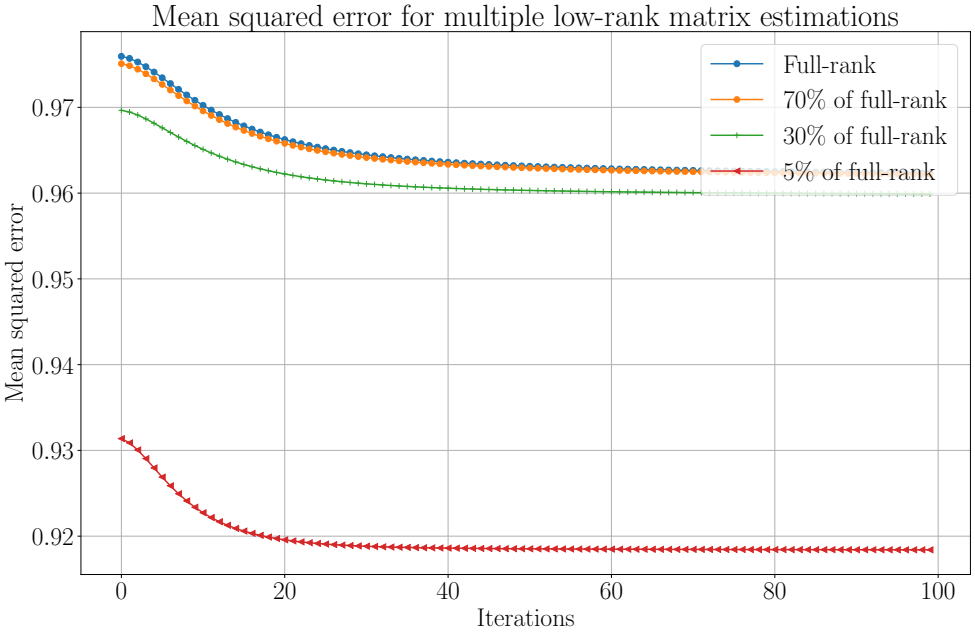

Figure 25.: Mean squared error using full-rank and multiple estimations of low-rank (latent semantic analysis approach) documents-tokens matrices: The line plot shows a comparison of mean squared error computed separately using full-rank and multiple estimations of low-rank documents-tokens matrices. Each line shows an average error over all the tools and attributes which drops and becomes stable as we move along the iterations of gradient descent.

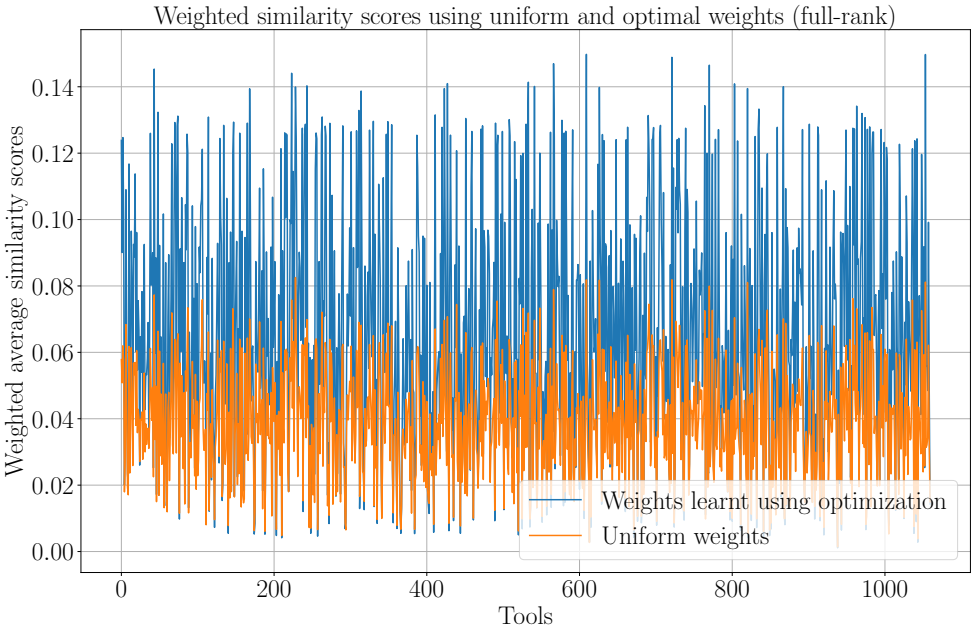

Figure 26.: Average of uniformly and optimally weighted similarity scores computed using full-rank documents-tokens matrices across all tools: The plot shows the average similarity scores where full-rank documents-tokens matrices are used to compute similarity matrices and they are combined using uniform and optimal weights. We see that the average similarity scores computed using optimal weights are higher as compared to using uniform weights to compute similarity scores.

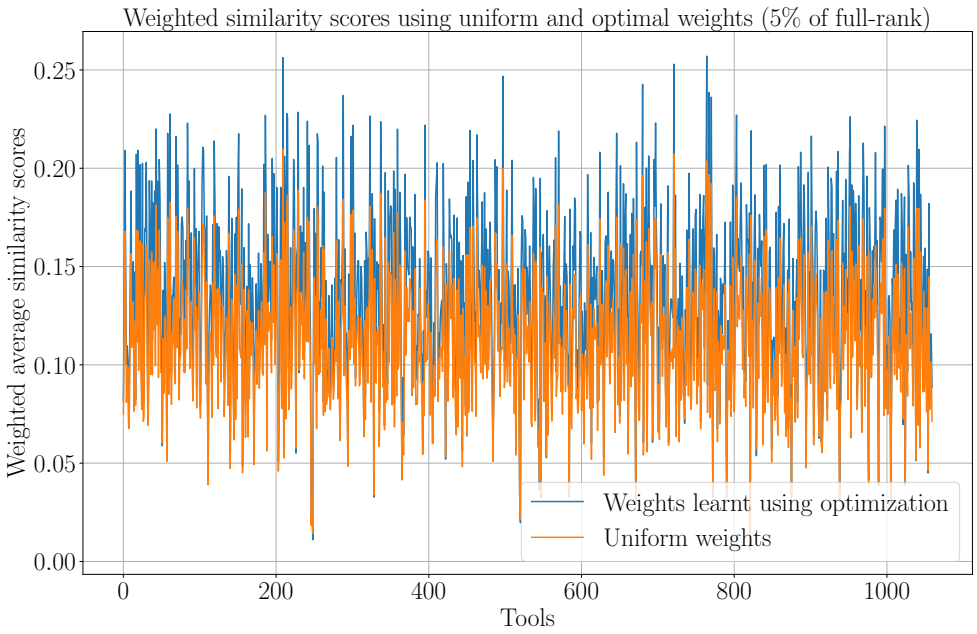

Figure 27.: Average of uniformly and optimally weighted similarity scores computed using documents-tokens matrices reduced to 5% of full-rank across all tools: The plot shows the average similarity scores where documents-tokens matrices reduced to 5% of full-rank are used to compute similarity matrices and they are combined using uniform and optimal weights. We see that the average similarity scores computed using optimal weights are higher as compared to using uniform weights to compute similarity scores. Compared to figure 26, both the approaches, using uniform and optimal weights, measure higher similarity scores.

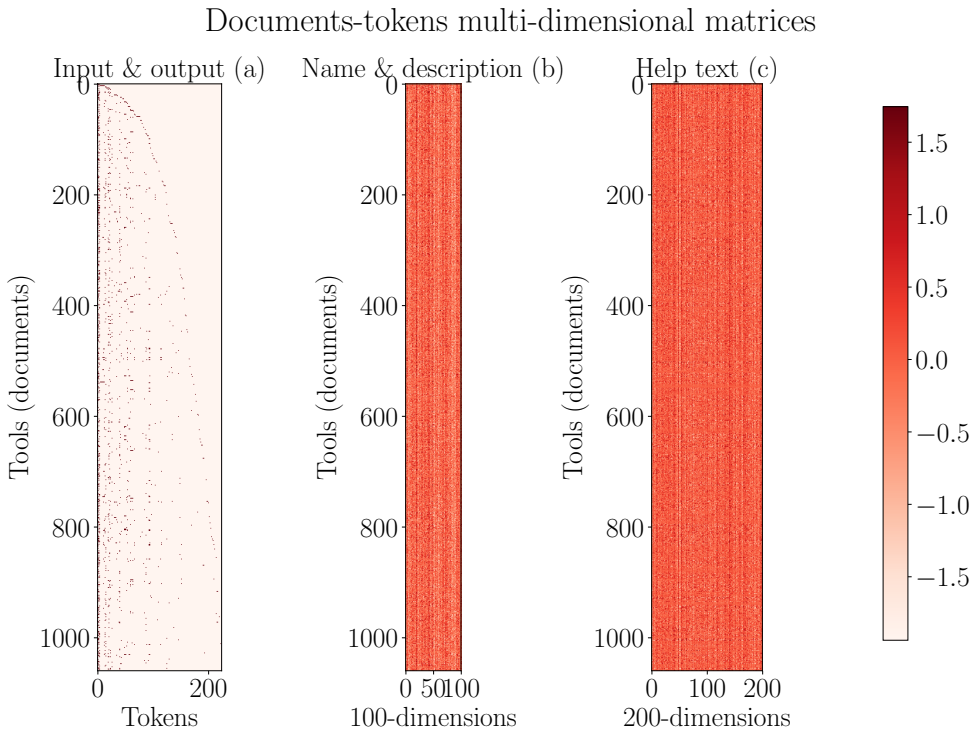

Figure 28.: Documents-tokens matrices for paragraph vectors approach: The heatmap shows documents-tokens matrices for input and output file types (a), name and description (b) and help text (c) attributes. The matrix in (a) is a document-token matrix where each entry shows a relative frequency of the token's occurrence. The matrices in (b) and (c) are 100 and 200 dimensional (each row is a document vector) respectively which means that each row of a matrix belongs to a document (paragraph) and is fixed-length and dense in nature.

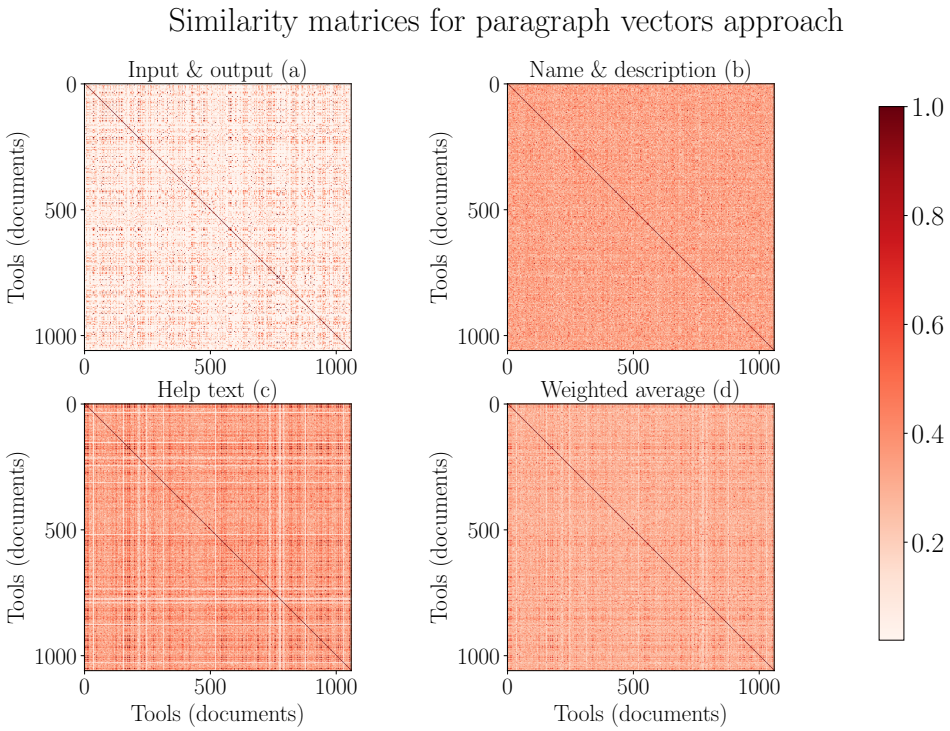

Figure 29.: Similarity matrices using paragraph vectors approach: The heatmap shows documents-documents (tools-tools) correlation matrices for input and output (a), name and description (b) and help text (c) attributes. The (d) shows a documents-documents (tools-tools) correlation matrix which is the weighted average computed using (a), (b) and (c) and weights (figure 22) given by the gradient descent optimizer (equation 15). The corresponding documents-tokens matrices are computed as shown in figure 28. The similarity matrices (b), (c) and (d) are denser compared to those from figure 23.

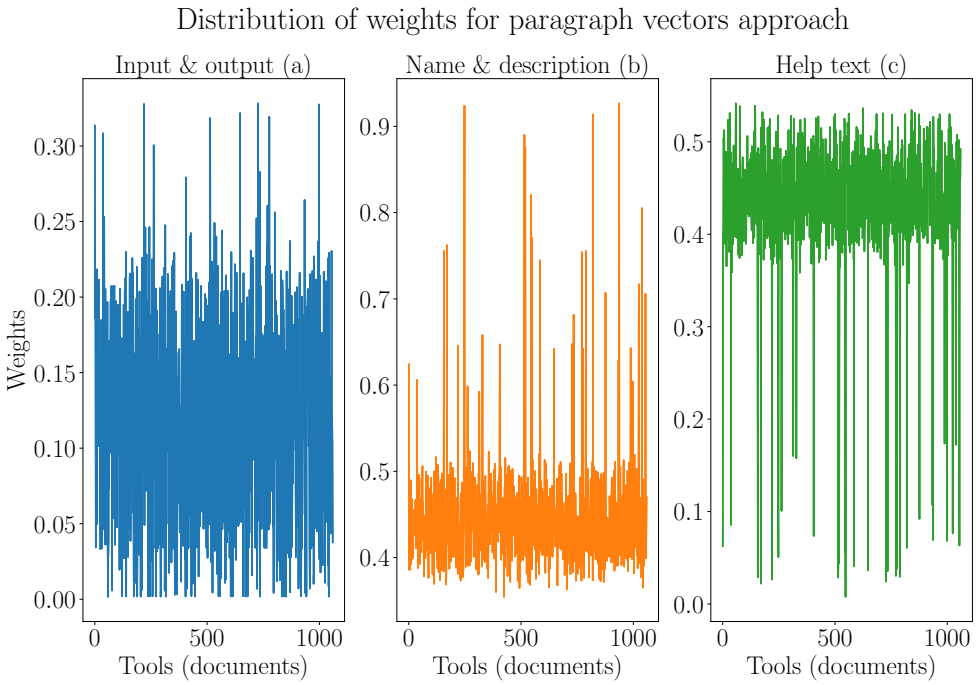

Figure 30.: Distribution of weights for similarity matrices computed using documents-tokens matrices for paragraph vectors approach: The plot shows the distribution of weights learned by gradient descent optimizer on the similarity matrices (29a, 29b and 29c) for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding documents-tokens matrices are computed as shown in figure 28.

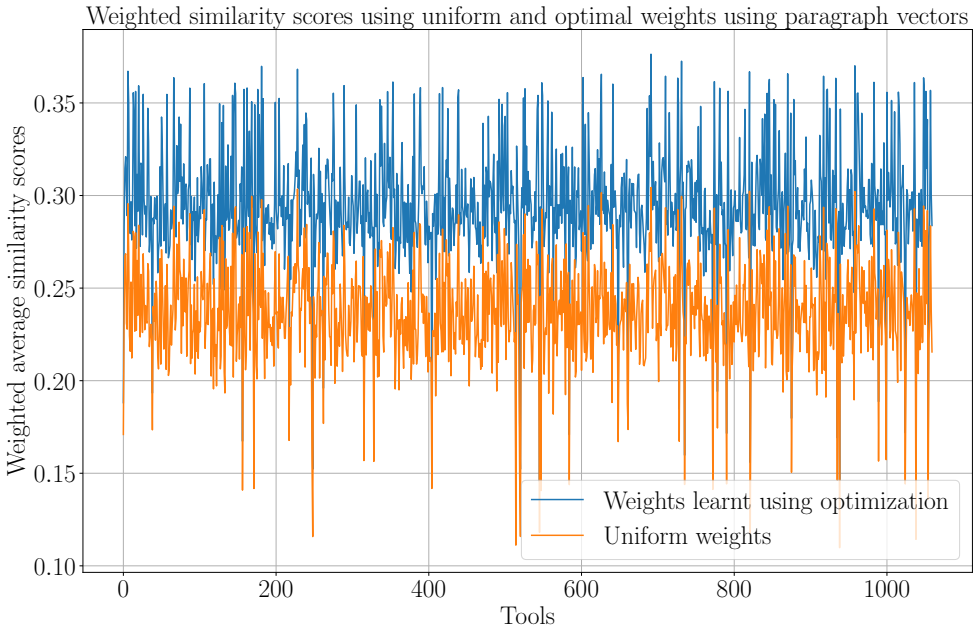

Figure 31.: Average of uniformly and optimally weighted similarity scores across all tools for paragraph vectors approach: The plot shows a comparison of average similarity scores computed using weights learned using optimization and uniform weights across all tools. We can see that the average similarity scores using optimal weights are higher than using uniform weights. We can conclude that the optimizer learns higher weights on those similarity scores which are higher in magnitude. It learns higher similarity scores compared to latent semantic analysis approach (figure 26 and 27).

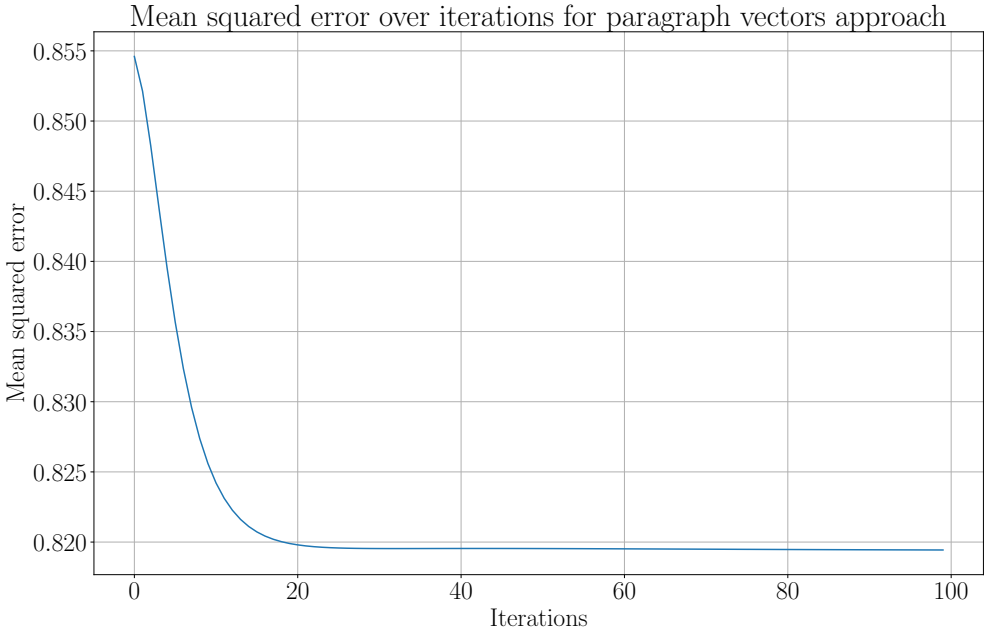

Figure 32.: Mean squared error across all tools over iterations for paragraph vectors approach: The plot shows a drop in the mean squared error across all the tools and attributes over gradient descent iterations. As we move along the iterations, the error drops and stabilizes at $\approx 30^{th}$ iteration.

Similar tools	input & output	name & desc.	help text	weighted similarity
hisat2	0.38	0.46	1	0.67
tophat	0.25	0.73	0.54	0.58

Table 4.: Similar tools (top-2) for "hisat" extracted using paragraph vectors approach: The table shows top-2 similar tools selected for "hisat". The weights learned by optimization are 0.14 (input and output file types), 0.44 (name and description) and 0.42 (help text).

5. Conclusion

5.1. Tools data

The text for help text attribute was noisy containing lots of words which were generic and provided little information to categorize tools. On the other hand, it was necessary as it supplied more information about them and their functions. Due to the noisy nature of help text data, it needed more filtering and we considered only the first 4 lines of text from this attribute. The data for the other attributes, input and output file types and name and description were helpful. The extraction of tools data from github's multiple repositories was slow[1] and due to which we read information from the xml files of tools into our local tabular file and performed our analysis using that tabular file.

5.2. Approaches

We investigated two approaches to find similarity in Galaxy tools. The latent semantic analysis approach which relied on matrix rank reduction in order to remove unimportant concepts or dimensions worked better than using full-rank documents-token matrices. However, we were not sure about the quantity of reduction of noise dimensions and because of this, we ran the risk of losing important dimensions while reducing the noise. During optimization, it gave more importance to input and output file types which were many times undesirable. But, in general, this approach is simple and takes less time ($\approx$ 350 seconds for $\approx$ 1050 tools) to complete. We can easily compute the low-rank estimations of sparse matrices using singular value decomposition.

The paragraph vectors approach worked better than the previous rank reduction technique in terms of finding similar tools in a robust way. The documents-tokens matrices it learned for name and description and help text were dense and encoded the semantic similarities among the documents. The documents which were similar in

[1]The extraction of $\approx$1050 tools took $\approx$ 30 minutes

context learned similar vectors as proved by their dense similarity matrices. However, this approach is slow as it takes $\approx$ 1000 seconds to finish, most of which is taken to learn document vectors. We trained for 10 epochs and each epoch had 800 iterations to learn vectors for each document. Running for 200 iterations over 10 epochs did not fetch relevant results. Moreover, to specify the number of dimensions of paragraph vectors was an important concern. From figure 5, we saw that the average number of tokens for name and description was $\approx$ 2-3 times higher than that of help text. So, the vector dimensions for name and description attribute was set to a smaller number (100).

The worst mean squared error is 1.0. The paragraph vectors approach dropped the mean squared error to ≈ 0.82 while the latent semantic analysis approach could drop it only to ≈ 0.92 by reducing the rank of documents-tokens matrices to 5%.

5.3. Optimization

Learning the weights on the similarity scores from multiple attributes worked in a good way and we reached the saturation point already before 50^{th} iteration. We can reduce the number of iterations for gradient descent (100 to $\approx$ 40). We used gradient descent optimizer with mean squared error as loss function. We used mean squared error because we desired the hypothesis similarity scores distribution to be as close to the true distribution (based on similarity measures) as possible. We decreased the risk of getting stuck at saddle points or local minima by using momentum with an accelerated gradient to update the weight parameters.

6. Future work

6.1. Get true similarity values

To quantify the improvements, we need to set absolute true values among tools. It means that we should create a dictionary of say 10 similar tools for each tool. Using this constraint, we can verify our findings in a more reliable way. Otherwise, we need to have a visualizer to look through the similar tools to verify whether we improve. On the other hand, it is not an easy task to create logical categories and have similarity scores for thousands of tools.

6.2. Correlation

Our similarity matrices are dense. We can nullify some low scores to retain only the high similarity scores. We can do it by finding the median value of similarity scores for a tool and set the scores to 0 which are lower than the median. After this, we can optimize the scores and see the distribution and results.

6.3. Other error functions

We used mean squared error as loss function for optimizing the weights parameters. We can try out other error measures like cross-entropy error. With this error, we need to redefine our true similarity value (as we took 1.0 as the best similarity score between a pair of tools and with cross-entropy error, one term would always be zero) and similarity measures. Based on these entities, we can compute the gradient of the error function with respect to the weights parameters.

6.4. More tools

We can increase the number of tools to do the analysis on. It would provide more data and consequently more context for the paragraph vectors to learn better semantics among the documents.

Part II.

Predict next tools in Galaxy workflows

7. Introduction

7.1. Galaxy workflows

A Galaxy workflow is a sequence of tools to process biological data. In this sequence, Galaxy tools are connected one after another forming a data processing pipeline. The linked tools in a workflow have compatible datatypes which means that the output datatype of one tool should be consumable by the next tool. A workflow can also be inferred as a directed acyclic graph. Figure 33 shows a workflow containing two paths. In general, a workflow can have multiple paths between its input and output tools. Each of these paths has a direction commencing from an input tool and ending at an output tool. Moreover, the paths in a workflow can have one or more tools in common. We can see in the figure 33 that tools like "Get flanks", "AWK" and "Intersect" are common in both the paths.

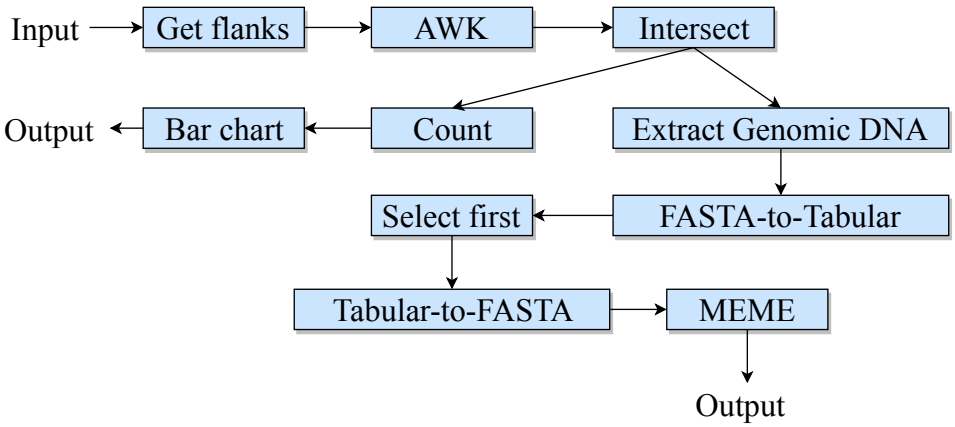

Figure 33.: A workflow: The image shows a workflow with two paths. It takes input data, processes it through multiple steps involving many tools in both the paths, a few tools being common, and gives two outputs.

While creating a workflow, at each stage, a user chooses a tool from clusters of tools and connect it to the previous tool. The user repeats this until a desired stage of output is reached. In each of these stages, a tool can connect to one or more tools

which are compatible to the previous tool. A user needs to decide which tool to connect next which depends on the multiple factors like the kind of input data, the nature of processing and so on.

8. Motivation

To create a workflow enclosing multiple paths where each path comprises of multiple tools is a complex task. The users need to keep knowledge of the tools that can come next. These next tools should be compatible to the previous one which is a necessary criterion for the tools to be connected on the canvas of Galaxy workflow editor. No two incompatible tools can connect to each other. This knowledge of compatibility is a must which can be gained only through experience. A user who is new to the Galaxy platform and is not aware of the existing tools, to create a workflow can be a labourious and time-consuming effort.

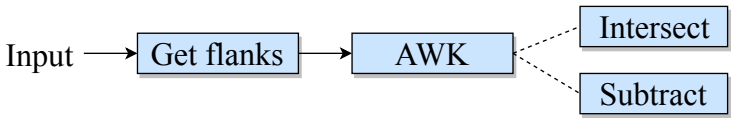

Figure 34.: Mulitple next tools: The image shows a part of a workflow where one tool can connect to more than one tool.

In figure 34, we can see that a tool "AWK" can connect to two tools, "Intersect" and "Subtract". The number of next tools can vary from tool to tool. Some tools can have just one next tool but others can have many different next tools they can connect to. To impart the knowledge of the next tools (for a tool or a sequence of tools) and assist a user at each stage of creating a workflow, we propose to build a system to predict next tools. It would recommend a set of next possible tools to the user whenever he/she chooses and connects a tool to a workflow. It would scale down the size of the collection of next tools to choose from and thereby we expect a reduction in the time taken to create workflows. Many users create and store their workflows on Freiburg Galaxy server. We intend to learn the patterns/features of tools connections present in these workflows after dividing them into long sequences of tools using machine learning. Using these patterns, we can predict the next possible tools.

9. Related work

The paths we extract from workflows are sequential where tools are connected one after another. In order to predict the next tools, we should take into account all the previous connected tools. As our data possesses long chains of tools, we explore some literature from machine learning research which work on similar data. Interestingly, learning from sequential data is a popular task in many other fields like natural language processing [14, 15]. They use deep learning models to learn sequences of words which aims to classify sentiments in text, learn part-of-speech tagging and unique dense vector for each word. The classification of sentiments involves learning core contexts present in the chain of words. The part-of-speech tagging which divides a sentence into multiple parts of speech like adjective, adverb, conjunctions also depends on finding context hidden in the long chains of words. For sentiment analysis [14] achieves an accuracy of $\approx 85\%$ using recurrent neural network (gated recurrent unit). For part-of-speech tagging, the accuracy goes upto $\approx 93\%$.

For clinical data as well, learning long sequences of data proves to be beneficial [4]. In this work, health states of patients recorded at different point of times are analysed by accessing their electronic health records. Using the learned model, the authors predict the future health states of a patient using the sequence of his/her health states in the past. The authors use long-short term memory (LSTM), a kind of recurrent neural network, to classify the sequences of health states. They achieve an accuracy of $\approx 85\%$ after applying necessary regularization techniques like dropout and weight normalization.

[16, 17] uses recurrent neural networks to model music and speech signals. The authors analyze the traditional recurrent and long-short term memory and gated recurrent units (GRU) and come to conclusion that the traditional recurrent units do not learn the semantics present in complete sequences while LSTM and GRU do. In this study, the authors learn sequences of 20 continuous samples (20 steps in time-series of speech) and predict the next 10 continuous samples (next 10 time steps). They also find out that GRU performs better than LSTM in terms of accuracy and

running time. For musedata[1], a collection of piano classical music, the performances (average of the negative log-likelihood) on test set noted by the authors are: GRU -5.99, LSTM - 6.23 and traditional recurrent units - 6.23. They test on 6 different types of musical and speech data and 4 out of these 6 types, GRU works better than the other two units.

These successful studies inspire us to avail benefits from the state-of-the-art sequential learning techniques like LSTM and GRU. They learn the tools connections inherent in the workflows to predict next possible tools at each time-step.

[1] www.musedata.org

10. Our approach

We discussed in section 7 that a workflow consists of multiple tools arranged one after another. To create a workflow, a user selects a tool from a set of tools. At each phase of tool selection to design a workflow, we propose to display a set of possible next tools. To compute this set of next tools, we need to design a learning algorithm that learns the connotations of tools connections. We designate this learning algorithm as a classifier. A classifier is an algorithm which indentifies the categories of input data. The complete data is divided into training and test sets. Each entry in the train or test set has a sample and its category. The classifier consumes the training set to learn the characteristics or features hidden in the data along with their categories. The robustness of learning by the classifier is verified on the test set. This is the basic principle followed in machine learning tasks. Moreover, there are two crucial features of this classifier - it should grasp the latent semantics in the tools connections and use them to predict next tools correctly (with an accuracy of $\geq 90\%$). Along side, the learning time should be reasonable. While developing our classifier, we take note of the classifier's running-time complexity and its precision to predict next tools.

10.1. Next tools

From figure 33, we conclude that a workflow can have more than one path and these paths can share few tools. Also, one tool or a sequence of tools can connect to more than one tool at any stage of workflow creation. Our classifier needs data in the form of sample and its category. For our workflows data, a sample is either a tool or a sequence of tools. The category is also a tool or a set of tools which forms a set of next possible tools for the sample (a tool or a sequence of tools). To delineate this idea of extracting samples and their categories, let's take an example workflow from figure 33. We decompose this workflow into following smaller sequences of tools taking a path from tool "Get flanks" until "Bar chart".

Using the technique explained in table 5, we decompose the workflows into samples and their respective categories. We do it in this way because of the necessity of

Samples (tools sequences)	Categories
Get flanks	AWK
Get flanks, AWK	Intersect
Get flanks, AWK, Intersect	Count, Extract genomic DNA
Get flanks, AWK, Intersect, Count	Bar chart

Table 5.: Workflow decomposition into samples and categories: This table shows multiple samples (tools sequences) and their categories computed using a workflow shown in figure 33. Each row in the table shows one sample with either one tool or a sequence of tools with its category (a tool or tools).

having prediction in the same way. It means that when we select "Get flanks" tool while creating a workflow, our classifier should sugges "AWK" as a next possible tool. Let's say we select "AWK" and our sequence becomes "Get flanks, AWK". Given this sequence, the classifier should predict "Intersect". When we select "Intersect" as the next tool, the classifier should predict two next tools - "Count" and "Extract genomic DNA" (as present in the workflow in figure 33). Following this approach, we compute samples and their categories. The categories that we assign are actual next tools which means that a tool or a sequence of tools is connected to these next tools (categories) in workflows. In the next section, we see that these categories are not the only tools that a tool or a sequence of tools can connect to.

10.2. Compatible next tools

The categories which we saw in the previous section we term them as actual categories. The tools in the workflows are connected because of their compatibility in their input and output file types. One tools cannot connect to all the other tools but only a handful. By extracting smaller tools sequences, we compute the actual next tools. But, there are other next tools which we should consider as well. The tool "Intersect" connects to multiple other tools given different or none previous tools in a sequence. For example, in a workflow - "Calling of methylated regions" > "Intersect" > "Differentially methylated region" > "Deeptools compute matrix" > "Deeptools heatmap", "Intersect" tool connects to "Differentially methylated region" tool. It means that these two tools are compatible as well in terms of their filetypes. There can be multiple other tools which can connect to "Intersect". We call these set of next tools as compatible tools. For each tool we assemble a set of compatible tools. The

significance of these tools is that even if our classifier predicts tools present in this set as a next tool for a tool sequence, it is still a valid next tool on the account of being compatible. It can be used as a next tool. The actual next tools take into account the same tool sequence while the compatible next tools consider only the last tool in the tools sequence to find tools which are valid next tool. For example, "Intersect" tool can connect to these tools - "Subtract", "Convert", "Group", "Differentially methylated region" and many others. These form the compatible set of next tools for "Intersect".

10.3. Variation of number of compatible tools

10.4. Topk predictions - absolute precision, top3, top5

10.5. Encoding the compatible types into the label vector

10.6. Compatible types just for verification

10.7. Length of input sequences

10.8. Effect of learning rates

10.9. Loss function

10.10. Optimizer

10.11. Classifiers - Neural network and classical ones

11. Experiments

12. Results and analysis

12.1. No decomposition of paths

12.2. Decomposition keeping first one fixed

12.3. Decomposition for each pair of tools

13. Conclusion

14. Future work

Bibliography

[1] P. W. Foltz, "Latent semantic analysis for text-based research," *Behavior Research Methods, Instruments, & Computers*, vol. 28, pp. 197–202, Jun 1996.

[2] Q. V. Le and T. Mikolov, "Distributed representations of sentences and documents," *CoRR*, vol. abs/1405.4053, 2014.

[3] Z. Kaoudi, J. Quiané-Ruiz, S. Thirumuruganathan, S. Chawla, and D. Agrawal, "A cost-based optimizer for gradient descent optimization," *CoRR*, vol. abs/1703.09193, 2017.

[4] Z. C. Lipton, D. C. Kale, C. Elkan, and R. C. Wetzel, "Learning to diagnose with LSTM recurrent neural networks," *CoRR*, vol. abs/1511.03677, 2015.

[5] H. Sak, A. W. Senior, and F. Beaufays, "Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition," *CoRR*, vol. abs/1402.1128, 2014.

[6] S. Robertson and H. Zaragoza, "The probabilistic relevance framework: Bm25 and beyond," *Found. Trends Inf. Retr.*, vol. 3, pp. 333–389, Apr. 2009.

[7] A. M. Shapiro and D. S. McNamara, "The use of latent semantic analysis as a tool for the quantitative assessment of understanding and knowledge," *Journal of Educational Computing Research*, vol. 22, no. 1, pp. 1–36, 2000.

[8] T. K. Landauer, "Learning and representing verbal meaning: The latent semantic analysis theory," *Current Directions in Psychological Science*, vol. 7, no. 5, pp. 161–164, 1998.

[9] J. Yang, "Notes on low-rank matrix factorization," *CoRR*, vol. abs/1507.00333, 2015.

[10] G. I. Ivchenko and S. A. Honov, "On the jaccard similarity test," *Journal of Mathematical Sciences*, vol. 88, pp. 789–794, Mar 1998.

[11] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.

[12] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," pp. III–1139–III–1147, 2013.

[13] A. Botev, G. Lever, and D. Barber, "Nesterov's accelerated gradient and momentum as approximations to regularised update descent," pp. pp. 1899–1903., 2017.

[14] W. Yin, K. Kann, M. Yu, and H. Schütze, "Comparative study of CNN and RNN for natural language processing," *CoRR*, vol. abs/1702.01923, 2017.

[15] X. Li, T. Qin, J. Yang, and T. Liu, "Lightrnn: Memory and computation-efficient recurrent neural networks," *CoRR*, vol. abs/1610.09893, 2016.

[16] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *CoRR*, vol. abs/1412.3555, 2014.

[17] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription," 2012.