

Master Thesis

Recommendation system for scientific tools and workflows

Anup Kumar

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

University of Freiburg
Department of Computer Science
Bioinformatics Group Freiburg

July 9, 2018

Thesis period

08.01.2018 – 09.07.2018

Examiners

Prof. Dr. Rolf Backofen and Prof. Dr. Wolfgang 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

Dedicated to my mom and dad

Acknowledgement

I would like to offer 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 the Galaxy. He facilitated this work by providing me with all 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 visualiser 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 visualiser for looking through the next predicted tools was conducive in all merits. I offer regards and thanks to Dr. Ulrike Wagner-Höher and Kai Höher (MSc.) for the German translation of the abstract. I offer thanks to Dr. Mehmet Tekman and Joachim Wolff for their expert feedback, insights and general advice. I appreciate and thank Helena Rasche for extracting the workflows. I thank Andrea Bagnacani for the intuitive discussions. At length, I wish to thank all other members, Dr. Anika Erxleben and Dr. Bérénice Batut, of the Freiburg Galaxy team for their continued support and help. I appreciate and thank Dr. Anika Erxleben, Dr. Björn Grüning and Tonmoy Saikia for proofreading the thesis.

Abstract

The thesis researches two concepts to develop a recommendation system for the Galaxy’s scientific data-processing tools and workflows. The concepts incorporate finding similarities among tools and predicting next tools in workflows. They are dealt separately in the thesis. The first part explores a way to find similarities among tools. It implies which similar tools exist for each tool within a collection of tools. In addition, it quantifies the similarities among tools by assigning a similarity score for each pair of tools. Similarities among tools are computed using approaches from natural language processing and optimisation. Over 1,000 tools are used for this task. The metadata of tools is gathered from multiple attributes including name, description, input and output data types and help text. It is hard to quote accuracy for this approach due to the lack of true similarity values. Visualisers are created to see similar tools. The second part formulates a prediction system to display next possible tools in scientific workflows. These scientific workflows are complex and are created using more than 4,000 tools in the bioinformatics field. The knowledge of next possible tools can make it easier for the less experienced Galaxy users to create them. Moreover, it can curtail the amount of time taken to create a workflow. The unique paths (tool sequences) extracted from workflows are fed to the recurrent neural networks to learn the semantics of tool connections. The predictions are made by learning higher-order dependencies prevalent in these tool connections. More than 167,000 paths are used to learn the predictive model. The absolute classification precision of $\approx 89\%$ and compatible classification precision of $\approx 99\%$ are achieved.

Zusammenfassung

Die Arbeit erforscht zwei Konzepte, um ein Empfehlungssystem für Galaxy's wissenschaftliche Datenverarbeitungs-Werkzeuge/Tools und Workflows zu entwickeln. Die Konzepte werden in dieser Arbeit separat behandelt und beinhalten das Auffinden von Ähnlichkeiten bei Werkzeugen und die Vorhersage nachfolgender Werkzeuge in Workflows. Der erste Teil erforscht die Methode zum Auffinden von Ähnlichkeiten bei Tools innerhalb einer Gruppe von Tools. Darüber hinaus wird die Ähnlichkeit zwischen Tools paarweise durch die Verwendung eines Punktesystems quantifiziert. Zur Berechnung der Ähnlichkeit von Tools wird dabei die natürliche Sprachverarbeitung und Optimierung berücksichtigt. Für diese Aufgabe wurden über 1.000 Tools verwendet. Die Daten der Tools wurden aus verschiedenen Merkmalen, welche Name, Beschreibung, Input- und Output-Typen und Hilfstexte umfassten, gewonnen. Es ist schwierig, für diesen Ansatz Genauigkeit anzugeben, da echte Ähnlichkeitswerte fehlen. Visualisierer werden erstellt, um ähnliche Tools zu sehen. Der zweite Teil entwirft ein Vorhersage-System, welches die nächsten möglichen Tools bei Workflows darlegt. Diese wissenschaftlichen Workflows sind komplex und werden mit mehr als 4.000 Tools in der Bioinformatik erstellt. Die Kenntnis möglicher nächster Tools kann es wenig erfahrenen Galaxy-Benutzern leichter machen, Workflows zu erstellen. Außerdem kann die zur Schaffung eines Workflow benötigte Zeitdauer verkürzt werden. Die einzelnen, aus Workflows entnommenen Pfade (Tool-Sequenzen) werden in rückgekoppelte neuronale Netze eingebracht, um die Bedeutung von Tool-Verbindungen zu ermitteln. Die Vorhersagen wurden durch das Lernen vorherrschender Abhängigkeiten höherer Ordnung der Tool-Verbindungen gewonnen. Zur Erstellung des Vorhersage-Modells wurden über 167.000 Pfade verwendet. Die absolute Klassifikationsgenauigkeit von $\approx 89\%$ und die kompatible Klassifikationsgenauigkeit von $\approx 99\%$ werden erreicht.

Contents

I	Find similar scientific tools	1
1	Introduction	2
1.1	Galaxy	2
1.2	Scientific tools	3
1.3	Motivation	4
2	Approach	5
2.1	Metadata of tools	5
2.1.1	Attributes	5
2.1.2	Useful metadata	7
2.2	Dense vector for a document	11
2.2.1	Latent semantic analysis	11
2.2.2	Paragraph vectors	15
2.3	Similarity measures	18
2.3.1	Cosine similarity	18
2.3.2	Jaccard index	18
2.4	Optimisation	19
2.4.1	Gradient descent	20
2.4.2	Learning rate decay	21
2.4.3	Weight update	22
3	Experiments and results	25
3.1	Latent semantic analysis	25
3.1.1	Full-rank document-token matrices	26
3.1.2	5% of full-rank	29
3.2	Paragraph vectors	32
3.3	Comparison of two approaches	35
4	Conclusion	38
4.1	Metadata of tools	38
4.2	Approaches	38
4.2.1	Latent semantic analysis	38

4.2.2	Paragraph vectors	39
4.3	Optimisation	40
5	Future work	41
5.1	Get true similarity values	41
5.2	Exclude low similarity scores	41
5.3	Formulate different error functions	41
5.4	Acquire more tools	42
5.5	Learn tool similarity using workflows	42
II	Predict next tools in scientific workflows	43
6	Introduction	44
6.1	Galaxy workflows	44
6.1.1	Motivation	45
7	Related work	46
8	Approach	48
8.1	Actual next tools	49
8.2	Compatible next tools	50
8.3	Length of workflow paths	51
8.4	Learning	52
8.4.1	Workflow paths	52
8.4.2	Bayesian networks	54
8.4.3	Recurrent neural networks	55
8.5	Network architecture	59
8.5.1	Embedding layer	59
8.5.2	Recurrent layer	59
8.5.3	Output layer	59
8.5.4	Activations	60
8.5.5	Regularisation	60
8.5.6	Optimiser	61
8.5.7	Precision	63
8.6	Prediction pattern	63

9	Experiments	64
9.1	Decomposition of paths	64
9.2	Dictionary of tools	64
9.3	Padding with zeros	65
9.4	Network configuration	66
9.4.1	Mini-batch learning	67
9.4.2	Dropout	67
9.4.3	Optimiser	67
9.4.4	Learning rate	68
9.4.5	Activations	68
9.4.6	Number of recurrent units	68
9.4.7	Dimension of embedding layer	68
10	Results and analysis	69
10.1	Notes on plots	70
10.2	Performances of different approaches of path decomposition	71
10.2.1	Decomposition of only test paths	71
10.2.2	No decomposition of paths	71
10.2.3	Decomposition of the train and test paths	72
10.3	Performance evaluation on different parameters	73
10.3.1	Optimiser	74
10.3.2	Learning rate	76
10.3.3	Activation	76
10.3.4	Batch size	76
10.3.5	Number of recurrent units	77
10.3.6	Dropout	78
10.3.7	Dimension of embedding layer	79
10.3.8	Accuracy (top-1 and top-2)	80
10.3.9	Precision with the length of paths	82
10.3.10	Performance with a small number of workflows	83
10.3.11	Neural network with dense layers	84
11	Conclusion	87
11.1	Network configuration	87
11.2	The amount of data	87
11.3	Decomposition of paths	88

11.4 Classification	88
12 Future work	89
12.1 Train on longer and test on shorter paths	89
12.2 Restore original distribution	89
12.3 Use convolution	89
12.4 Use other classifiers	89
12.5 Decay prediction based on time	90
Bibliography	90
A Appendix	95
A.1 Visualiser	95
A.2 Code repository	95
A.2.1 Find similar scientific tools	95
A.2.2 Predict next tools in scientific workflows	96

List of Figures

1	Basic flow of dataset transformation	2
2	Common features of two tools	3
3	Similarity graph	4
4	Sequence of steps to find similar tools	6
5	Distribution of tokens	8
6	Tool, document and tokens	9
7	Singular value decomposition	12
8	Singular values of document-token matrices	14
9	Variation of the fraction of ranks of document-token matrices with the fraction of sum of singular values	15
10	Distributed memory	17
11	Distributed bag-of-words	17
12	Decay of learning rate for gradient descent optimiser	22
13	Difference between actual and approximated gradients	24
14	Similarity matrices computed using full-rank document-token matrices	27
15	Distribution of weights learned for similarity matrices computed using full-rank document-token matrices	28
16	Average of weighted similarities computed using uniform and optimal weights	29
17	Similarity matrices computed using document-tokens matrices reduced to 5% of their full-ranks	30
18	Distribution of weights learned for similarity matrices computed using document-token matrices reduced to 5% of their full-ranks	31
19	Average of weighted similarities computed using uniform and optimal weights	32
20	Similarity matrices using paragraph vectors approach	33

21	Distribution of weights for similarity matrices computed using paragraph vectors approach	34
22	Average of weighted similarities computed using optimal and uniform weights	35
23	A workflow	44
24	Multiple next tools	45
25	Sequence of steps to predict next tools in workflows	49
26	Distribution of the number of compatible next tools	51
27	Distribution of sizes of workflow paths	52
28	No path decomposition	53
29	Path decomposition	54
30	Higher-order dependency	55
31	Recurrent neural network	57
32	Gated recurrent unit	58
33	Predicted scores for next tools	63
34	Vectors of a path and its next tools	66
35	Performance of the decomposition of only test paths	72
36	Performance of no decomposition of paths	73
37	Performance of the decomposition of all paths	74
38	Performance of different optimisers	75
39	Performance of multiple learning rates	77
40	Performance of multiple activation functions	78
41	Performance of different batch sizes	79
42	Performance of multiple values of memory units	80
43	Performance of multiple values of dropout	81
44	Performance of different dimensions of embedding layer	82
45	Absolute and compatible top-1 and top-2 accuracies	83
46	Variation of precision with the length of paths	84
47	Performance with a small number of workflows	85
48	Performance of a neural network with dense layers	86

List of Tables

1	A sparse document-token matrix	11
2	Similar tools (top-2) for <i>hisat</i> computed using full-rank document-token matrices	36
3	Similar tools (top-2) for <i>hisat</i> computed using document-token matrices reduced to 5% of full-rank	36
4	Similar tools (top-2) for <i>hisat</i> computed using paragraph vectors approach	37
5	Decomposition of a workflow	49

Part I.

Find similar scientific tools

1. Introduction

1.1. Galaxy

Galaxy is an open-source biological data processing and research platform [1]. It supports numerous types of data formats like *fasta*, *fastq*, *gff*, *vcf* and many more¹ and they are extensively used for data processing. It offers scientific tools and workflows to transform these datasets (figure 1). Each tool has an exclusive way to process datasets. An example of data processing is to convert a sequence of nucleotides² into its reverse-complement counterpart.

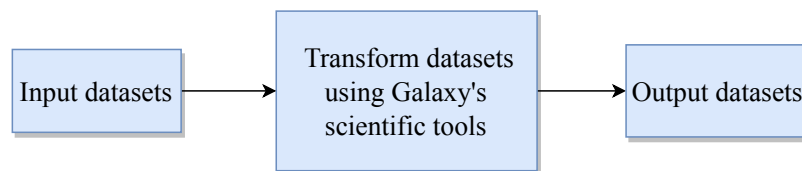


Figure 1.: Dataset transformation: The image shows a general flow of data transformation using the Galaxy’s scientific tools.

Tools are classified into multiple categories based on their functions and types. For example, the tools which manipulate text, like replacing texts and selecting lines of a dataset, are grouped together under *text manipulation*. Similarly, there are various tool categories like *imaging*, *convert formats*, *genome annotation* and many others³. These tools are the building blocks of a workflow. A workflow is a data processing pipeline where a set of tools are connected one after another. The connected tools in a workflow should 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. An example of a workflow is *variantcalling-freebayes*. It is used for the detection of variants (specifically single and multiple nucleotide polymorphisms and insertions and deletions) following a bayesian approach.

¹<https://galaxyproject.org/learn/datatypes>

²https://usegalaxy.eu/?tool_id=MAF_Reverse_Complement_1&version=1.0.1

³<https://toolshed.g2.bx.psu.edu/repository>

1.2. Scientific tools

A tool entails a specific function. It consumes a dataset, brings about some transformations and produces an output dataset which is fed to other tools. For example, the tool *trimmomatic* trims a *fastq* file and produces a *fastqsanger* file which is used as an input file for a different tool like *fastqc*. A tool has multiple attributes which include its input and output file types, name, description, help text and many more⁴. These attributes constitute the metadata of tools. The metadata shows that some tools have similar functions while some share similarities in their input and output file types. For example, the tool *hicexplorer_hicpca*⁵ has an output type named *bigwig*. A tool which also has *bigwig* as its input and/or output type, there is some similarity between these tools as they do a transformation on a similar file type. Similar tools can also be found by analysing other attributes like name and description.

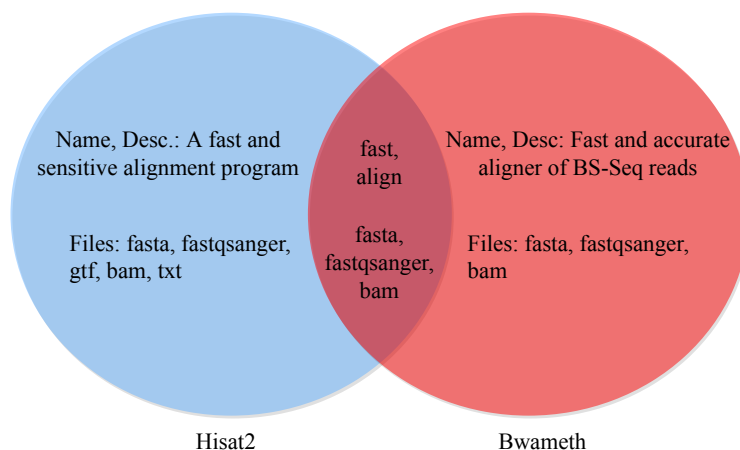


Figure 2.: Common features of two tools: The venn diagram shows common features of two tools - *hisat2* and *bwameth*. Few attributes like name, description and file types (input and output) are used to extract common features. Based on the common features, shown in the middle of the venn diagram, similarity between them can be assessed.

Figure 2 shows two tools - *hisat2* and *bwameth*. Their respective metadata is collected from their input and output file types and name and description attributes. Both of them have common file types (*fasta*, *fastqsanger*, *bam*). Moreover, they share

⁴<https://docs.galaxyproject.org/en/master/dev/schema.html>

⁵https://usegalaxy.eu/?tool_id=toolshed.g2.bx.psu.edu/repos/bgruening/hicexplorer_hicpca/hicexplorer_hicpca/2.1.0&version=2.1.0

a similar function of aligning. By extrapolating this way of finding similar features among tools, a set of similar tools for each tool can be created.

1.3. Motivation

The Galaxy has thousands of tools with a diverse set of functions. New tools keep getting added to the existing set of tools. For a user, it is hard to have knowledge about so many existing tools. In addition, it is important to make users aware of the presence of the newly added tools. They may be similar to some of the existing tools. A set of similar tools for a tool would give more options to the users for their data processing.

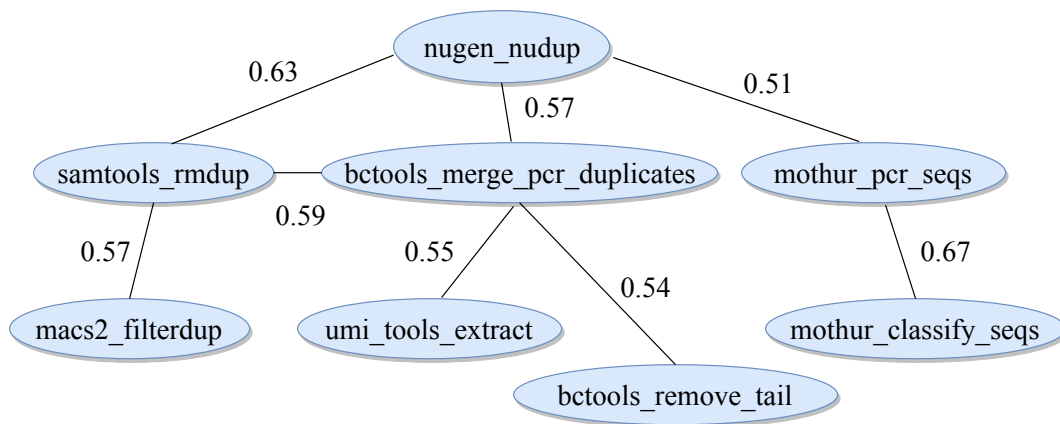


Figure 3.: Similarity graph of tools: In the graph, the nodes are represented by tools and the edges show similarity scores for each pair of tools. The higher the similarity score, the more similar a pair of tools is.

To elaborate more, a tool *nugen_nudup*⁶ (figure 3) is taken. It finds and removes *pcr* duplicates. A few similar tools are collected. Tools like *samtools_rmdup* and *bctools_merge_pcr_duplicates* also have a similar function. Each tool has a set of similar tools and together they make a graph of related tools. This similarity graph shows *connectedness* among tools and can help a user to find multiple ways to process data by replacing a tool with a similar tool. Figure 3 shows a small example of the similarity graph.

⁶https://toolshed.g2.bx.psu.edu/repository?repository_id=4f614394b93677e3

2. Approach

This section gives a comprehensive description of an approach to compute similarities among tools. It involves a sequence of steps (figure 4). The Galaxy’s codebase and tool repositories are at *github*. The metadata of tools is extracted from the *github*’s tool repositories. It is cleaned to get a set of words for each tool which uniquely identifies it. This set is used to create a fixed-length vector for each tool. The words from all the tools are merged to create a collection. Its size gives the size of the vector. Each word from the collection has a position in the vector. In the vector of a tool, the positions of the respective words (words from the tool’s set) contain the frequencies of their occurrence. The positions of those words not present in the set for a tool are equal to zero. These vectors are used to compute similarity scores for each pair of tools using similarity measures. The similarity scores of each tool with all the other tools create a similarity matrix. A similarity matrix is computed for each attribute. These similarity matrices are combined by optimisation to get a weighted average similarity matrix. All steps to compute similarities among tools are explained in further sections.

2.1. Metadata of tools

One of the Galaxy’s tool repositories is *Galaxy tools maintained by iuc*¹. A tool is defined in an *extensible markup language (xml)* file which can be parsed efficiently to gather the metadata from multiple attributes. The *xml* file of a tool starts with a *tool* tag.

2.1.1. Attributes

A tool has multiple attributes which include input and output file types, help text, name, description, citations and many more². But, not all of these attributes are

¹<https://github.com/galaxyproject/tools-iuc>

²<https://docs.galaxyproject.org/en/master/dev/schema.html>

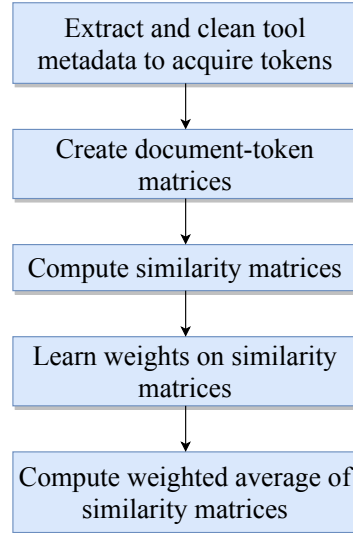


Figure 4.: Sequence of steps to find similar tools: The flowchart shows a series of steps to establish similarities among tools using the approaches from natural language processing to compute similarity matrices and optimisation to compute a weighted average of the similarity matrices.

significant in identifying a tool. Therefore, only the following attributes are considered to collect the metadata of tools:

- Input file types
- Output file types
- Name
- Description
- Help text

Moreover, the input and output file types are combined by taking a union set to create one attribute as together they contain information about file types of a tool. A similar combination is done for the name and description attributes as well. These combined attributes give a complete metadata of a tool's file types (input and output) and its functionality (name and description). Further, help text attribute is also included in the metadata. This attribute contains more information compared to the previous two combined attributes. Apart from being larger in size, it is noisy as well. It gives a detailed information about the functions of a tool and the format of

its input data³. Much of the information collected by this attribute is not important to clearly distinguish a tool. Therefore, only the first few lines of text from this attribute are considered which illustrate the core functionality of tools. The rest of the information is discarded.

2.1.2. Useful metadata

Duplicates and stop-words removal

The metadata of tools collected from multiple attributes is raw. It contains lots of noisy and duplicate items which do not add any value. These items should be removed from the metadata to retain items which are unique and useful. For example, the tool *bamleftalign* has *bam* and *fasta* as input files and *bam* as an output file. While combining these file types to make a set of file types for a tool, the duplicates are discarded. Hence, *bam* and *fasta* together make a set of file types for the tool. This set does not maintain any order of the file types. The attributes like name, description and help text contain sentences (complete or partially complete) in the English language. Therefore, to process these, strategies from natural language processing⁴ are needed. A sentence contains many words and can have many different parts of speech. The parts of speech include a subject, object, preposition, interjection, verb, adjective, adverb, article and many more⁵. Only those words are retained from the sentences which categorise a tool uniquely. For example, the tool *tophat* has "*tophat for illumina find splice junctions using rna-seq data*" as its combined name and description attribute. The words like *for*, *using* and *data* cannot distinguish the tool as they can be present in the description of many other tools. These words are called stop-words⁶ and are discarded. In addition, numbers are also removed. All the remaining words are converted to lower case.

Stemming

After removing the duplicates and stop-words, the metadata becomes clean and contains words which can identify tools uniquely. Different forms of a word are used in sentences due to the rules of grammar. For example, a word *regress* has

³https://planemo.readthedocs.io/en/latest/standards/docs/best_practices/tool_xml.html#help-tag

⁴<https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3168328>

⁵<https://web.stanford.edu/~jurafsky/slp3/10.pdf>

⁶<https://www.ranks.nl/stopwords>

multiple forms like *regresses*, *regression* and *regressed*. All these forms share the same root and point towards the same concept. Therefore, it is beneficial to converge all different forms of a word to one basic form. This process is called stemming⁷. The *nltk*⁸ package is used for stemming. It reduces the size of metadata and keeps the meaning of a word same across its multiple forms. After stemming, the words are called tokens. Figure 5 shows a distribution of tokens in the metadata for all three attributes of tools. These tokens are used for computing similarities among tools. The help text attribute contains more tokens than input and output file types and name and description attributes.

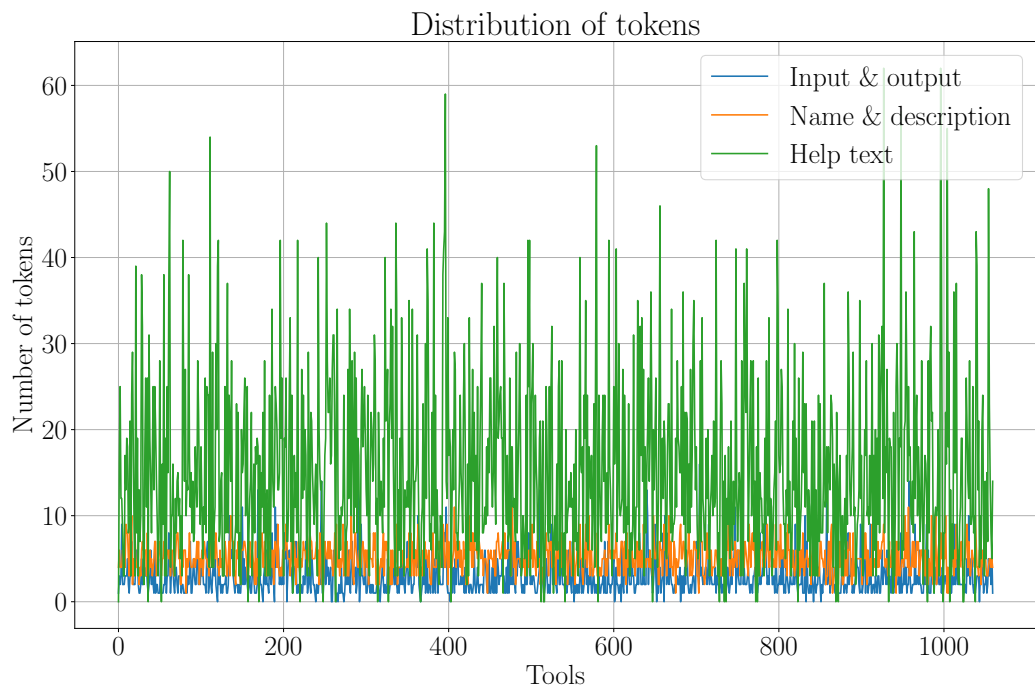


Figure 5.: Distribution of tokens: The plot shows a distribution of the number of tokens for input and output file types, name and description and help text attributes of tools.

⁷<https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html>

⁸<http://www.nltk.org>

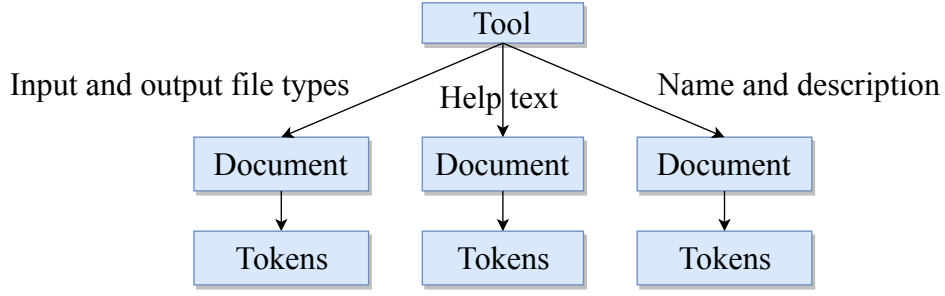


Figure 6.: Relationship between a tool, its documents and their tokens:

The image shows that a tool has three documents corresponding to each attribute and each document contains tokens. The documents are equal to the number of tools for each attribute. The number of tokens in each document varies.

Relevance factors for tokens

After stemming the words from metadata, meaningful sets of tokens are collected for input and output file types, name and description and help text for each tool. These sets are called documents (figure 6). The tokens present in these documents do not carry equal importance. Some tokens are more relevant to a document, but some are not. An importance factor is computed for each token in a document. These factors are arranged in a big, sparse document-token matrix. Each attribute has its own document-token matrix. In these matrices, each row represents a document and each column represents a token. To compute these importance factors, the *bestmatch25* (*bm25*) [2] algorithm is used. The variables used in implementing this algorithm are as follows:

- Token frequency⁹ (tf)
- Document frequency (df) and inverted document frequency (idf)
- Average document length ($|D|_{avg}$)
- Number of documents (N)
- Size of a document ($|D|$)

Token frequency (tf) gives the count of a token's occurrence in a document. If a token *regress* appears twice in a document, its tf is 2. It is a weight given to the token. Inverted document frequency (idf) for a token is defined as:

⁹<https://nlp.stanford.edu/IR-book/pdf/06vect.pdf>

$$idf = \log \frac{N}{df} \quad (1)$$

where df is the count of documents in which the token is present and N is the total number of documents. If a document is randomly sampled from a set of documents, the probability of the token to be present in this document is $p_i = \frac{df}{N}$. From information theory [3], the information contained by this event is computed by $-\log p_i$. Idf is higher when a token appears only in a few documents. It means that the token is a good candidate for representing those documents. A token which appears in many documents is not a good representative of those documents. $|D|_{avg}$ is the average number of tokens computed over all documents. $|D|$ is the number of tokens for a document. $bm25$ score of a token is calculated using the following equations:

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

$$tf^* = tf \cdot \frac{k + 1}{k \cdot \alpha + tf} \quad (3)$$

$$bm25_{score} = tf^* \cdot idf \quad (4)$$

where k and b are the hyperparameters of the $bm25$ algorithm and their ideal values should be found according to the data. k is always a positive number. When k is zero, the $bm25$ score for a token is defined only by its idf ($tf^* = 1$). When k is much larger than tf ($k \gg tf$), the $bm25$ score is computed using the raw term frequency ($tf^* \approx tf$). b is a real number between zero and one ($0 \leq b \leq 1$). When b is zero, the document length is not normalised. It means that the fraction $\frac{|D|}{|D|_{avg}}$ is not accounted for computing the $bm25$ score. When b is one, this fraction is completely accounted for. In this case, when $|D| \gg |D|_{avg}$ for any document, its tokens would get a lower $bm25$ score. For this work, the standard values of k and b (1.75 and 0.75, respectively) are used. Using equation 4, the $bm25$ score is computed for each token in all the documents. Table 1 shows the $bm25$ scores for four documents (rows) along with five tokens (columns). Following this approach, document-token matrices are computed for all three attributes of tools. For input and output file types, the matrix will have only two values, one if a token is present for a document and zero if not. For other attributes, the $bm25$ scores are positive real numbers.

Document/Token	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 document-token matrix: The table shows a matrix of documents arranged along the rows and tokens along the columns. Each value in the matrix is a *bm25* score assigned to a token. The matrix is sparse because the total number of tokens (for all the documents) is larger than the number of tokens present for each document.

The document-token matrices are sparse. Each entry in these matrices is a *bm25* score for each token. This representation is important to know which tokens are better representatives, but which are not for a document. A token is important for a document if the *bm25* score is high. However, it does not give any information about its relation to the other tokens. The information about the co-occurrence of tokens in a document defines a concept hidden in that document. A concept in a document is formed by using the relation among a few tokens. To illustrate this idea, an example of three words, *relabel list identifiers*, is considered. These three words mean little and point to different things if each word is considered separately. But, together they point towards a concept. The *bm25* model lacks the ability to find the correlation among tokens. To learn hidden concepts within documents and correlation among tokens, two approaches are explored:

- *Latent semantic analysis*¹⁰
- *Paragraph vectors*

Using these approaches, a multi-dimensional dense vector is learned for each document.

2.2. Dense vector for a document

2.2.1. Latent semantic analysis

A concept is a relationship among few tokens. *Latent semantic analysis* is a mathematical way to learn these hidden concepts in documents. It is achieved by computing

¹⁰<http://lsa.colorado.edu/papers/dp1.LSAintro.pdf>

a low-rank representation of a document-token matrix [4, 5, 6]. *Singular value decomposition (svd)* is used to achieve it. This decomposition follows the equation:

$$X_{n \times m} = U_{n \times n} \cdot S_{n \times m} \cdot V_{m \times m}^T \quad (5)$$

where n is the number of documents and m is the number of tokens. S is a diagonal matrix containing singular values in descending order and has the same shape as the document-token matrix X . The singular values are real numbers. It contains the weights of concepts present in the document-token matrix X . The matrices U and V are orthogonal matrices and satisfy:

$$U^T \cdot U = I_{n \times n} \quad (6)$$

$$V^T \cdot V = I_{m \times m} \quad (7)$$

where I is an identity matrix. Mathematically, singular values¹¹ from matrix S are the square roots of the eigenvalues of $X^T \cdot X$ or $X \cdot X^T$. The eigenvectors of $X \cdot X^T$ are the columns of matrix U and the eigenvectors of $X^T \cdot X$ are the columns of matrix V .

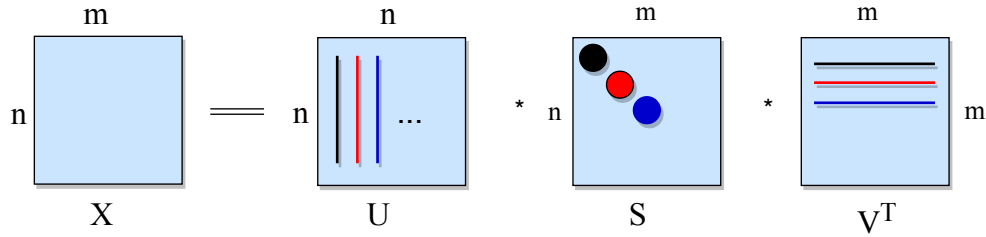


Figure 7.: Singular value decomposition: The image shows how a matrix is decomposed using *singular value decomposition (svd)*. The matrix X is factorised into three matrices - U , S and V using equation 5. These factor matrices (U , S and V) are used for the low-rank estimation of matrix X .

Figure 7 explains how the *svd* of a matrix is computed (inspired from a lecture¹²). Matrix U contains information about how the tokens, arranged along the columns, are mapped to concepts. Matrix V stores information about how concepts are mapped to documents which are arranged along the rows.

¹¹http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm

¹²<http://theory.stanford.edu/~tim/s15/l/19.pdf>

Low-rank approximation

A document-token matrix suffers from sparsity and shows no relation among tokens. A low-rank approximation of this matrix discards the features with small scaling factors to deal with issues of sparsity and correlation among tokens. The features with small singular values (the last entries of the matrix S along the diagonal in equation 5) represent noise and are discarded. The features with large singular values (the top entries of the matrix S along the diagonal in equation 5) are retained. A low-rank approximation X_k ($k < m$) is computed using equation 8. The resulting low-rank matrix is dense [7]. The size of the reconstructed matrix X remains the same but its rank reduces to k .

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

where U_k is the first k columns of U , V_k is the first k rows of V and S_k is the first k singular values in S . X_k is called as the rank- k approximation of full-rank matrix X . Figure 9 shows how the fraction of the sum of singular values changes with the fraction of the ranks of document-token matrices. If the ranks of matrices are reduced to 70% of corresponding full-ranks, $\approx 90\%$ of the sum of singular values can be captured. The reduction to half of the full-ranks achieves $\approx 80\%$ of the sum of singular values. This variation is shown for the document-token matrix of input and output file types attribute in figures 8 and 9 but its rank is not reduced.

The ranks of the document-token matrices for name and description and help text attributes are reduced 5% of their corresponding full-ranks. It gives dense, low-rank approximations of these document-token matrices. In these low-rank matrices, the dense vector representations for the documents are shown along the rows and the tokens are arranged along the columns. The similarity (correlation or distance) is computed using similarity measures for each pair of document vectors. There are many similarity measures which can be used like *euclidean distance*, *cosine similarity*, *manhattan distance* or *jaccard index*. In this work, *cosine similarity* is used for name and description and help text attributes and *jaccard index* for input and output file types attribute to compute the similarity between each pair of document vectors. A real number between zero and one is assigned as the similarity score for a pair of documents. The higher the score, the higher is the similarity between a pair of documents. Computing this similarity for all the documents gives a similarity matrix $SM_{n \times n}$ where n is the number of documents. It is a symmetric matrix (also called

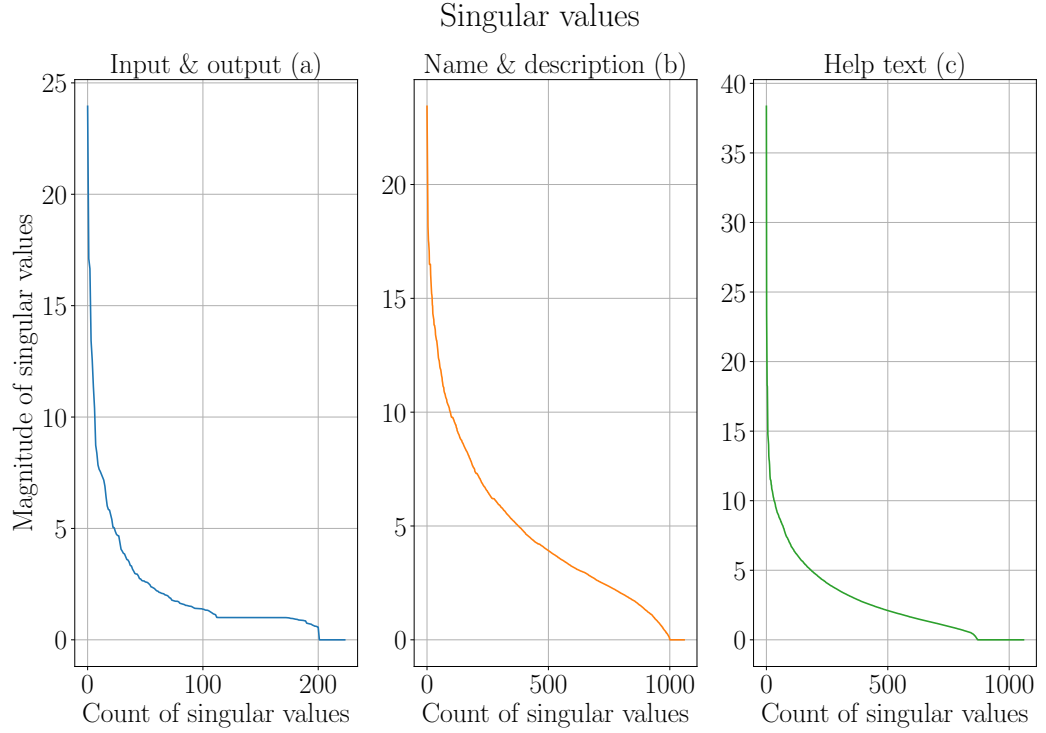


Figure 8.: Singular values of document-token matrices: The plot shows singular values computed using singular value decomposition (equation 5) for document-token matrices of three attributes (input and output file types (a), name and description (b) and help text (c)). The diagonal matrix S contains singular values sorted in descending order. The x-axis shows the count of singular values and the y-axis shows the corresponding magnitude. The singular values are always real numbers. In (a), (b) and (c), very few singular values have a large magnitude and most of them are small.

correlation matrix). Three such matrices are computed, each corresponding to one attribute (input and output file types, name and description and help text) (figures 14, 17 and 20).

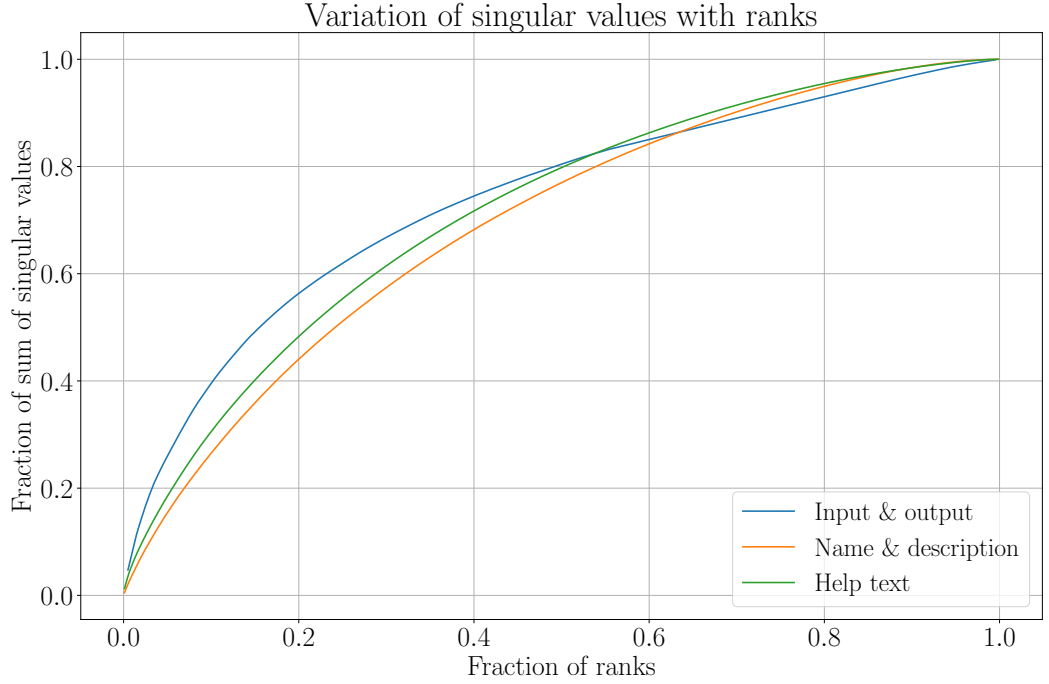


Figure 9.: Variation of the fraction of ranks of document-token matrices with the fraction of sum of singular values: This plot merges the singular values from three different matrices from figure 8 into one plot. As the ranks and sum of singular values of all the document-token matrices vary, they are converted to respective percentages for each matrix. For example, 0.2 on x-axis means 20% of the original rank of a matrix. In equation $rank_{fraction} = \frac{k}{N}$, k is the reduced rank and N is the full-rank of a matrix. Similarly, the y-axis shows the fraction of the sum of all singular values for each matrix. In equation $sum_{fraction} = \frac{\sum_{i=1}^k s_i}{\sum_{i=1}^K s_i}$, K is the number of all singular values and s_i is the i^{th} singular value. k defines the number of top singular values considered for a document-token matrix when it is reduced to rank k . This plot shows that rank reduction of a document-token matrix is directly proportional to the reduction of the sum of its singular values.

2.2.2. Paragraph vectors

Using *latent semantic analysis*, dense vectors are learned to represent each document. One limitation of this approach is to assess the quantity by which to lower the ranks of document-token matrices in order to achieve the right document vectors. This factor can be different for different document-token matrices. There are ways to find an

optimal rank by optimisation using *frobenius norm* (as a loss function). But, it is not simple [8]. The weighted similarity scores are dominated by the similarity scores of input and output file types attribute. The similarity scores from the other attributes remain under-represented (section 4.1). Due to these drawbacks, the tools which have similar functions do not get pushed up on the ranking ladder (more similar tools should be at the top of the ranking ladder). To avoid these limitations, an approach known as *doc2vec* (document to vector) [9] is explored. It learns dense, fixed-size vectors for documents using a neural network. They are unique because they capture semantics present in the documents. The documents which share a similar context are represented by similar vectors. When *cosine similarity* is computed between a pair of vectors sharing similar context, a higher similarity score is observed. Moreover, it allows documents to have variable lengths.

Approach

It learns vector representations for documents and their words. The words which are used in a similar context have similar vectors. For example, tokens like *data* and *dataset* are generally used in similar context. Therefore, they are represented by the vectors which are similar. The vector representations of words in a document are learnt by maximising the following function:

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

where T is the total number of words (w_i) in a document and k is the window size. A window defines a context. A few words, which make a context, are taken and using them, each word is predicted [9]. The probability p is computed using a *softmax classifier*¹³ and *backpropagation*¹⁴ is used to compute the gradient. The *stochastic gradient descent* is used as an optimiser. The paragraph vectors are used to learn the probabilities of next words in a context. Dense vectors are learned for each word and are known as word vectors. The paragraph and word vectors are averaged or concatenated to make a classifier which predicts the next words in a context. There are two ways to choose a context:

- Distributed memory: In this approach, a fixed length window of words are chosen and the paragraph and word vectors are used to predict words in this

¹³<http://cs231n.github.io/linear-classify/#softmax>

¹⁴<http://colah.github.io/posts/2015-08-Backprop>

context. The word vectors are shared across all the paragraphs (documents) and a paragraph vector is unique to each paragraph. Figure 10 explains this approach.

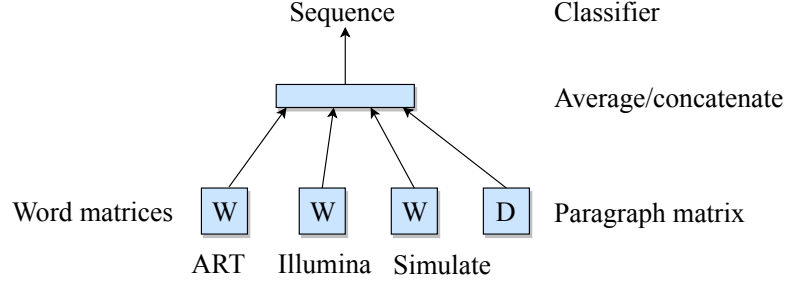


Figure 10.: Distributed memory: The image shows a mechanism for learning paragraph (document) and word vectors. W is a word matrix where each word is represented by a vector. D is a paragraph matrix where each paragraph (document) is represented by a vector. The word vectors are shared across all paragraphs (documents) but not the paragraph vectors. Three words *art*, *illumina* and *simulate* represent a context. The averaged or concatenated words and paragraph vectors make a classifier which is used to predict the word *sequence*.

- Distributed bag-of-words: In this approach, words are randomly chosen from a paragraph. A word is chosen randomly from this set of words and predicted using paragraph vectors. No order is followed in choosing words. Figure 11 explains this approach.

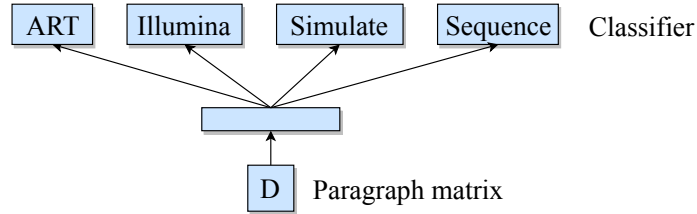


Figure 11.: Distributed bag-of-words: The image shows how paragraph vectors are learned by predicting a random word chosen from a randomly selected set of words. D is a paragraph matrix where each paragraph (document) is represented by a vector. In this approach, the order of the words does not matter.

The figures 10 and 11 are inspired by the original work, distributed representations of sentences and document¹⁵. The second approach of learning paragraph vectors

¹⁵<https://arxiv.org/abs/1405.4053>

(distributed bag-of-words) is simple and is used to learn document (paragraph) vectors for name and description and help text attributes. Only paragraph vectors are learned in this approach and the number of parameters is also less than the distributed memory approach. These reasons make it computationally less expensive [9]. The number of parameters 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

Using *latent semantic analysis* and *paragraph vectors* approaches, document vectors are learned. A similarity measure is applied to get a similarity score for a pair of document vectors. This score quantifies how much similar the pair of documents are. In this work, two similarity measures, *cosine similarity* and *jaccard index*, are used. They return a real number between zero and one as a similarity score.

2.3.1. Cosine similarity

It calculates the value of cosine angle between a pair of document vectors. Two vectors, x and y , are taken:

$$x \cdot y = |x| \times |y| \times \cos \theta \quad (10)$$

$$\cos \theta = \frac{x \cdot y}{|x| \times |y|} = \frac{x \cdot y}{\sqrt{x \cdot x} \times \sqrt{y \cdot y}} \quad (11)$$

where (\cdot) is the dot product. If $(x \cdot x)$ or $(y \cdot y)$ is zero, $\cos \theta$ is set to zero. If the documents are dissimilar, then it is zero and if they are completely similar, it is one. For all other cases, it stays between zero and one. This score is understood as the probability of similarity between a pair of documents¹⁶.

2.3.2. Jaccard index

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

¹⁶<https://nlp.stanford.edu/IR-book/html/htmledition/dot-products-1.html>

$$j = \frac{A \cap B}{A \cup B} \quad (12)$$

where A and B are two sets. The operation \cap gives the number of entities present in both the sets and operation \cup gives the number of unique entities combining both the sets [10]. This measure is used to compute similarity between two documents based on their file types. For example, the tool *linear_regression* has *tabular* and *pdf* as its file types. Another tool *lda_analysis* has *tabular* and *txt* as its file types. Jaccard index (j) for the pair of tools is:

$$j = \frac{\text{length}[(\text{tabular}, \text{pdf}) \cap (\text{tabular}, \text{txt})]}{\text{length}[(\text{tabular}, \text{pdf}) \cup (\text{tabular}, \text{txt})]} = \frac{1}{3} = 0.33 \quad (13)$$

2.4. Optimisation

The similarity matrices are computed by applying similarity measures for each pair of document vectors. These matrices have the same dimensions ($N \times N$, N is the number of documents (tools)). To combine these matrices, a simple idea would be to take an average of the corresponding rows from three similarity matrices. By doing this, the combined similarity scores of one document (tool) with all other documents (tools) are achieved. Iterating this process for all documents gives an average similarity matrix. The diagonal entries of this matrix are one and all the other entries are real numbers between zero and one. A better way to compute the combination of similarity matrices is to learn weights on the corresponding rows of three similarity matrices. These weights, real numbers between zero and one, are used to compute the weighted average of similarity matrices (equation 14). For the corresponding rows in similarity matrices, they sum up to one. Instead of using fixed weights (of $1/3 = 0.33$ (3 is the number of similarity matrices)), an optimisation technique is employed to compute them. A weighted similarity of a tool¹⁷ with all other tools is computed as:

$$SM^k = w_{io}^k \cdot SM_{io}^k + w_{nd}^k \cdot SM_{nd}^k + w_{ht}^k \cdot SM_{ht}^k \quad (14)$$

where each weight (w) is a positive, real number and satisfies $w_{io}^k + w_{nd}^k + w_{ht}^k = 1$. SM_{io}^k , SM_{nd}^k and SM_{ht}^k are similarity scores (matrix rows) for input and output

¹⁷A tool has three documents and each document has a similarity matrix of size $N \times N$ where N is the number of tools.

file types, name and description and help text attributes, respectively for the k^{th} document. The similarity scores, SM^k , has a dimensionality of $1 \times N$ where N is the number of tools. The scores in the similarity matrices are independent of one another. The gradient descent optimiser is used to learn the weights by minimising an error function. The ideal similarity values are set based on similarity measures and are used to define an error function. The maximum similarity between a pair of documents can be one (*cosine distance* and *jaccard index* can be at most one).

$$SM_{ideal} = [1.0, 1.0, \dots, 1.0]_{1 \times N} \quad (15)$$

where SM_{ideal} is ideal similarity scores of a document against all other documents and N is the number of documents. Using the ideal and computed similarity scores, mean squared error is computed (equations 16-18) for all three attributes. The attribute which measures lower error should get a higher weight and the attribute which measures higher error should get a lower weight.

2.4.1. Gradient descent

Gradient descent is a popular algorithm for optimising a function with respect to its parameters. In this work, weights are learned by minimising mean squared error functions:

$$Error_{io}(w_{io}^k) = \frac{1}{N} \times ((w_{io}^k \times SM_{io}^k - SM_{ideal}) \cdot (w_{io}^k \times SM_{io}^k - SM_{ideal})) \quad (16)$$

$$Error_{nd}(w_{nd}^k) = \frac{1}{N} \times ((w_{nd}^k \times SM_{nd}^k - SM_{ideal}) \cdot (w_{nd}^k \times SM_{nd}^k - SM_{ideal})) \quad (17)$$

$$Error_{ht}(w_{ht}^k) = \frac{1}{N} \times ((w_{ht}^k \times SM_{ht}^k - SM_{ideal}) \cdot (w_{ht}^k \times SM_{ht}^k - SM_{ideal})) \quad (18)$$

$$Error(w^k) = Error_{io}(w_{io}^k) + Error_{nd}(w_{nd}^k) + Error_{ht}(w_{ht}^k) \quad (19)$$

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

$$w_{io}^k + w_{nd}^k + w_{ht}^k = 1 \quad (21)$$

In equations 16-19, the subscripts *io*, *nd* and *ht* refer to input and output file types, name and description and help text, respectively. The symbol (\cdot) refers to the dot product of two vectors. Each weight term in equation 21 is a real number between zero and one. To minimise the equation 20, the gradient (partial derivative) of the error function is calculated (equation 22) and weights are learned. The learned weights always satisfy equation 21. The gradient specifies the rate of change of error with respect to weights.

$$Gradient(w^k) = \frac{\partial Error}{\partial w^k} = \frac{2}{N} \times ((w^k \times SM^t - SM_{ideal}) \cdot SM^t) \quad (22)$$

Gradient is a three-dimensional vector ($Gradient_{io}, Gradient_{nd}, Gradient_{ht}$). The symbol (\cdot) refers to dot product of two vectors. Using the gradient vector, weights are updated for each attribute using the following equation:

$$w^k = w^k - \eta \cdot Gradient(w^k) \quad (23)$$

where η is learning rate.

2.4.2. Learning rate decay

Finding a good learning rate is an important part of gradient descent optimisation. If it is high, it poses a risk of optimiser divergence. On the other hand, if it is small, the optimiser can take a long time to converge. Both of these situations are undesirable and should be avoided by starting off with a small learning rate and gradually decrease it over iterations. The decay is important because as learning saturates, only small steps towards the minimum of error function are needed to converge. This technique is called time-based decay [11]. Figure 12 shows the decay of learning rate over iterations computed using the following equation:

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

where lr^{t+1} and lr^t are learning rates for $t + 1$ and t iterations, respectively and *decay* controls how steep or flat the learning rate curve is. *Iteration* is an iteration number. A high value of decay can make the learning rate drop quickly while a low value can make the learning rate drop slowly.

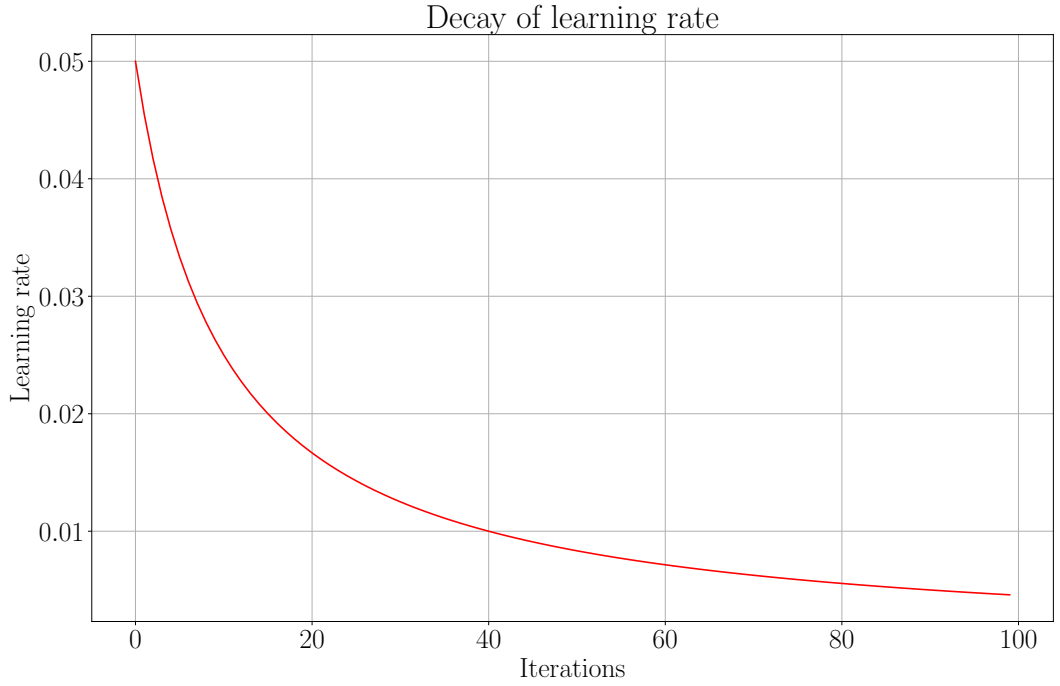


Figure 12.: Decay of learning rate for gradient descent optimiser: The plot shows how the learning rate for gradient descent changes with the iterations. It starts with a small value and decreases gradually over time. It is essential to have the learning rates which neither drops too quickly nor drops too slowly. Both of these ways can lead to the divergence or slow convergence of the optimiser.

2.4.3. Weight update

Momentum

To reach the minimum point of an error function (equation 20), an optimiser should go down continuously without being blocked at saddle points. At these points, the derivative of a function is zero. Adding a momentum term to weights avoids these saddle points and an optimiser converges to the lowest point quickly. It gives a necessary push to keep going down the error function by adding a fraction of the previous update to the current update [11, 12]. The weight is updated for each iteration using:

$$update_{t+1} = \gamma \cdot update_t - \eta \cdot Gradient(w_t) \quad (25)$$

$$w_{t+1} = w_t + update_{t+1} \quad (26)$$

where $update_{t+1}$ is the update for changing weights for the current iteration ($t+1$). $update_t$ is the previous update. η is a learning rate and $Gradient$ is with respect to weight w_t . γ specifies the fraction of previous update to be used.

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 an 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]. The weight is updated for each iteration using:

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

$$w_{t+1} = w_t + update_{t+1} \quad (28)$$

The meanings of terms in equation 28 remain the same as in the previous equations 25 and 26.

Gradient verification

To verify that the gradient computed using partial derivative is correct, an approximated gradient (linear approximation) is computed using equation 29. The difference between these two variants should be close to zero.

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

where ϵ is a small number ($\approx 10^{-4}$). Figure 13 shows the difference of the actual (using partial derivative) and approximated (linear approximation using error function) gradients for all three attributes.

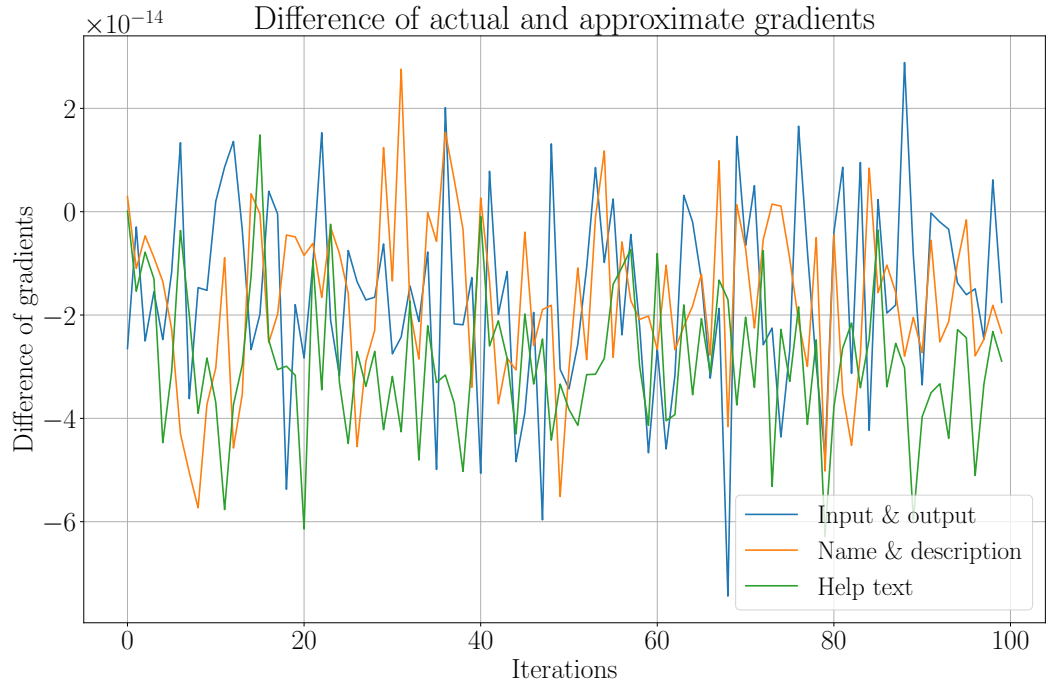


Figure 13.: Difference between actual and approximated gradients: The plot shows a difference between actual and approximated gradients. They are computed for three attributes and averaged for all tools over 100 iterations of optimisation. The difference of gradients is consistent ($\approx 10^{-14}$) which means that the actual and approximated gradients are same. It proves that gradients computed using partial derivatives are correct.

3. Experiments and results

Over 1,050 scientific tools are used for computing similarities among them. Three different attributes, input and output file types, name and description and help text, are used for compiling metadata of tools. The help text attribute is noisy and contains text parts which are not useful for finding similarities among tools. Therefore, only the first four lines of text are used from this attribute. *Jaccard index* is used as a similarity measure for input and output file types attribute and *cosine similarity* is used for name and description and help text attributes. *Gradient descent* is used to optimise the combination of similarity scores computed from three attributes by learning weights for the similarity scores. *Mean squared error* is used as an error function for the optimiser. A true similarity value between a pair of tools is set to one. The optimiser executes for 100 iterations to stabilise the error drop. *Nesterov's accelerated gradient* is used to ensure a steady drop in error. A time-based decay strategy is implemented to decay the learning rate after each iteration. The initial value of learning rate is set to 0.05 and its drop is kept steady.

3.1. Latent semantic analysis

Document-token and the corresponding similarity matrices are sparse. Using *latent semantic analysis*, dense vectors are learned for each document by reducing the ranks of document-token matrices. It allowed the corresponding similarity matrices to become dense too. Due to this, larger weights are learned for similarity matrices. The rank reduction is not applied to the document-token matrix of input and output file types. It is not beneficial to find any correlation among file types. *Singular value decomposition (svd)* is used from its implementation at *numpy's linear algebra package*¹.

¹<https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.linalg.svd.html>

3.1.1. Full-rank document-token matrices

Figure 14 shows the similarity matrices computed using full-rank document-token matrices for input and output (figure 14a), name and description (figure 14b) and help text (figure 14c) attributes. These similarity matrices and weights (from figure 15) are combined to get a weighted average similarity matrix (figure 14d). The weights for input and output file types are larger than that of the other two attributes (figure 15). It is because the similarity matrix of input and output file types (figure 14a) is denser than for name and description (figure 14b) and help text (figure 14c). Figure 16 shows the average of weighted similarities computed using weights learned by the optimiser (optimal weights) and uniform weights ($\frac{1}{3}$, where 3 is the number of attributes). A row-wise average is computed from the weighted similarity matrix (14d) to get an average of weighted similarities with optimal weights. The similarity matrices from figures 14a, 14b and 14c are combined using uniform weights and then, a row-wise average is computed to get an average of weighted similarities using uniform weights. The similarity values in figure 16 are small ($\approx 0.02 - 0.14$).

Similarity matrices

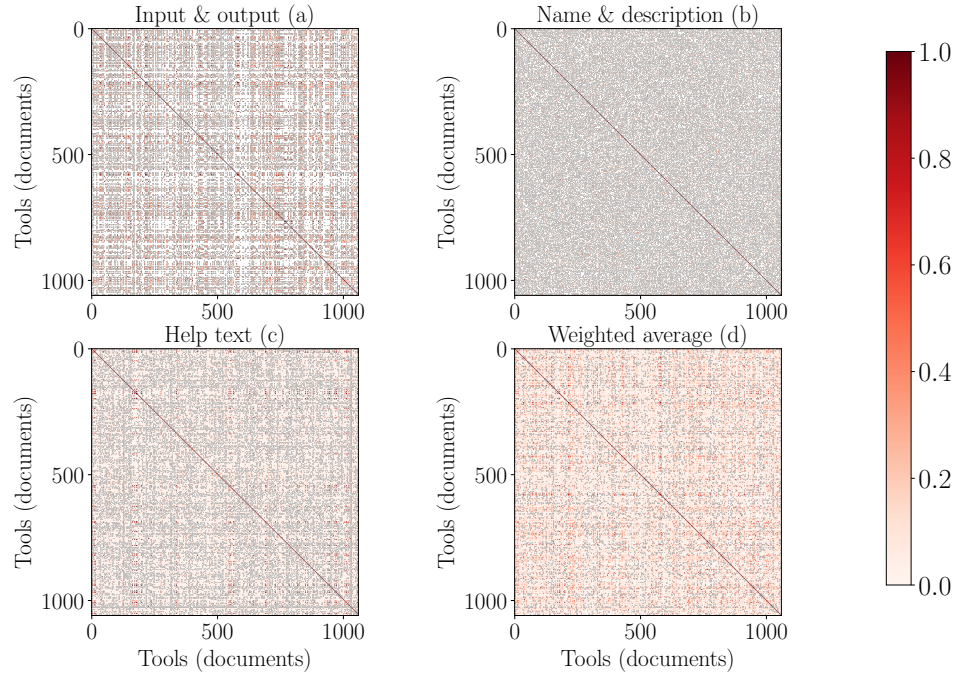


Figure 14.: Similarity matrices computed using full-rank document-token matrices: The heatmaps show the similarity matrices for input and output file types (a), name and description (b) and help text (c) attributes. The subplot (d) shows a weighted average of the similarity matrices computed in (a), (b) and (c). The weights used in computing (d) are shown in figure 15. The corresponding document-token matrices have their full-ranks.

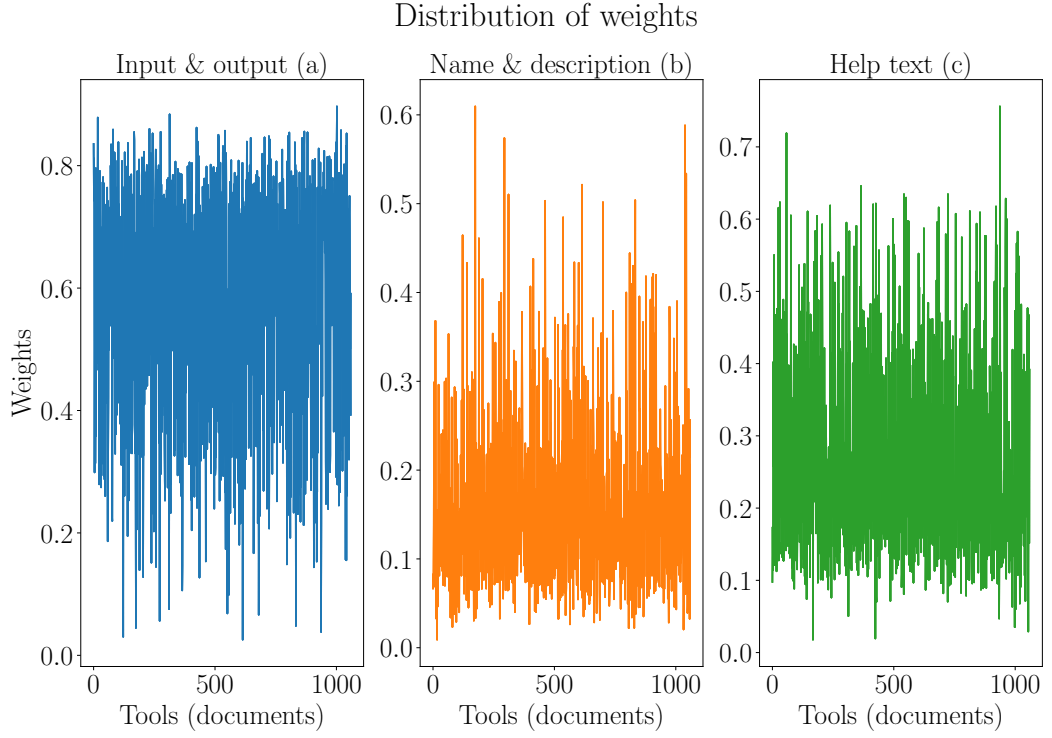


Figure 15.: Distribution of weights learned for similarity matrices computed using full-rank document-token matrices: The plot shows the distribution of weights learned for the similarity matrices in 14a, 14b and 14c by the optimiser for different attributes. The corresponding document-token matrices have their full-ranks.

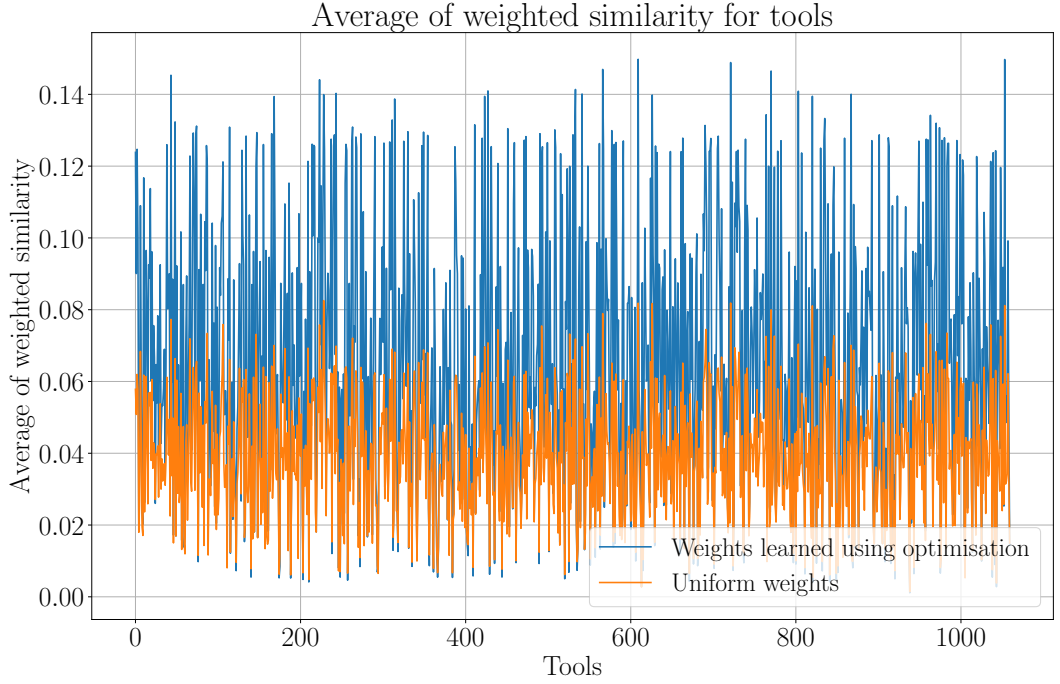


Figure 16.: Average of weighted similarities computed using uniform and optimal weights: The plot shows the averages of weighted similarities computed using the uniform and optimal weights (learned by the optimiser). The corresponding document-token matrices have their full-ranks.

3.1.2. 5% of full-rank

The ranks of the document-token matrices for name and description and help text attributes are reduced to 5% of their corresponding full-ranks. By choosing this low value, only the top $\approx 20\%$ of the sum of singular values is considered (figure 9). Figure 17 shows that the similarity matrices of name and description and help text become denser compared to figure 14. The weights learned for name and description (18b) and help text (18c) are also larger than for input and output file types (18a). An average of the weighted similarities is computed (figure 19) following the same approach explained in section 3.1.1. The similarity values in figure 19 increase ($\approx 0.05 - 0.22$) compared to figure 16.

Similarity matrices

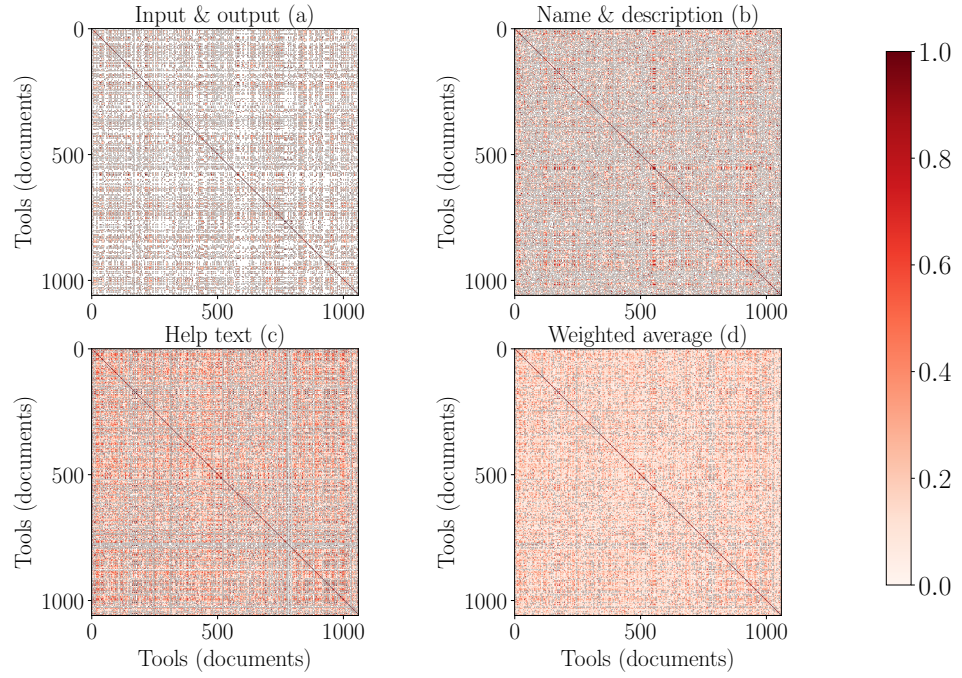


Figure 17.: Similarity matrices computed using document-tokens matrices reduced to 5% of their full-ranks: The heatmaps show the similarity matrices for input and output file types (a), name and description (b) and help text (c) attributes. The subplot (d) shows the weighted average of similarity matrices computed in (a), (b) and (c). The corresponding document-token matrices are reduced to 5% of their full-ranks. The weights used in computing (d) are shown in figure 18.

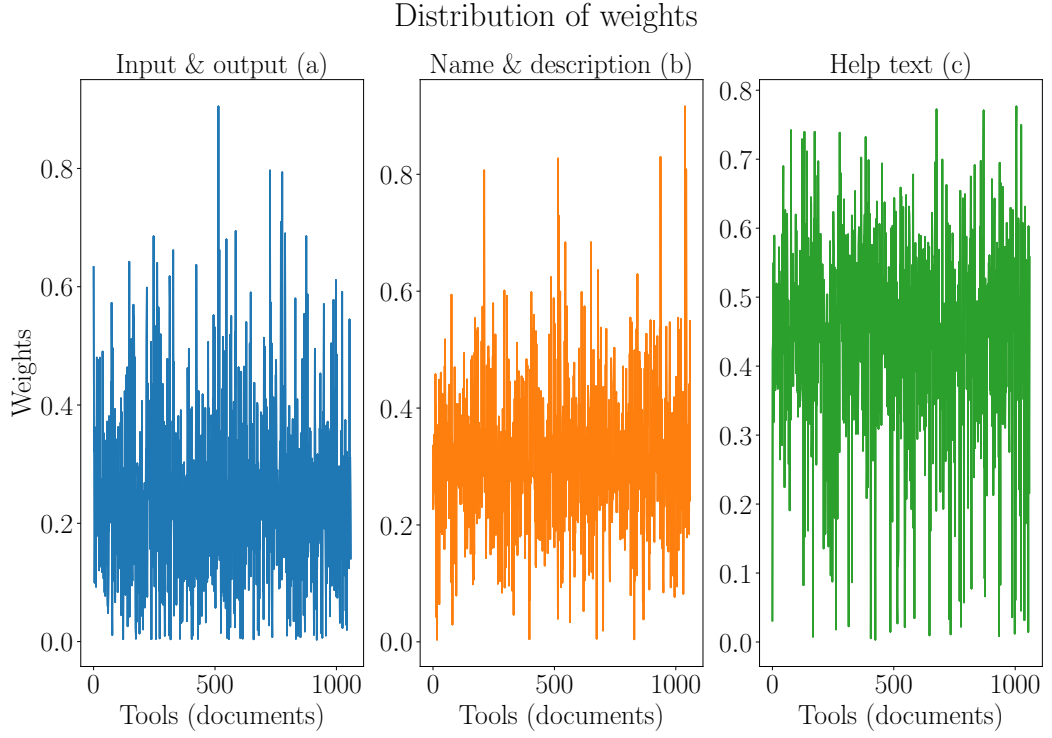


Figure 18.: Distribution of weights learned for similarity matrices computed using document-token matrices reduced to 5% of their full-ranks: The plot shows the distribution of weights learned for the similarity matrices (17a, 17b and 17c) by the optimiser. The corresponding document-token matrices are reduced to 5% of their full-ranks.

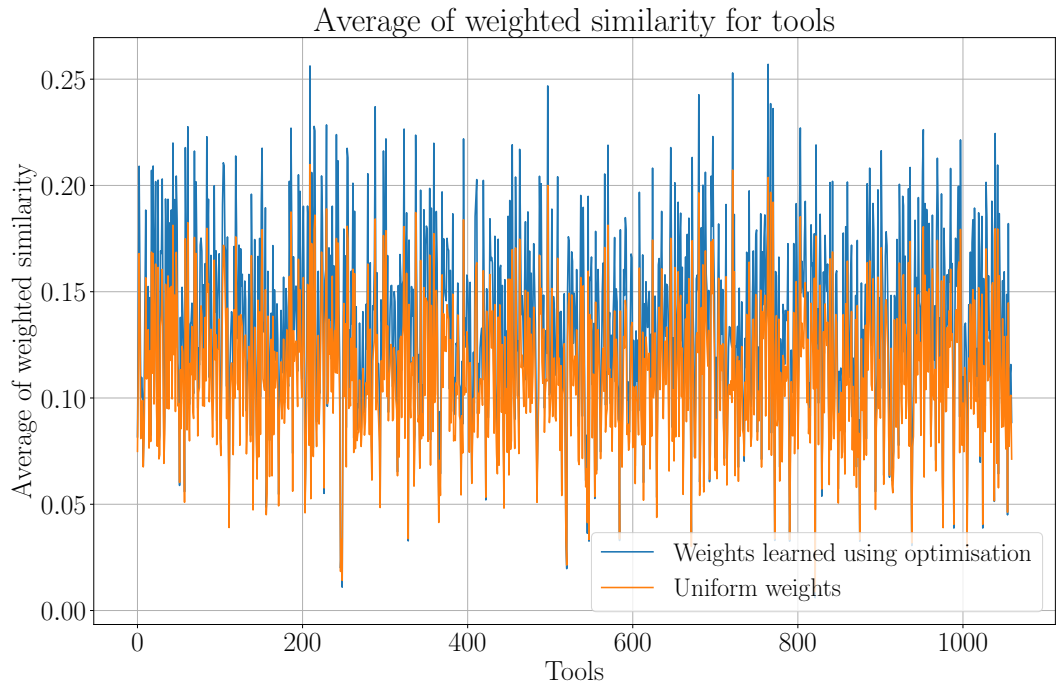


Figure 19.: Average of weighted similarities computed using uniform and optimal weights: The plot shows the averages of weighted similarities computed using uniform and optimal (learned by the optimiser) weights. The corresponding document-token matrices are reduced to 5% of their full-ranks.

3.2. Paragraph vectors

Paragraph vectors approach is used to learn a fixed-length, multi-dimensional vector for each document of name and description and help text attributes. When the number of tokens for a document is lower, a lower number of dimensions is used for computing dense vectors. Figure 5 shows that the number of tokens is higher for help text than name and description attributes. Therefore, the dimension is set to 100 and 200 for name and description and help text, respectively. The window size for sampling text is 5 and 15 for name and description and help text, respectively. The neural network is iterated over 10 epochs and each epoch has 800 iterations. In each epoch, all documents are used for training. A python implementation of this approach, *gensim*², is used to learn these vectors. Out of the two approaches

²<https://radimrehurek.com/gensim/models/doc2vec.html>

(*distributed memory* and *distributed bag-of-words*) to learn dense vectors, *distributed bag-of-words* approach is applied to learn these vectors. It does not compute word vectors which makes it computationally less expensive. Using document-token matrices, the similarity matrices are computed (figure 20) by applying similarity measures. These similarity matrices for name and description and help text are denser compared to figures 14 and 17. The weights learned for name and description (21b) and help text (21c) attributes are larger compared to input and output file types (21a) attribute. Figure 22 shows that the averages of weighted similarities are larger than those of the *latent semantic analysis* approaches (figures 16 and 19). The average of weighted similarity increases to ≈ 0.30 using optimal weights. For uniform weights as well, the average is ≈ 0.25 (figure 22).

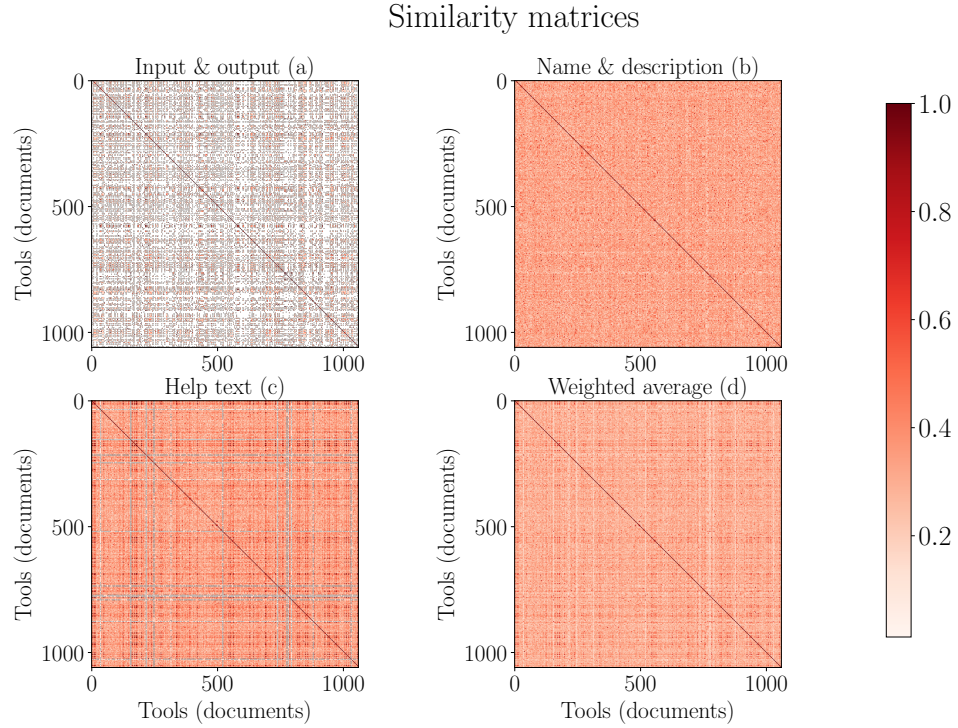


Figure 20.: Similarity matrices using paragraph vectors approach: The heatmaps show the similarity matrices for input and output (a), name and description (b) and help text (c) attributes. The subplot (d) shows a similarity matrix which is the weighted average computed using (a), (b) and (c). The weights used in computing the weighted average similarity matrix in (d) are shown in figure 21.

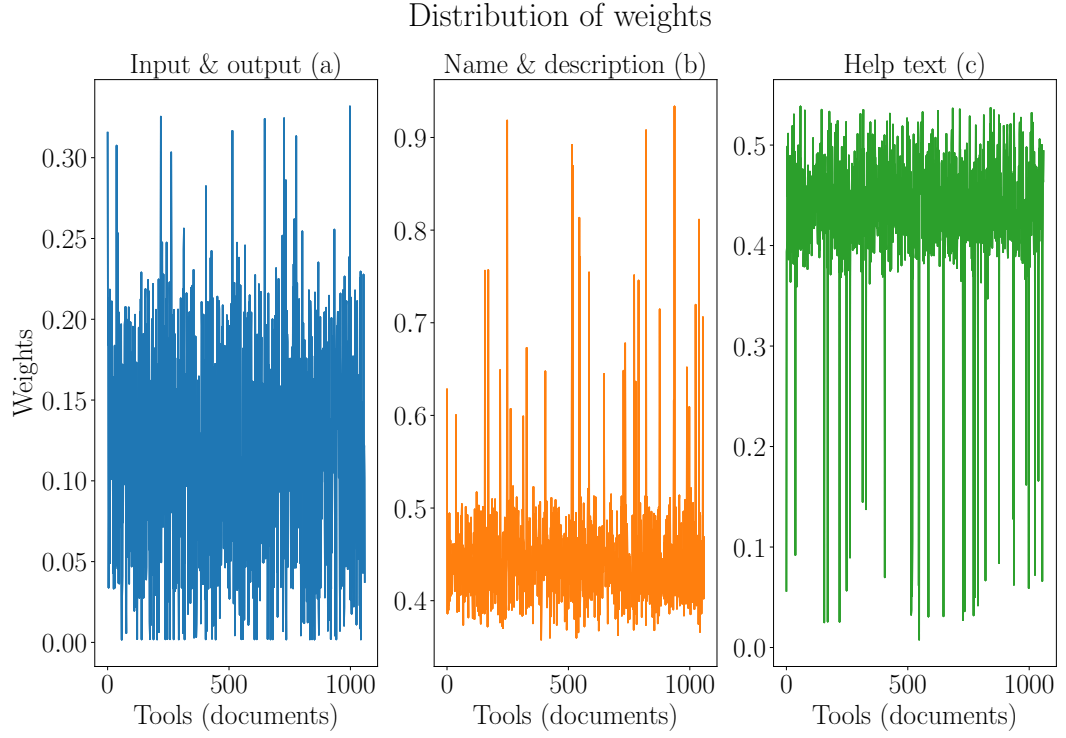


Figure 21.: Distribution of weights for similarity matrices computed using paragraph vectors approach: The plot shows the distribution of weights learned by the *gradient descent* optimiser for the similarity matrices (20a, 20b and 20c) of input and output file types (a), name and description (b) and help text (c) attributes.

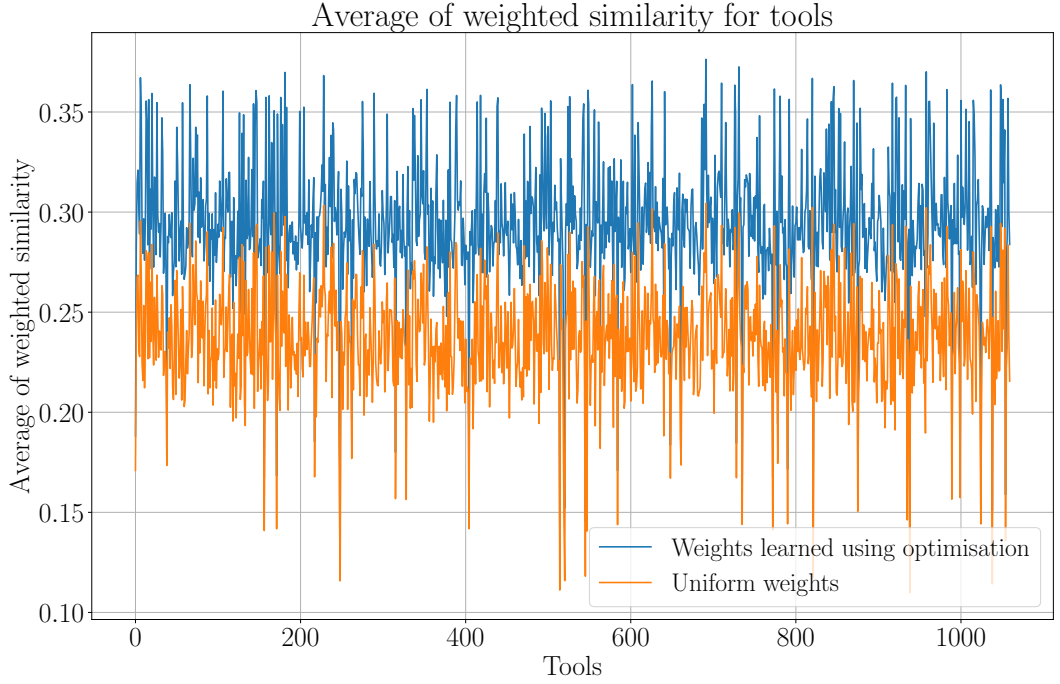


Figure 22.: Average of weighted similarities computed using uniform and optimal weights: The plot shows a comparison of the averages of weighted similarities computed using optimal (learned by the optimiser) and uniform weights.

3.3. Comparison of two approaches

A comparison is done among similar tools computed by two approaches (*latent semantic analysis* and *paragraph vectors*) used in this work. All similar tools for *hisat* are sorted in the descending order of their similarity scores and only top two similar tools are selected for doing a comparison. A complete list of similar tools computed using all different approaches can be seen using the links of visualisers mentioned in the appendix. Top-2 similar tools for *hisat* are computed, once with its full-rank document-token matrices (table 2) and again with document-token matrices reduced to 5% of their full-ranks (table 3). Moreover, two similar tools for *hisat* are computed (table 4) using *paragraph vectors* approach for comparison. Using tables 2, 3 and 4, the similar tools for *hisat* are compared with their respective similarity scores for different attributes. The description below each table gives the values of the optimal weights (learned by the optimiser). For example, weighted similarity in

the first row of table 2 is calculated using the following equation:

$$0.38 \cdot 0.77 + 0.10 \cdot 0.07 + 0.13 \cdot 1.0 = 0.42 \quad (30)$$

From tables 2, 3 and 4, it is concluded that *paragraph vectors* approach works better than *latent semantic analysis* approach. In table 2, for name and description column, the similarity scores are too less (0.07 and 0.12 for *hisat2* and *srma_wrapper*, respectively). In table 3, the similarity scores for name and description are not correct because their descriptions are not exactly same and they get 1.0 as similarity. These incorrect interpretations can lead to wrong similarity assessment. As the low-rank estimation of the document-token matrix of input and output file types attribute is not done, its score remains same in tables 2 and 3. *Hisat2* has the same score for input and output column in all the tables 2, 3 and 4. However, in table 4, the similarity scores seem to be reasonable, neither too high and nor too low. The similar tool ranked second (*tophat*) in table 4 is more relevant than those from tables 2 and 3.

Similar tools	Input & output	Name & desc.	Help text	Wt. 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* computed using full-rank document-token matrices: The table shows top-2 similar tools selected for *hisat*. Full-rank document-token matrices are used for computing similarity matrices. The weights learned by the optimiser are 0.77 (input and output file types), 0.10 (name and description) and 0.13 (help text). *Wt. similarity* is weighted similarity.

Similar tools	Input & output	Name & desc.	Help text	Wt. 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* computed using document-token matrices reduced to 5% of full-rank: The table shows top-2 similar tools selected for *hisat*. Document-token matrices of name and description and help text attributes are reduced to 5% of their full-ranks. The weights learned by the optimiser are 0.27 (input and output file types), 0.23 (name and description) and 0.5 (help text). *Wt. similarity* is weighted similarity.

Similar tools	Input & output	Name & desc.	Help text	Wt. 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* computed using paragraph vectors approach: The table shows top-2 similar tools selected for *hisat*. The weights learned by the optimiser are 0.14 (input and output file types), 0.44 (name and description) and 0.42 (help text). *Wt. similarity* is weighted similarity.

4. Conclusion

4.1. Metadata of tools

The help text attribute was noisy containing a lot of words which were generic and provided little information to identify the tools. On the other hand, it was necessary for the analysis as it supplied more metadata. Therefore, it needed more filtering than other attributes. Only the first four lines of text were taken for the analysis from help text. The metadata from input and output file types and name and description was helpful. The extraction of metadata from the *github*'s multiple repositories was slow¹. Therefore, the metadata from the *xml* files was read into a tabular file and it was used as the data source for this analysis.

4.2. Approaches

4.2.1. Latent semantic analysis

Latent semantic analysis approach relied on the rank reduction of document-token matrices. It removed unimportant dimensions and worked better than using the full-rank documents-token matrices. However, due to the lack of knowledge of the exact amount of rank reduction, the loss of important dimensions was also risked. During optimisation, more importance was given to input and output file types which were many times undesirable. The similarity scores learned for different attributes were sometimes under-represented or over-represented as explained in section 3.3. Due to these limitations, this approach should not be used for computing similarities among tools. It may lead to wrong similarity assessment. However, this approach was simple and took less time (≈ 350 seconds) to complete. The low-rank estimations of the sparse document-token matrices were easily computed using *singular value decomposition*. The results can be visualised using the links

¹The extraction of $\approx 1,050$ tools took ≈ 30 minutes

mentioned in the appendix. Though it did not compute similar tools correctly, it provided insights to look for approaches which can learn better document vectors.

4.2.2. Paragraph vectors

Paragraph vectors worked in a more robust way than the *latent semantic analysis* approach to find similarities among tools. The vectors learned for name and description and help text were better compared to the *latent semantic analysis* approach as they learned similar vectors for similar documents. Due to these vectors, relevant similarity scores were computed among tools. A better distribution of weights was learned for the attributes and the weighted similarity was more dominated by the similarity scores from name and description and help text. Though this approach performs much better than the previous approach, few dissimilar tools appear in the list of top-20 similar tools for a tool. These results can be seen in the visualiser². It happens because the similarity assessment is dependent on the plain text in name and description and help text attributes. Sometimes they are not descriptive (for name and description attribute) or even absent (for help text attribute). Due to this, the amount of text for the neural network is less to learn vectors which can differentiate among documents in a good way. Despite this limitation, this approach leads to a good starting point to visualise similar tools. The tools at the top of the list of similar tools are relevant for many tools. The computation of similar tools was slow as it took $\approx 1,000$ seconds to finish. Most of the time was spent to learn the document vectors.

Significance to the recommendation system

The second part of the thesis deals with the prediction of next tools for workflows. To build a recommendation system, alongside the predicted next tools for any workflow, their respective similar tools (for example, top-2 or top-3) computed using *paragraph vectors* approach can be shown. This would give more options to the users for their data processing using the Galaxy as they can replace a tool by choosing a tool from the combined set of the predicted next tools and similar tools. Moreover, this recommendation system can make users aware of the newly added tools if they are similar to the existing tools.

²https://rawgit.com/anuprulez/similar_galaxy_tools/doc2vec/viz/similarity_viz.html

4.3. Optimisation

Learning weights on similarity scores worked in a stable way and reached the saturation point already before the 50th iteration. Therefore, the number of iterations for the *gradient descent* can be reduced from 100 to ≈ 50 . The *gradient descent* optimiser was used with *mean squared error* as the loss function. *Mean squared error* was used because the hypothesis similarity scores distribution needed to be as close as possible to the true distribution (based on the similarity measures). The risk of getting stuck at saddle points or local minima was reduced by using momentum with an accelerated gradient to update the weights (parameters of the error function).

5. Future work

5.1. Get true similarity values

To quantify improvements in the ranking of similar tools for a tool computed by different approaches, true similarity scores should be set for each pair of tools. For example, a dictionary of 10 similar tools can be created for each tool. Then, the computed similar tools can be verified against this true set of similar tools. Otherwise, it is required to have a visualiser to look through the similar tools. It would ensure whether the approaches actually work in finding similar tools. At the same time, it is not an easy task to create these logical categories and define similarity scores within these categories.

5.2. Exclude low similarity scores

The similarity matrices were dense. The low scores from similarity matrices can be excluded to retain only high scores. One way is to find the median of similarity scores for a tool. The similarity scores lower than the median can be set to zero. Then, the optimisation would use only important scores to compute and minimise the error.

5.3. Formulate different error functions

Mean squared error was used as a loss function for the optimiser. Other error measures like *cross-entropy* can be used. With a new error function, it is necessary to redefine the true similarity value which would depend on the similarity measures. Using *cross-entropy* error function with the same similarity measures (*cosine similarity* and *jaccard index*), one term in the error function would always be zero as the true similarity between a pair of tools would be 1.0. Different similarity measures like

euclidean distance can be used. For this, the true similarity value between a pair of tools should be 0.

5.4. Acquire more tools

The number of tools for performing the analysis can be increased. It would provide more data and consequently, more context to learn better semantics in text documents.

5.5. Learn tool similarity using workflows

It can be assumed that the tools which are similar are used in similar context in workflows. It means that similar tools are used for similar kind of data processing. This concept can be used to assess similar tools. The distributed memory approach learns vectors for words in a document. The document can be replaced by paths extracted from workflows and tools can be replaced by words. Then, dense vectors for tools can be learned. The tools which are used in similar context would learn similar vectors.

Part II.

Predict next tools in scientific
workflows

6. Introduction

6.1. Galaxy workflows

A Galaxy workflow is a sequence of scientific tools to process biological data. The tools are connected one after another forming a data processing pipeline. This pipeline allows multiple transformations of a dataset in one operation¹. Adjacent tools are connected through compatible data types which means that the output data type of one tool is consumed by its next tool. A workflow is a directed acyclic graph (figure 23). It can have multiple paths between its input and output tools and these paths can have one or more tools in common. Each path has a direction commencing from an input tool and ending at an output tool. In figure 23, the tools like *get flanks*, *awk* and *intersect* are common in both the paths.

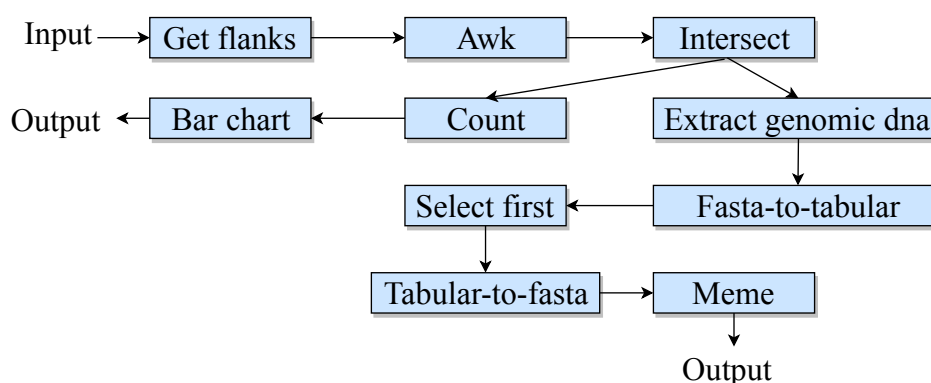


Figure 23.: A workflow: The image shows a workflow with two paths. It takes input data, processes it through multiple steps involving many tools and gives two outputs.

To create a workflow, a tool is chosen from the list of tools and is connected to the previous one. At each step, a tool can connect to one or more tools which are compatible with the previous one. The decision to select a tool may depend on

¹<https://galaxyproject.org/learn/advanced-workflow>

several factors like the kind of input data or data-processing required to attain the desired output. A workflow can also be created using datasets from the Galaxy history².

6.1.1. Motivation

To create a workflow, having multiple paths where each path can have many tools, is a complex task. A user needs to have knowledge of the tools and their parameters like input and output types that should come next. These next tools must be compatible with the previous tool because no two incompatible tools can connect to each other on the canvas of Galaxy workflow editor. This knowledge can be gained only through experience. A user who is new to the Galaxy platform and is not aware of the existing tools, creating a workflow can be a laborious and time-consuming effort.

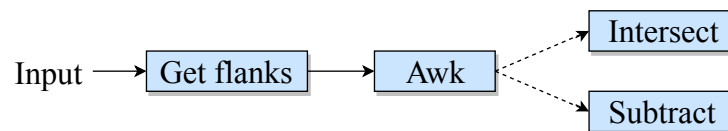


Figure 24.: Multiple next tools: The image shows a part of a workflow where one tool can be connected to multiple tools.

In figure 24, the tool *awk* can be connected to two tools, *intersect* and *subtract*. The number of next tools for any tool can vary. Some tools can have just one next tool but others can have multiple next tools. To impart the knowledge of next tools (for a path) and assist a user at each step of creating a workflow, a predictive system to recommend next tools is proposed. It would recommend a set of next possible tools to a user whenever he/she chooses or connects a tool to another tool. It would present the most probable set of next tools making it easier to create a workflow. A reduction in the amount of time taken to create workflows is also expected.

²<https://galaxyproject.org/tutorials/histories>

7. Related work

Paths extracted from workflows are sequential in nature as tools are connected one after another. All the previous connected tools should be taken into account to predict next tools. To make such a prediction system, it is important to explore some work from the machine learning field which analyse similar data. Interestingly, learning from sequential data is a popular task in many fields like natural language processing [14, 15], clinical data analysis [16] and speech processing [17, 18].

In the natural language processing field, deep learning models are used to learn sequences of words. It aims to categorise a corpus based on its sentiments and learn part-of-speech tagging and dense vector for each word. The categorisation of sentiments involves learning core contexts present in the sentences (sequences of words). The part-of-speech tagging divides a sentence into multiple parts-of-speech like an adjective, adverb or conjunctions. It also depends on finding the contexts hidden in the long sequences of words. An accuracy of $\approx 85\%$ is achieved using the recurrent neural network (*gated recurrent unit*) for sentiment analysis and for part-of-speech tagging, the accuracy goes up to $\approx 93\%$ [14].

For clinical data too, learning long sequences of data proves to be beneficial [16]. In this work, the health states of patients recorded at different time points are analysed by accessing their electronic health records. Using a learned model, the future health states of a patient are predicted using the sequence of his/her health states in the past. The *long-short term memory (lstm)*, a kind of recurrent neural network, is used to classify the sequences of health states. An accuracy of $\approx 85\%$ is achieved by applying the necessary regularisation techniques like dropout and weight normalisation.

The research presented in [17, 18] use the recurrent neural networks to model music and speech signals. The performances of the traditional recurrent, *long-short term memory (lstm)* and gated recurrent units (*gru*) are analysed. After the analysis, it is concluded that the traditional recurrent units do not learn semantics present in sequences, but the *lstm* and *gru* do. In this study, the sequences of 20 continuous

samples (20 steps in time-series of speech) are learned and the next 10 continuous samples (next 10 time steps) are predicted. It is also found out that the *gru* performs better than the *lstm* in terms of accuracy and running time. For musedata¹, a collection of piano classical music, the performances (average of the negative log-likelihood) noted on the test set are - *gru*: 5.99, *lstm*: 6.23 and traditional recurrent units: 6.23. Six different types of musical and speech data are tested and four out of these six types, the *gru* works better than the other two units.

These successful approaches benefit from the state-of-the-art sequential learning techniques like the *lstm* and *gru* recurrent neural networks. For the Galaxy's scientific workflows, learning tool connections to predict next possible tools is inspired by these approaches.

¹www.musedata.org

8. Approach

The prediction of a set of next tools at each step of designing a workflow is proposed in this work. A learning algorithm is needed which can learn tool connections in a path and predict its next tools. This learning algorithm is called a classifier. There are two coveted features of this classifier:

- It should grasp the semantics of tool connections in workflow paths and use them to predict next tools (with an accuracy of $\geq 90\%$).
- Running time and space complexity should be reasonable.

The classifier's running time and its accuracy to predict next tools are noted for doing a comparison. Preprocessing of the workflows is necessary to make them usable for analysis by a classifier. Figure 25 highlights the sequence of steps to preprocess them. First, paths are extracted from workflows and the duplicates are removed. These paths are required by the classifier to understand the dependencies among tools. They are processed by using three different approaches to create shorter paths. Section 8.4.1 describes these approaches in detail. The paths are divided into training and test paths. The classifier consumes training paths to learn their characteristics along with their next tools. The robustness of learning is verified on test paths.

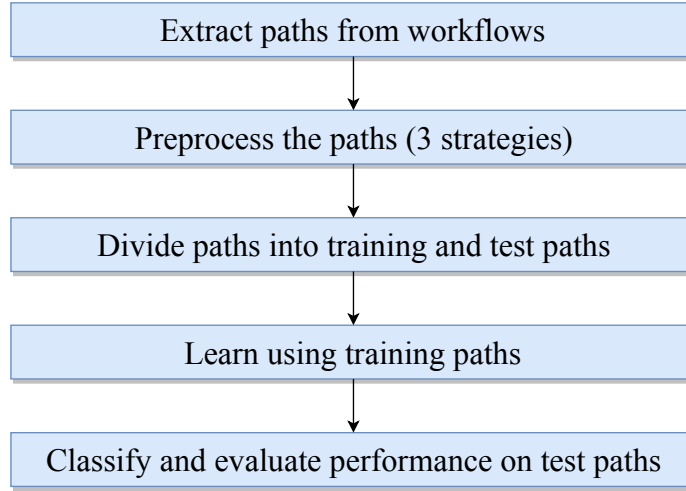


Figure 25.: Sequence of steps to predict next tools in workflows: The flowchart shows a sequence of steps to predict next tools in workflows. Workflows are decomposed into paths. Further, these paths of varying lengths are decomposed to have shorter paths by following three different strategies explained in section 8.4.1. A classifier learns tool connections from training paths. The learning is evaluated on test paths.

8.1. Actual next tools

Figure 23 shows that a workflow can have more than one path and these paths can share a few tools. Moreover, a path can connect to more than one tool. A classifier needs data in the form of a path and its next tools (a tool or a set of different tools).

Path	Tool(s)
Get flanks	Awk
Get flanks > awk	Intersect
Get flanks > awk > intersect	Count, extract genomic dna
Get flanks > awk > intersect > count	Bar chart

Table 5.: Decomposition of a workflow: This table shows a few paths and their respective next tools extracted from a workflow shown in figure 23. The next tools of a path are the actual next tools present in the workflow.

A workflow is decomposed into paths. Using the technique explained in table 5, these paths are further decomposed into shorter paths and their respective next tools. This idea is considered because of the necessity of getting predictions in the

same way. It means that when the tool *get flanks* is selected, a classifier should suggest *awk* as a next possible tool. Again, the tool *awk* is selected and the sequence becomes *get flanks > awk*. Given this sequence, the classifier should predict the tool *intersect*. To this sequence, when the tool *intersect* is added, the classifier should predict two tools, *count* and *extract genomic dna* (figure 23). Following this approach of decomposition of paths, shorter paths and their respective next tools are created. The next tools that are assigned to a path are its actual next tools. It means that the path is connected to these next tools in workflows.

8.2. Compatible next tools

Tools are connected in a workflow because of their compatible input and output data types. Some can connect only to a few tools, but some can connect to a larger set of tools. All pairs of connected tools are extracted from workflows. Each pair has compatible data types as they are connected. From this set of numerous pairs of tools, a list of compatible next tools is collected for each tool. In a path, *get flanks > awk > intersect > count > bar chart*, the tool *intersect* connects to *count* tool. It means that these two tools have compatible data types. There are multiple other tools which can connect to *intersect* like *join*, *cut*, *subtract* and many others. These form a compatible set of next tools for *intersect* and can be predicted as next tools for the paths ending in *intersect*. When a classifier predicts next tools for a path and if the last tool of this sequence is compatible with the predicted tools, then the predictions are correct and should be accounted for. These tools are not fed to the classifier for learning but are used only to verify whether the prediction of next tools includes these compatible tools. The number of compatible tools for a tool varies. For example, a tool *concatenate datasets* has ≈ 190 compatible next tools while another tool *mothur dist seqs* has only one compatible next tool. Figure 26 shows a distribution of the number of compatible next tools for each tool. It can be seen that some tools have just one or two next tools while some have a larger number of compatible next tools. The set of compatible next tools is computed from all workflows analysed in this work.

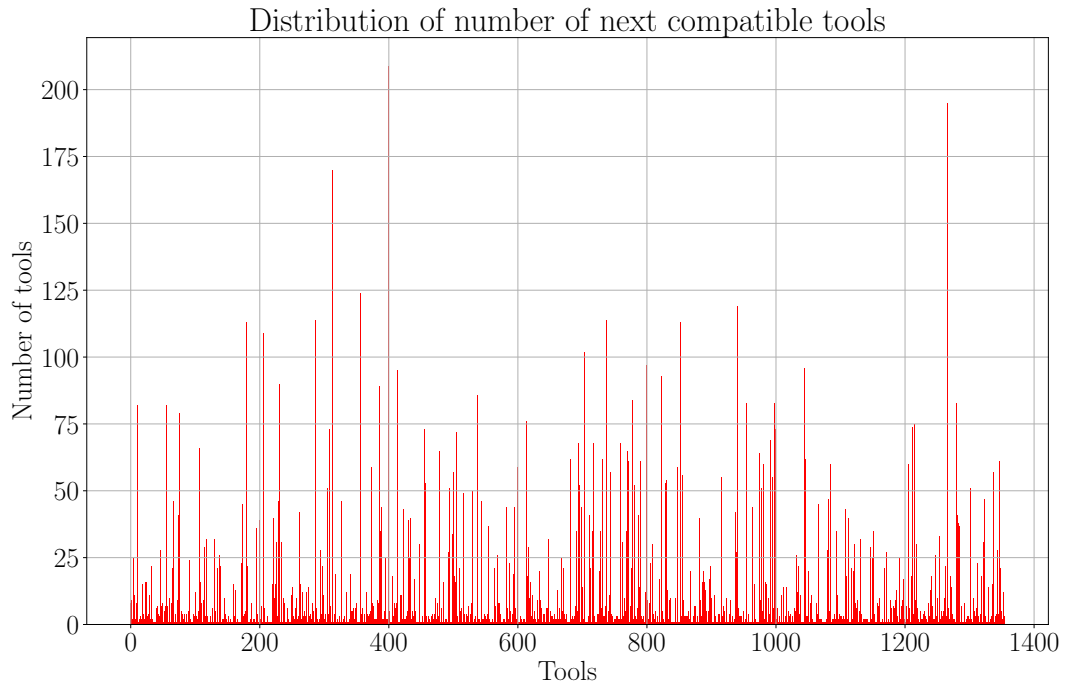


Figure 26.: Distribution of the number of compatible next tools: The bar plot shows a distribution of the number of compatible next tools for each tool. The x-axis shows tools which have at least one compatible next tool. The y-axis shows the number of compatible next tools.

8.3. Length of workflow paths

The size (number of tools in a path) of paths in workflows varies. A path can be long or short. Figure 27 shows a distribution of the number of tools present in each path. Most of the paths have less than 20 tools while a few contain over 20 tools. It is important to find how the prediction of next tools varies with the sizes of workflow paths and whether the classifier is sensitive to these sizes.

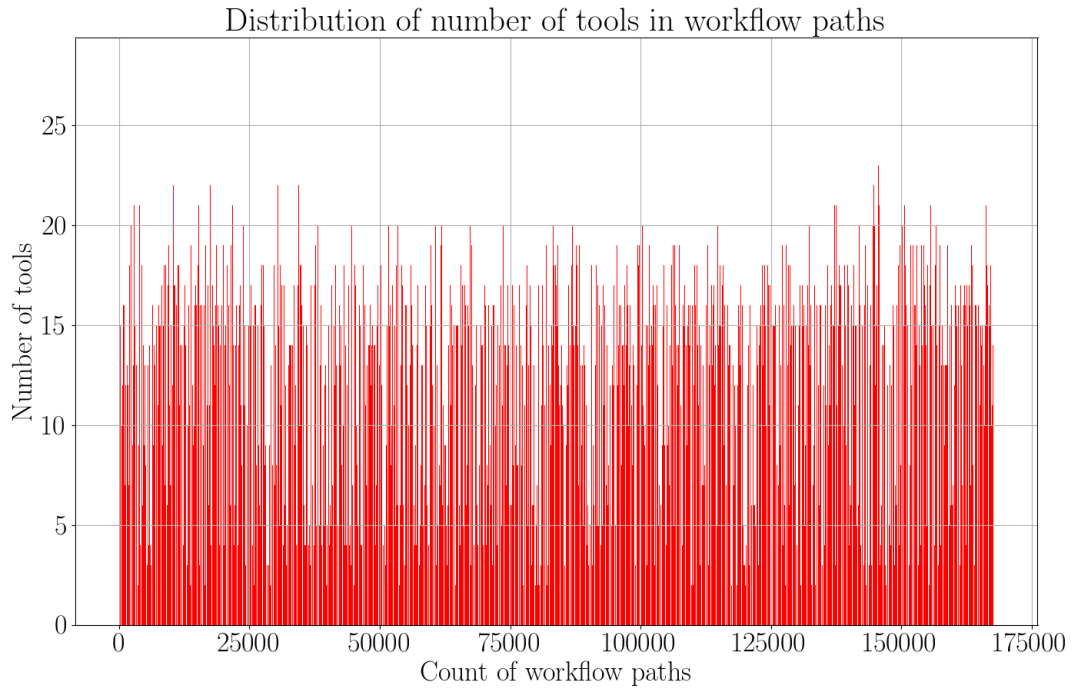


Figure 27.: Distribution of sizes of workflow paths: The plot shows a distribution of the sizes of workflow paths. A workflow path consists of multiple tools. This distribution shows how many tools are there for each path. About 167,000 paths are extracted from the Galaxy workflows and most of them have less than 20 tools.

8.4. Learning

8.4.1. Workflow paths

A path enforces a directed flow of information through multiple tools connected in a series (from left to right). The paths are preprocessed in such a way so that they can retain their semantics and can be understood by any classifier. Three different ways of decomposing these paths into tool sequences¹ are used. The decomposed tool sequences are further divided into training and test paths. The classifier uses training paths to learn features. The amount of learning is evaluated on test paths. It is ensured that the intersection set of training and test paths is empty. Otherwise, the evaluation of the learned model would be biased [19, 20, 21]. Different methods

¹A path and a tool sequence refer to the same entity, a chain of tools, in the thesis. They are used interchangeably.

of the decomposition of paths are discussed in the following sections.

No decomposition

In this approach, paths are used as they are. The last tool of each path is taken as its next tool. A dictionary is created in which these paths are keys and their next tools are values. This dictionary is shuffled and divided into training and test paths. It enforces that no path is present in both the sets of paths. A classifier needs to learn tool connections in these paths and their respective next tools. The learned model is used to predict the last tool of each path in the test set. Figure 28 explains this idea.

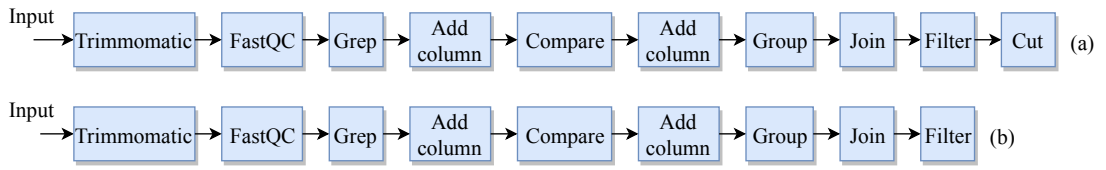


Figure 28.: No path decomposition: The image shows a workflow path. For a path of length n (a), a shorter path of length $n - 1$ (b) is taken out and the last tool is assigned as its next tool. For example, the tool *cut* is a next tool of the previous path.

Decomposition of test paths

In this approach, training paths are used as they are to train a classifier. But, test paths are decomposed keeping the first tool fixed in each path. Figure 29 shows this decomposition. For a path of length n , $n - 1$ unique paths are created and for each path, the last tool becomes its next tool. A path should contain at least two tools (the second tool is the next tool of the first). This decomposition increases the size of test paths. Moreover, it creates a problem of duplication of paths in training and test paths. A test path is decomposed into shorter paths. Maybe one or more of these shorter paths are present in training paths. The evaluation would be biased in this case. To avoid this situation, the duplicates are removed from the training and test paths before they are fed to the classifier.

Decomposition of test and train sets

By decomposing paths keeping the first tool fixed, learning task is made easier for the classifier. In the previous approach, only test paths are decomposed. But, in

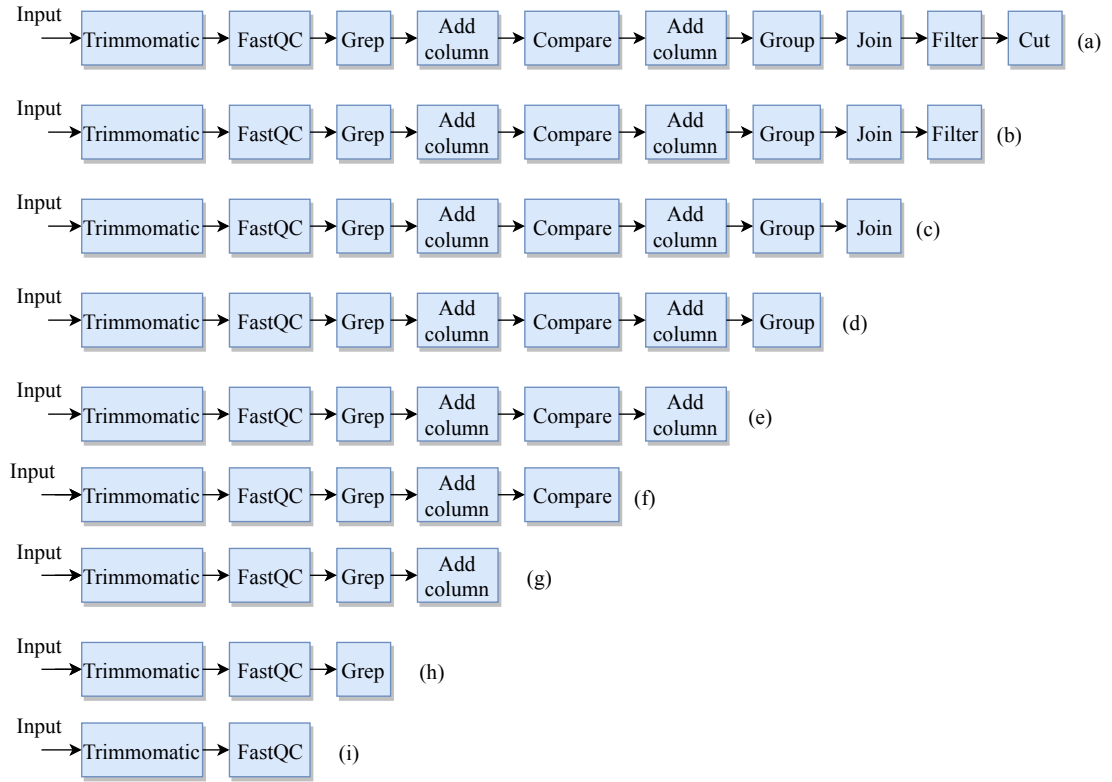


Figure 29.: Path decomposition: The image shows how a path is decomposed into smaller tool sequences keeping the first tool fixed. For each path, the last tool is dependent on all its previous tools (higher-order dependency).

this approach, all paths are decomposed. The strategy explained in figure 29 is used to obtain a mixture of shorter and longer paths. For this approach too, a dictionary is created where each key represents a path and its value is a set of next tools. This dictionary is shuffled and divided into training and test paths.

8.4.2. Bayesian networks

A workflow possesses the structure of a directed acyclic graph. Bayesian network is one of the approaches to model this graph. From bayesian network, it is inferred that if the parents of a node are given (in a directed graph), then this node is conditionally independent of all the other nodes which are not its descendants (the set of nodes it cannot reach through directed edges) [22, 23]. It means that each node in this graph is dependent only on its parents. The structure of workflows can be explained by this model. Bayesian networks can be used to model workflows and the missing values

(of nodes) can be predicted. Here, the missing values would be next tools. But, there are a few limitations of this model which are worth considering. The usage of this model involves computing joint probabilities of nodes in a graph and also the conditional probabilities among them. When the number of nodes increases, it will become hard to keep the computational cost low. Making predictions by learning a probabilistic network is a hard problem [24, 25, 26]. Due to these reasons, the bayesian network is not used in this work for predicting next tools in workflows.

8.4.3. Recurrent neural networks

A workflow may have multiple paths. These paths are extracted from workflows and the duplicates are removed. They are assumed to be independent of each other which keeps the analysis simple and powerful. In a path, each tool is dependent on all its parents (previous tools in the path). This relation is called higher-order dependency [27]. It is important to learn these higher-order dependencies in order to be able to predict next tools for a path. Figure 30 shows these dependencies for a path. Tool *sort* is not only dependent on *text reformatting* tool but on tools *datamash* and *concatenate datasets* too.

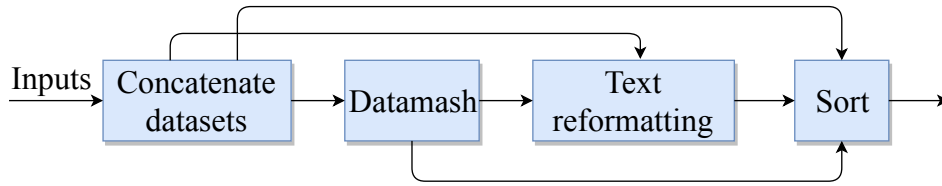


Figure 30.: Higher-order dependency: The image shows a path where four tools are arranged in a sequential manner. Each tool depends not only on its immediate parent tool, but all previous tools in the path.

A classifier should model these dependencies for variable length paths. The recurrent neural network is a natural choice for the classifier which can learn these dependencies in workflow paths. To model these dependencies, the network keeps a hidden state at each step of processing paths. It combines information from previous tools and the current tool. It is computed as:

$$h_t = \phi(h_{t-1}, x_t) \quad (31)$$

where h_t is the hidden state at step (or time) t , h_{t-1} and x_t are the hidden state at the previous step ($t - 1$) and input at t respectively. ϕ is a nonlinear function.

More formally, the equation 31 is written as:

$$h_t = g(W_{input} \times x_t + U_{recurrent} \times h_{t-1}) \quad (32)$$

where W_{input} and $U_{recurrent}$ are the weight matrices for the input and recurrent units respectively. The hidden state keeps information about the previous steps. At each step, x_t is an input. $g()$ is a bounded, nonlinear function like *sigmoid* (equation 39). The joint probability of all these input variables (x_i) is given by:

$$p(x_1, x_2, \dots, x_T) = p(x_1) \cdot p(x_2|x_1) \cdot p(x_3|x_1, x_2) \cdot \dots \cdot p(x_T|x_1, x_2, \dots, x_{T-1}) \quad (33)$$

where x_t is the input at step t , $p(.)$ is a probability distribution and T is the length of a path. From equation 33, the input at step t is reliant on the previous steps in a sequence. To predict next step (or next tool in a path), the last conditional probability ($p(x_T|x_1, x_2, \dots, x_{T-1})$) should be captured using the hidden state (h_t) [17, 28].

In equation 32, there are two matrices, one each for the recurrent units and inputs, respectively. Learning higher-order dependencies depends on the gradients of errors with respect to the parameters (recurrent and input weight matrices). The gradient at a step is the product of gradients from all previous steps. If the gradients from previous steps are small which inherently means that the recurrent units are not capturing higher-order dependencies, then their product can quickly slip towards zero and disappear (the product of small numbers would be small). In another scenario when gradients from previous steps are large, the product can easily explode to become a large number. Both these situations are collectively known as vanishing and exploding gradients problem respectively [29]. The traditional recurrent units are prone to show this behaviour. To avoid these situations, which can hamper learning, two variants of recurrent units are proposed:

- *Long-short term memory units (lstm)* [30]
- *Gated recurrent units (gru)* [28]

In this work, *gru* is explored which is proposed recently and is simpler than the *lstm* and contains fewer parameters. The performance of these two variants is comparable [17]. Figure 31 shows a recurrent neural network with two hidden layers.

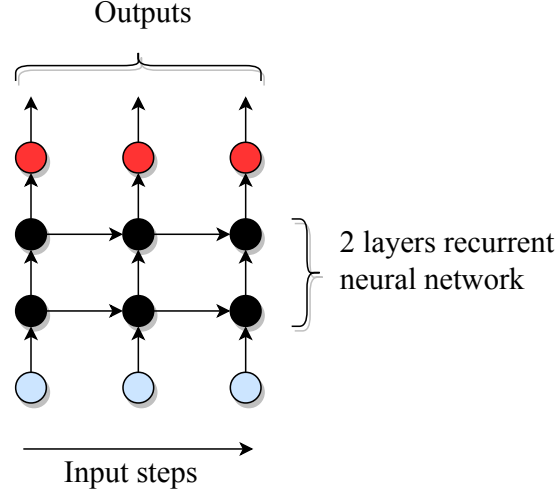


Figure 31.: Recurrent neural network: The image shows the stacked gated recurrent units layers and how they pass information from the input layer to the output. The output of the first recurrent layer is used as an input to the next recurrent layer. The blue circles denote the input steps (different tools in a path), the black ones form the two recurrent layers and the red circles form the output layer [31].

Gated recurrent units

The *gru* has gates which control the flow of information from earlier to later steps in a multi-step input sequence. Figure 32 shows the reset and update gates. The reset gate r checks how much information from previous steps should be carried to the next step. If it is 0, then all information from previous steps is discarded. If it is 1, all information is taken to the next step. The update gate controls how much of the unit's own activation would be used to compute the current activation.

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times h'_t \quad (34)$$

where h_t and h_{t-1} are the current and previous step activations respectively, z_t is the value of update gate and h'_t is the current proposed activation. The update gate is calculated using the following equation:

$$z_t = \sigma(W_z \times x_t + U_z \times h_{t-1}) \quad (35)$$

where $\sigma(\cdot)$ is the *sigmoid* function and W and U are the input and recurrent weight matrices respectively. The current proposed activation (in equation 34) is

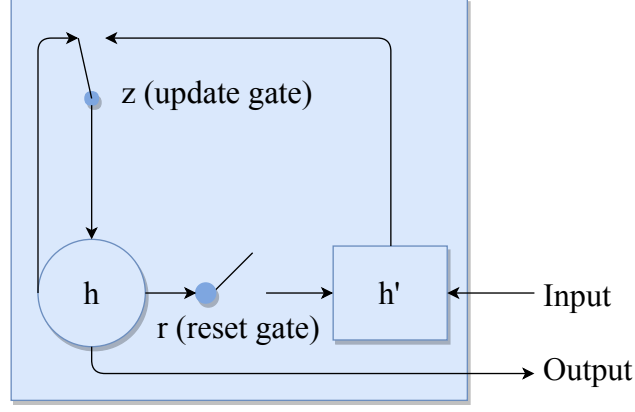


Figure 32.: Gated recurrent unit: The image shows a *gru* with two gates, r as a reset gate and z as an update gate. The activation of the *gru* is h and the proposed activation is h' [17].

computed using:

$$h'_t = \tanh(W \times x_t + U \times (r_t \odot h_{t-1})) \quad (36)$$

where h'_t is the current proposed activation, W and U are the input and recurrent weights matrices, r_t is the value of the reset gate and h_{t-1} is the previous step activation. The symbol \odot is an element-wise multiplication between r_t and h_{t-1} . \tanh is an activation function and is given as:

$$f(y) = \frac{e^y - e^{-y}}{e^y + e^{-y}} \quad (37)$$

The reset gate is given as:

$$r_t = \sigma(W_r \times x_t + U_r \times h_{t-1}) \quad (38)$$

where r_t is a reset gate which controls how much information from the previous activation should be used at step t . W_r and U_r are the input and recurrent weights matrices respectively. x_t is the current input at step t and h_{t-1} is previous step activation. An important point to note here is that an output does not only depend on current input but on previous inputs too (equation 34). A memory of previous inputs is maintained. It enables the gated recurrent units to extract latent features present in long sequences that lead to a specific output.

8.5. Network architecture

The recurrent neural network is used to classify paths and predict next tools. It consists of several layers. The first layer consumes paths and the last layer gives predictions. The hidden recurrent layers do all the computations required to do the classification of paths. While classifying, the paths are mapped to their respective next tools. Figure 33 shows how it predicts scores of tools of being next tools of a path.

8.5.1. Embedding layer

The first layer is an embedding layer. It learns a dense, fixed-length vector for each tool (figure 34e). An index (an integer) is assigned to each tool and it is converted to an embedding vector. It represents features associated with the tool. It means that an embedding vector for a tool encodes the information of the context in which the tool is being used. The tools which are used in similar context, their embedding vectors are similar.

8.5.2. Recurrent layer

Two hidden recurrent layers containing gated recurrent units are used. These layers are responsible for doing the computations given in equations 34-38. The hidden layers are stacked one upon another and they contain an equal number of gated recurrent units. Deeper features in paths are learned by stacking the recurrent layers. The hidden states of units in the first layer become inputs to the units of next hidden recurrent layer. Figure 31 shows these recurrent layers.

8.5.3. Output layer

An output is computed from the output (last) layer of the recurrent neural network. It is a dense layer having the same dimensionality as the number of tools. Each dimension holds a real number between zero and one which is a score assigned to a tool of being the next tool for any path.

$$f(x) = \frac{1}{1 + e^{-x}} \quad (39)$$

8.5.4. Activations

An activation is a function which transforms the input of a neuron (a node in a neural network) to an output. It can either be linear or nonlinear. There are multiple activation functions like *softmax*, *sigmoid*, *tanh*, *relu*, *linear* and many more². It is chosen based on the kind of output required for its further evaluation. The output of a neuron is given by:

$$y^j = \phi\left(\sum_{i=1}^N w_i \times x_i + b\right)^j \quad (40)$$

where y is an output of neuron j , w_i is the weight of the i^{th} input connection to the neuron j , x_i is an input and b is bias for neuron j . ϕ is an activation function. N is the number of input connections to neuron j . *Sigmoid* function (equation 39) is used as an output activation because the next tools are independent of one another and it learns the output of each tool separately. *Softmax* activation can replace it, but it normalises the exponents of output values which is undesired. *Sigmoid* assigns a real number between zero and one to each tool at the output layer. Another activation is exponential linear unit (*elu*). It is used as an activation for the recurrent layers. In equation 36, *tanh* is replaced by *elu* activation. It is given by:

$$f(x) = \begin{cases} x, & \text{if } x > 0 \\ \alpha \times (e^x - 1), & \text{if } x \leq 0, \alpha > 0 \end{cases}$$

It has a special feature of being negative when an input is negative. This feature allows mean activation (output) to get closer to zero compared to other activation functions like *relu* which is always positive. As mean activations get closer to zero, the approximated and actual gradients get closer to each other. Due to this, learning becomes faster and increased drop in error is achieved by [32].

8.5.5. Regularisation

Overfitting is a common phenomenon in the field of machine learning. It occurs when a classifier starts memorising training data without learning general features present in the data. It leads to an increased learning on training data but no learning on test data. The error on training data decreases but the error on test data either

²<https://keras.io/activations>

stops decreasing or sometimes increases. The classifier stops generalising and would predict new data with an increased uncertainty. A variant of a neural network is used in this work for classification. It is prone to overfitting as it tends to derive a complex model from training data. If the size of training data is small, it starts overfitting. To generate a model which learns general features, it is important to apply measures to remove or reduce overfitting. Regularisation is a technique to overcome this common problem. There are many techniques to regularise a neural network like dropping out random units (dropout) and decaying weights to stop them from becoming large. In this work, dropout is used as a regularisation method to tackle overfitting [33].

Dropout

Dropout removes connections momentarily from a neural network and thereby changing the network structure during each weight update. It randomly sets the output of some connections to zero and due to this, the network behaves in a novel manner. It becomes less powerful and stops picking bias from training data. A real number between zero and one is specified as dropout which is the fraction of units (memory units) to be dropped. It is applied to the input, output and recurrent connections of the recurrent neural network. The embedding layer also has a dropout layer for its output. The amount of dropout is a hyperparameter and a suitable value should be found out for which the drop in training and validation losses remains stable and close to each other [34, 35]. Moreover, the amount of dropout depends on the complexity of a neural network and the amount of data.

8.5.6. Optimiser

An optimiser is used to minimise error computed by an error function. Mini-batch *rmsprop* optimiser is used for the recurrent neural network in this work. It is a variant of stochastic gradient descent with an adaptive strategy for learning rate [36]. It adapts learning rate according to gradients. It keeps a track of previous gradients and updates learning rate by dividing with an average of the square of previous gradients. The weight update is given by the following equations:

$$MeanSquare(w, t) = 0.9 \times MeanSquare(w, t - 1) + 0.1 \times \left(\frac{\partial E}{\partial w}(t)\right)^2 \quad (41)$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{MeanSquare(w, t) + \epsilon}} \times \frac{\partial E}{\partial w}(t) \quad (42)$$

where $MeanSquare(w, t)$ is the mean of squared gradients until time t , $\frac{\partial E}{\partial w}$ is a partial derivative (gradient) of the error function with respect to parameter w at time t and η is learning rate. ϵ is a small number (for example, 10^{-5}) and is added to avoid division by zero.

Learning rate

Learning rate is an important parameter for an optimiser which determines the amount of update at each time step. If learning rate is high, there is a risk of an optimiser divergence. Instead of going to the minimum of error surface, which is the expected behaviour, the optimiser keeps oscillating on the error surface and bypass the minimum. On the other hand, if it is low, the optimiser may never reach or take a large amount of time to reach minimum as learning steps become too small. Due to these reasons, setting a good learning rate is a key to guarantee convergence in a reasonable amount of time.

Loss function

*Binary cross-entropy*³ is used as a loss function for *rmsprop*. It is well-suited for multilabel classification. It is a kind of classification of data with multiple outputs. For this work, there can be multiple next tools for a path. Therefore, multilabel classification is needed for this work. The loss function is given by:

$$loss_{mean} = -\frac{1}{N} \left(\sum_{i=1}^N y_i \times \log(p_i) + (1 - y_i) \times \log(1 - p_i) \right) \quad (43)$$

where N is the dimension of output layer (total number of next tool positions), y is actual next tool vector for a path and contains either zero or one, p is predicted next tool vector for the same path. The predicted next tool vector p contains real numbers between zero and one. The mean loss is a small number when actual and predicted next tool vectors are comparable. In equation 43, the sum is always negative or zero making the mean loss to be always positive or zero.

³https://github.com/keras-team/keras/blob/master/keras/backend/tensorflow_backend.py

8.5.7. Precision

Precision is computed for each path at the end of each training epoch. All the predicted next tools are matched either with actual or compatible next tools for a path and the correctly predicted next tools are computed. It is averaged over all paths to get an average precision for a training epoch. Two types of precision are computed, absolute and compatible. They are explained in section 10.1.

8.6. Prediction pattern

The recurrent neural network is designed to generate a probabilistic prediction of next tools for each path. There are n unique tools which are used to create workflows. It assigns a score to each tool of being the next tool of a path. The dimensionality of the output is n . Figure 33 shows these scores. The position of each score is assigned to a tool.

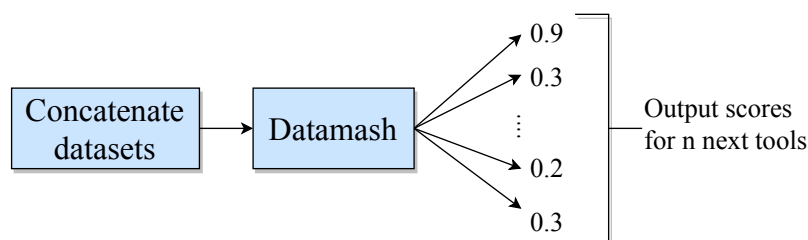


Figure 33.: Predicted scores for next tools: The image shows how the scores are assigned by the recurrent neural network to a set of n tools. The scores vary between zero and one. Each score contains an importance of each tool to become the next tool of a path. In the example, there are two tools in the path and the predicted scores of next tools are given. The predicted scores are sorted in descending order and a few top (for example top-1 or top-2) tools are selected.

9. Experiments

Workflows are collected from the Galaxy main¹ and Europe servers². There are $\approx 900,000$ paths in these workflows. Out of all these paths, $\approx 167,000$ paths are unique. The duplicates are removed. The recurrent neural network (*gru*) is trained on training paths (80% of all paths) and evaluated on test paths (20% of all paths). A small part of the training paths (20%) is reserved as validation paths. It is used to find the correct values of multiple parameters of the recurrent neural network. The loss is also computed on these validation paths. *BwForCluster*³ provides computing resources for preprocessing workflows and training and evaluating the recurrent neural network.

9.1. Decomposition of paths

The paths are decomposed by following the approaches explained in section 8.5.1. The performance is measured separately for each approach. In the first approach, no path is decomposed and the recurrent neural network is trained and tested on actual paths. In the second approach, only test paths are decomposed as described in figure 29 and the recurrent neural network is trained on actual paths. In the last approach, both the training and test paths are decomposed as described in figure 29. The recurrent neural network is trained using a mixture of smaller and longer paths. The configuration of the recurrent neural network is kept same for all the three approaches.

9.2. Dictionary of tools

Tool names present in paths cannot be used by any classifier. They need to be converted into numbers. To do that, a dictionary is created where each tool name is

¹<https://usegalaxy.org>

²<https://usegalaxy.eu>

³https://www.bwhpc-c5.de/wiki/index.php/Main_Page

assigned to an integer. The tool names are replaced by their respective indices from the dictionary and each path becomes a sequence of integers (figure 34a and 34c). In addition, a reverse dictionary is also created to replace any index by its tool name.

9.3. Padding with zeros

Training, test and validation paths have variable sizes. Some of these paths are short while some are long. But, the recurrent neural network can take only a fixed-size input. To deal with this issue, the maximum length of a path (number of tools) is set to 25 to captures all paths. Figure 27 shows that the number of tools in each path is less than 25. For shorter paths, a padding of zeros is added in the beginning to ensure that all paths have the same length. Moreover, it is undesirable that the recurrent neural network learns any feature from these padded zeros. To avoid this, a flag is set (in the implementation of the embedding layer) to mask⁴ these streams of zeros. Due to this, the recurrent neural network considers only the useful indices (> 0) in a path. As zero is chosen for padding, it does not represent any tool in the dictionary.

Figure 34 shows how a path and its possible next tools are transformed into their vectors as shown in figures 34d and 34e respectively. They are used by the recurrent neural network for training and evaluation. Figure 34a shows a path with three tools arranged in an order as it would appear in any workflow. Figure 34b shows next tools for the path. Section 9.2 explains that each tool is represented by an integer and is shown in figure 34c. The length of the vector in 34c is 25. The padding of zeros is also shown followed by the corresponding indices of tools (figure 34c). In each vector, the padding of zeros precedes the sequence of tool indices. Each index is further transformed into a fixed-length vector (embedding) as shown in figure 34e. The length of this embedding vector remains same for all tools and is defined by the size of embedding layer (512). 512 dimensional vector is learned for each tool's index and arranged as shown in figure 34e. The order of tools is maintained as present in an original path as shown in figure 34a. Figure 34d shows how the next tools vector is arranged. A vector of zeros with a size equal to the number of tools is taken. The corresponding indices of next tools are set to one in this vector. For example, *addvalue* tool has an index 4 in the dictionary. Therefore, the fourth position of next tool vector is set to one (figure 34d). This is repeated for all next tools for a path.

⁴<https://keras.io/layers/embeddings>

Figure 34d contains two positions which are set to one representing next tools for the path shown in figure 34a.

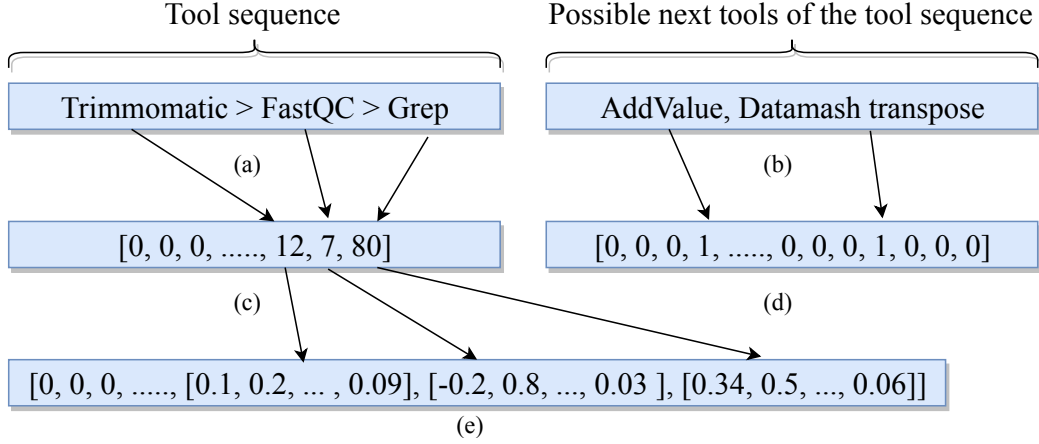


Figure 34.: Vectors of a path and its next tool: The image shows how vectors are created for a path and its next tools in order to make them available for the recurrent neural network. Figure 34a shows a path with three tools. 34b shows its next tools. Figure 34c shows the arrangement of padding of zeros along with the indices of tools in a fixed-length vector (25). Figure 34d shows how the next tools vector is arranged. The corresponding positions of next tools are set to one and the rest of the positions are set to zero. In figure 34e, each index belonging to a tool is transformed into a fixed-length embedding vector. This is done by the embedding layer of the recurrent neural network. The size of embedding layer defines the length of this embedding vector.

9.4. Network configuration

The gated recurrent unit, a variant of the recurrent neural networks, is used as the classifier. It has several layers and many hyperparameters. An embedding layer is used as an input layer which learns a fixed-size vector (512) for each index belonging to a tool. The number of unique tools in all the workflows is 1,800. As the number of tools increases, it is important to increase the size of embedding layer. Dropout is applied to the output of this layer in order to reduce overfitting. Two hidden recurrent layers with gated recurrent units are used to model paths. Each layer has a fixed number of memory units. The number of memory units specifies the dimension of hidden state (equation 34). Dropout is applied to the inputs, outputs and recurrent connections for each hidden recurrent layer. The amount of dropout

remains same everywhere in the recurrent neural network. The last layer is a dense layer and has a size equal to the number of tools. This is because, for each path, the recurrent neural network generates scores for all tools to be the next tool of a path. The tools having higher scores at the output layer possess higher probability of being the next tools. The learning saturates and training and validation losses do not decrease anymore after 40 epochs. It means no more learning is possible and training should be stopped. The learned model saved for the last epoch is used for making the predictions of test paths.

9.4.1. Mini-batch learning

In mini-batch learning, a smaller set from training paths is chosen to update the weights. Multiple candidate values (batch size) like 64, 128, 256 and 512 are chosen to see the effect on the loss and precision while keeping all other parameters fixed. Its value is set to 512 for the baseline recurrent neural network (section 10). It is allowed to learn on training paths over multiple epochs. An epoch consists of multiple iterations and in each epoch, all the training paths are used. In each iteration, 512 paths are used to approximate the weight update. If there are 2560 (512×5) training paths, then 5 iterations will make one epoch with 512 as batch size. The recurrent neural network is sensitive to this number as a small number can add a lot of noise to the weight update [37].

9.4.2. Dropout

For setting dropout, different numbers are used to find the best one. The values like 0.0 (no dropout), 0.1, 0.2, 0.3 and 0.4 are used while keeping all other parameters fixed. The loss on validation paths is a key indicator of overfitting.

9.4.3. Optimiser

A few optimisers like *stochastic gradient descent (sgd)*, *adam*, *rmsprop* and *adagrad* are used to find which one provides a stable learning and enables the recurrent neural network to achieve the best precision. All other parameters are kept fixed. They all minimise cross-entropy loss between actual and predicted next tools. The default configurations of these optimisers are used as set by the *keras* library⁵.

⁵<https://github.com/keras-team/keras/blob/master/keras/optimizers.py#L209>

9.4.4. Learning rate

Different learning rates are used to find a stable learning pattern by the recurrent neural network. Their values are 0.0001, 0.005, 0.001 and 0.01. A small learning rate tends to slow down the convergence of the optimiser while a high rate can diverge it. It is important to avoid both the situations to ensure a stable learning. All other parameters are kept fixed.

9.4.5. Activations

There are multiple choices of activation functions like *tanh*, *sigmoid*, *relu* and *elu*. *tanh* is the default activation⁶ while *elu* is one of the recently proposed activations. All other parameters are kept fixed.

9.4.6. Number of recurrent units

The hidden recurrent layers need memory units (dimensionality of hidden state) to learn the mapping of paths to their respective next tools. Several sizes like 64, 128, 256 and 512 are chosen to find which one expresses the hidden states to ensure better learning. All other parameters are kept fixed.

9.4.7. Dimension of embedding layer

This dimension specifies the length of the dense vectors learned for each tool. Various sizes like 64, 128, 256, 512 and 1024 are used to see the effect on the precision and loss. All other parameters are kept fixed.

⁶<https://keras.io/layers/recurrent/#gru>

10. Results and analysis

In this section, the performance of the recurrent neural network is visualised and discussed for three different approaches of decomposition of paths (section 8.4.1). Different values of various parameters like optimiser, learning rate, activation, batch size, number of memory units, dropout, embedding dimension and length of paths are used. The performances of different values for each parameter are compared. Top-1 and top-2 accuracies are computed and compared. A slightly different neural network with only dense layers is used too to find whether it performs better than the recurrent neural network. The values of the parameters used by the recurrent neural network are as follows:

- Number of epochs is 40
- Batch size is 512
- Dropout is 0.2
- Number of memory units is 512
- Dimension of embedding layer is 512
- Maximum length of paths 25
- Training paths are 80% and test paths are 20% of the complete set of paths
- Validation set is 20% of training paths
- Optimiser is *rmsprop*
- Output activation of recurrent layer is *elu*
- Output activation of dense layer is *sigmoid*
- Loss function is *binary cross-entropy*

These parameters remain same for all three approaches and define a baseline configuration of the recurrent neural network. By experimenting with different values of these parameters, the possibilities are explored to improve performance.

10.1. Notes on plots

For the plots in figures 35-44 and 47-48, there are few common points to note:

- The x-axis shows the number of training epochs. For subplots (a) and (b), the y-axis shows an average precision and for subplots (c) and (d), the y-axis shows cross-entropy loss.
- The subplot (a) shows absolute precision. If a path has five actual next tools, top five predicted next tools are taken out from using predicted model. If out of the five predicted next tools, only four of them match the actual next tools, then the absolute precision for this path is $\frac{4}{5}$. The average precision is computed over test paths.
- The subplot (b) shows compatible precision. While computing absolute precision, some false positives (wrong next tools) are also predicted which are not present in the actual next tools for a path. In this case, these false positives are matched with the compatible set of tools belonging to the last tool of the path. If some or all of the false positives are present in this compatible set, then they add up to absolute precision to give compatible precision. For example, in the last point, there is one false positive out of the five predicted next tools. This false positive is checked in the compatible set of the last tool of the path. If it is present, then compatible precision becomes 1.0. If not, it stays equal to the absolute precision ($\frac{4}{5}$). The compatible precision is at least as good as the absolute precision.
- The subplot (c) shows the cross-entropy loss on training paths. It is computed by the recurrent neural network as shown in equation 43.
- The subplot (d) shows the cross-entropy loss on validation paths. The last 20% of training paths are validation paths. This loss is computed by the recurrent neural network after training on the first 80% of training paths.

10.2. Performances of different approaches of path decomposition

10.2.1. Decomposition of only test paths

In this approach, paths in the test set are decomposed keeping the first tool fixed as explained in figure 29. The training paths are kept as such. The results are shown in figure 35. The idea is to make the recurrent neural network learn tools connections using longer paths and predict next tools for shorter and longer paths. But, as the result suggests, the learning does not happen as desired. Absolute precision ($\approx 22\%$) is worse than the compatible precision ($\approx 43\%$) achieved at the end of training as shown in figures 35a and 35b. However, the cross-entropy loss for training and validation paths drops in a steady manner and saturates as shown in figures 35c and 35d. The learning happens but only for training paths and not for test paths. The overall training and evaluation time was ≈ 28 hours. Each epoch took ≈ 20 minutes for training. The number of training paths was $\approx 96,000$ while the number of test paths was $\approx 62,000$. But, only half of the test paths ($\approx 31,000$) were used for evaluation.

10.2.2. No decomposition of paths

In this approach, paths are used as such for both, training and test paths. The recurrent neural network is made to learn and evaluate on longer paths. The last tool in each path is used as its next tool. The results in figure 36 are encouraging. Subplot 36a reaches a precision of $\approx 89\%$ at the end of training. The compatible precision is $\approx 98\%$ and is much better than the previous approach. The validation loss (figure 36d) drop is steady and comparable to the training loss (figure 36c). The overall training and evaluation time was ≈ 22 hours. Each epoch took ≈ 20 minutes for training which is same as the previous approach because the number of training paths remains almost same. Overall, it took less time compared to the previous approach because of the smaller size of test paths. The number of training paths was $\approx 90,000$ while the number of test paths was $\approx 22,000$.

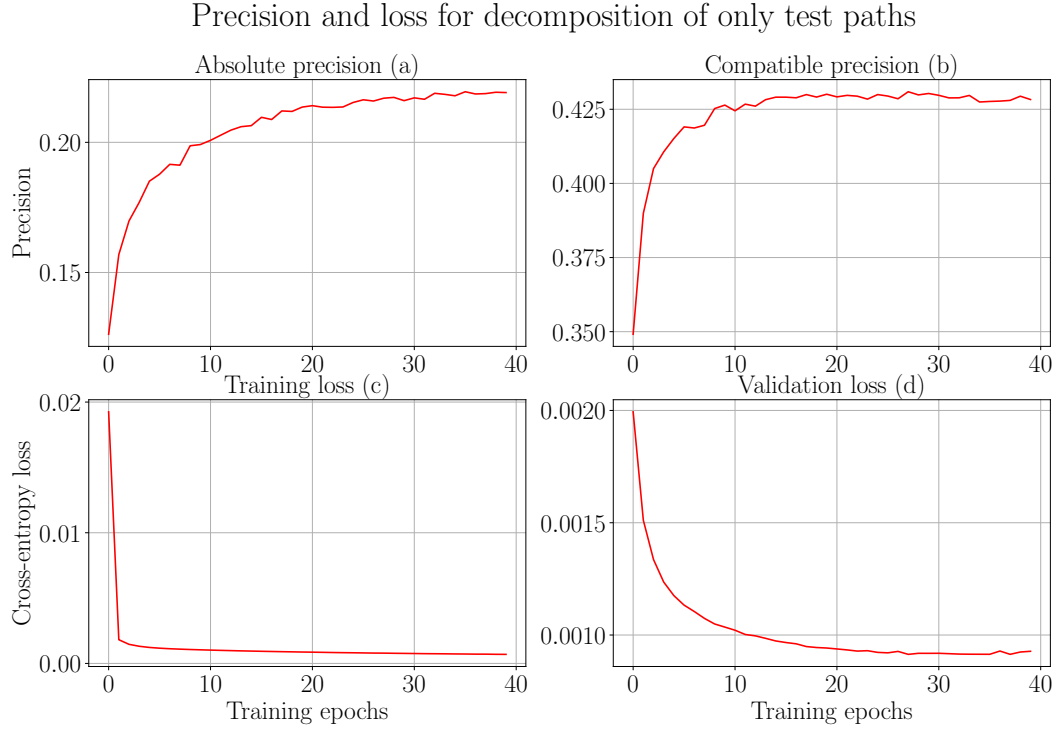


Figure 35.: Performance of the decomposition of only test paths: The plot shows classification performance of an approach where only test paths are decomposed keeping the first tool fixed. The training paths are kept as such. The idea is to train a recurrent neural network on longer paths and test on shorter and longer paths. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

10.2.3. Decomposition of the train and test paths

All paths are decomposed keeping the first tool fixed. The idea here is to make the recurrent neural network learn and predict on shorter as well as on longer paths. The results are shown in figure 37. Absolute precision is $\approx 89\%$ while compatible precision is $\approx 99\%$. The results are comparable to the previous approach. To create a workflow, a tool is chosen and its next tools are predicted. Again, one more tool is added to the previous tool and using these two connected tools, the next tools are predicted and so on. Therefore, this approach is more practical than the previous approach. The paths are decomposed to have many shorter paths. Due to this, size of the training paths increases which means that more training time is required. The training along with the evaluation of test paths finished in ≈ 48 hours. The training

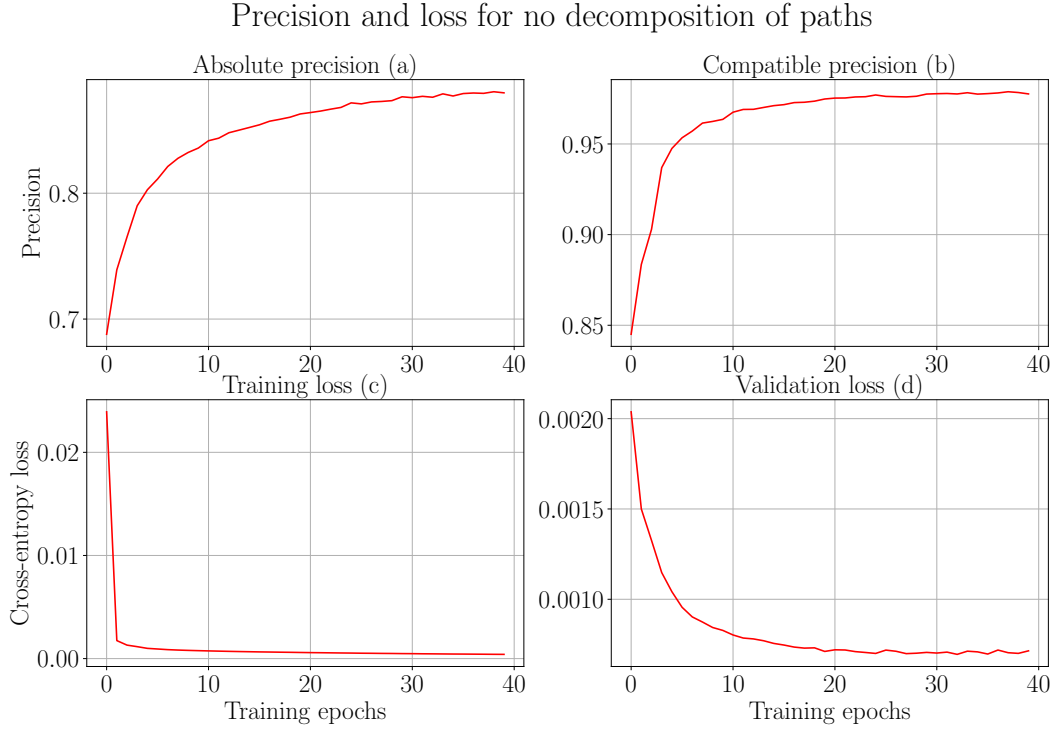


Figure 36.: Performance of no decomposition of paths: The plot shows classification performance of an approach where no paths are decomposed. The idea is to train and test the recurrent neural network on longer paths. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

for each epoch took ≈ 45 minutes. The number of training paths was $\approx 168,000$ while the number of test paths was $\approx 42,000$.

10.3. Performance evaluation on different parameters

The recurrent neural network has many parameters and they need to be tuned to the amount of data and to each other so that it learns and predicts in a reliable way. Their right combination is needed to attain a reasonable accuracy and to avoid too much (overfitting) and too less learning (underfitting). These parameters include optimiser, learning rate, activation, batch size, number of recurrent (memory) units, dropout and dimension of embedding layer. There are a few more metrics on which

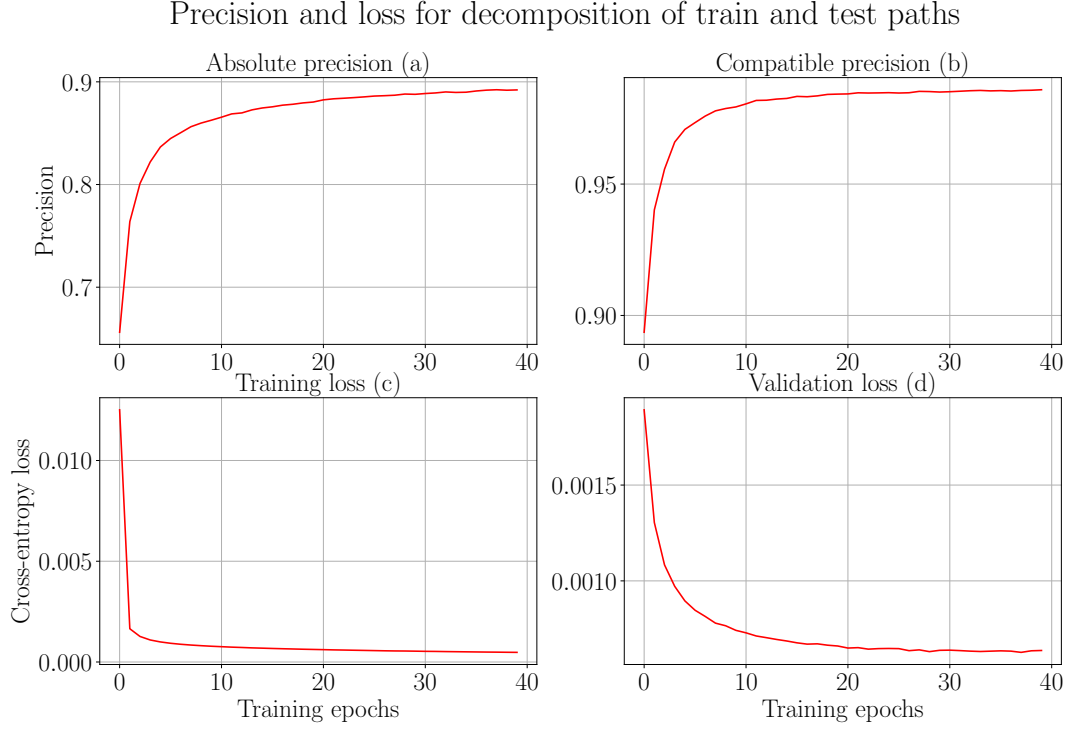


Figure 37.: Performance of the decomposition of all paths: The plot shows classification performance for an approach where all the paths are decomposed keeping the first tool fixed. The idea is to make the recurrent neural network learn and predict on shorter as well as longer paths. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

the evaluation is done. These include top-k accuracy and length of paths. A deep network with only dense layers is also used as a classifier to compare classification performance with the recurrent neural network.

10.3.1. Optimiser

Four optimisers, *stochastic gradient descent (sgd)* [36], *adaptive sub-gradient (adagrad)* [36], *adam* [38] and *root mean square propagation (rmsprop)*, [36] are used for the analysis to find which one achieves the highest precision. All other parameters are kept constant. From figure 38, it is concluded that the *sgd* performs worst on both the metrics, precision and loss. The absolute and compatible precision do not improve and drop in the loss curve starts very late during training. It starts off

with a high value (0.65) and does not drop much within the 40 epochs of training. Out of the remaining three optimisers, *rmsprop* performs best. The performance of *adam* is comparable to *rmsprop*. The adam optimiser catches up with the precision measured by *rmsprop*, but slowly. Their performances converge later in the training. The plot shows that the choice of *rmsprop* as an optimiser for the baseline network is beneficial for learning. In general, *rmsprop* is a good choice for the recurrent neural networks [39]. The bad performance of the *sgd* may be attributed to its non-adaptive parameter update method which does not adapt to the gradients. The adaptive optimisers (*rmsprop* and *adam*) adjust parameter update with the gradients of previous steps.

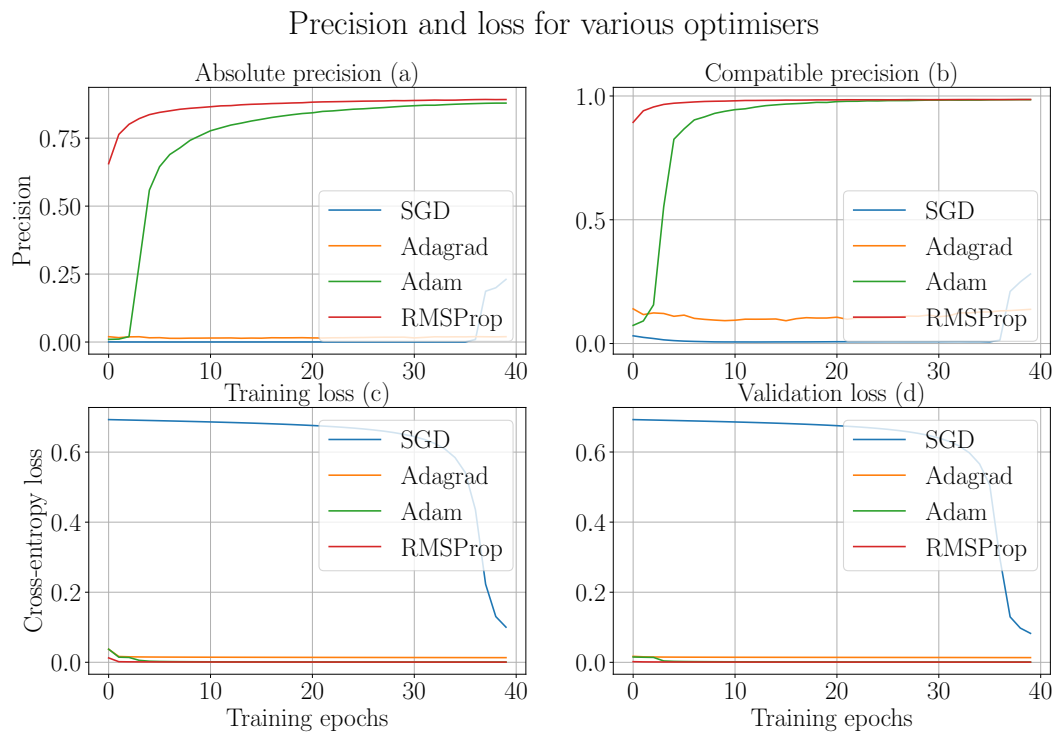


Figure 38.: Performance of different optimisers: The plot shows the performance of different optimisers. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. The four optimisers are the *stochastic gradient descent* (*sgd*), *adaptive sub-gradient* (*adagrad*), *adam* and *root mean square propagation* (*rmsprop*). All other parameters of the network are kept constant.

10.3.2. Learning rate

Multiple values of learning rate from small (0.0001) to high (0.01) are used to find which one achieves the highest precision. A performance comparison of different learning rates while keeping other parameters constant is shown in figure 39. A higher learning rate (0.01) diverges the optimiser as precision drops during training. The training loss increases and its curve has sharp edges (figure 39c). For the validation loss as well (figure 39d), there is no stable pattern (continuous drop). These situations are undesirable and they inform that the learning rate should be kept smaller. A smaller value of 0.005 works better than the previous one but not completely because precision drops slightly towards the end of training. The smaller values 0.001 and 0.0001 help in learning as the precision improves and loss drops during entire learning in a steady way. The value 0.0001 ensures good learning but it is slower and would need more epochs (and more time) to converge. The value 0.001 works best out of all these values on both the metrics, precision and loss.

10.3.3. Activation

Many activation functions like *tanh*, *sigmoid*, *relu* and *elu* are used to find which one achieves the highest precision. Figure 40 shows the performances of these different activation functions. The activation functions *tanh*, *relu* and *elu* perform better than *sigmoid* on both the metrics, precision and loss. The activation functions *relu* and *elu* perform close to each other for precision as well as loss and are better than *tanh* and *sigmoid*.

10.3.4. Batch size

Different numbers like 64, 128, 256 and 512 are used as the mini-batch size to find which one achieves the highest precision. In mini-batch optimisation, a number of paths of size equal to the mini-batch size are taken and an average weight update is computed using these paths. This average update approximates update for the complete set of training paths. A smaller number adds more noise to the update as it captures less variance of the complete set. The performance of various batch sizes is shown in figure 41. It is deduced from the plot that the batch size of 64 does not perform well. Figure 41c shows that the training loss starts increasing instead of decreasing. This proves that 64 is not the right choice of batch size. As batch size is increased, the precision improves and the loss drops. The drop is higher for the

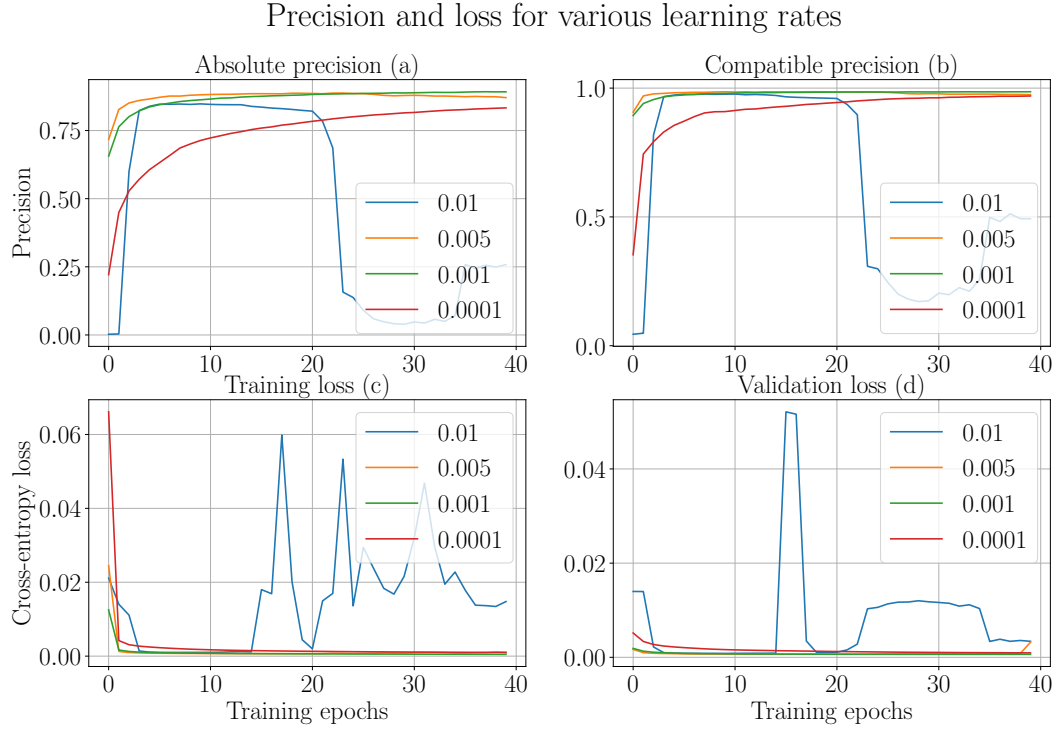


Figure 39.: Performance of multiple learning rates: The plot shows the performance of different values of learning rate. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. High learning rates do not provide stable learning while a small one slows down the learning.

batch size of 512 for validation loss compared to training loss. The baseline network uses 512 as batch size. A batch size of 256 also looks promising.

10.3.5. Number of recurrent units

Four different numbers for recurrent units are used to find which one achieves the highest precision. It specifies the dimensionality of the hidden state. The higher the number, the more expressive the model becomes. It means that the model's prediction strength or "memory" of the model increases. This behaviour can be seen in figure 42. As the number of units increases, precision becomes better and the loss drop is more. 512 number of units performs the best out of the four choices of the number of recurrent units. But, the increasingly strong model tends to overfit and tries to memorise training paths. Combating overfitting is necessary when the

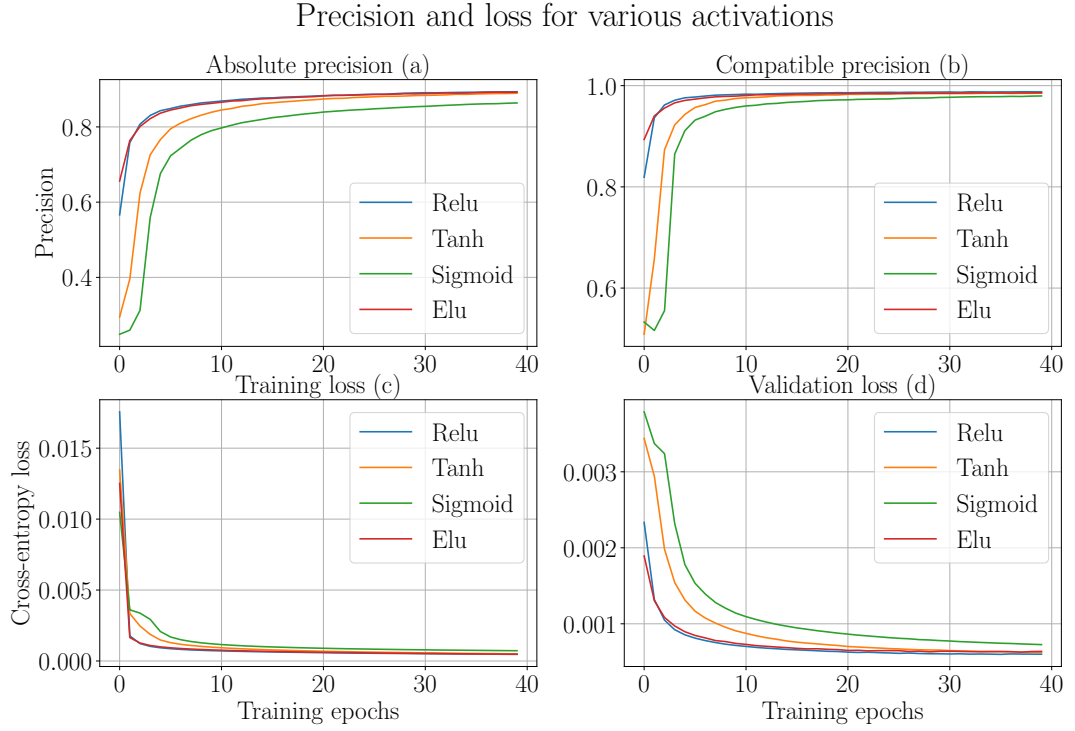


Figure 40.: Performance of multiple activation functions: The plot shows the performance of different activation functions. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

network becomes strong. Using a higher number of memory units also increases the training time. With 64 as the number of memory units, training time for each epoch was ≈ 6 minutes while with 512 memory units, each epoch took ≈ 45 minutes.

10.3.6. Dropout

Dropout is used as a measure to overcome overfitting. To improve learning, the recurrent neural network is made stronger by adding more number of memory units and two hidden recurrent layers. Five different values of dropout are used to verify which one combats overfitting while keeping the performance high on unseen paths. Figure 43 shows the performance of different values of dropout. When no dropout (0.0) or smaller dropout (0.1) is used, the validation loss starts increasing while training loss still decreases. The precision starts decreasing for these values of dropout. It is a clear sign of overfitting. When a higher value (0.4) is used, the recurrent neural network becomes weaker and due to this, the precision increases

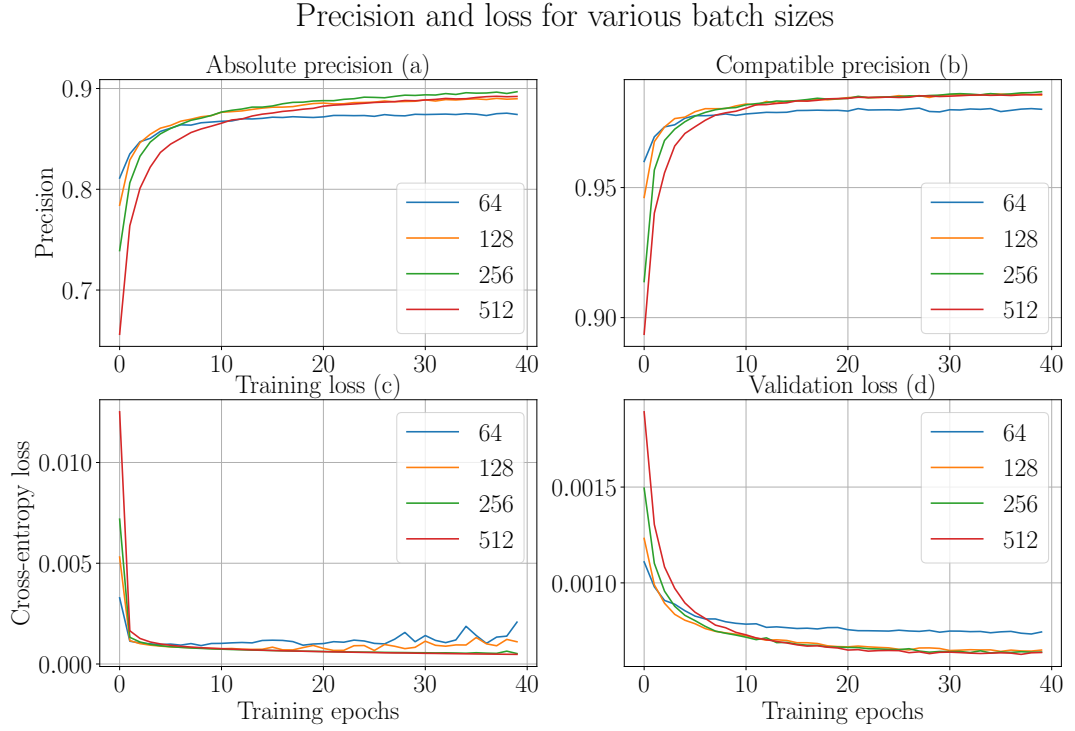


Figure 41.: Performance of different batch sizes: The plot shows the performance of different batch sizes. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. A batch size of 64 does not perform well compared to large batch sizes (256 and 512).

slowly. 0.2 and 0.3 give better choices of dropout as they achieve better precision and the drop in the training and validation losses are more robust. The baseline network configuration uses 0.2 as a dropout. A dropout 0.3 performs close to 0.2 and can be used as well. The higher the value of dropout, the larger is the training time. Using no dropout took ≈ 44 hrs for training and evaluation while with 0.3 dropout, it took ≈ 48 hrs [40].

10.3.7. Dimension of embedding layer

Embedding layer learns a fixed-length, unique dense vector for each tool. The larger the size of this layer, the higher is its expressive power to distinguish among tools. But, with higher dimensions, there is a risk of overfitting and with a smaller dimension, underfitting can occur. Figure 44 shows the performance with different

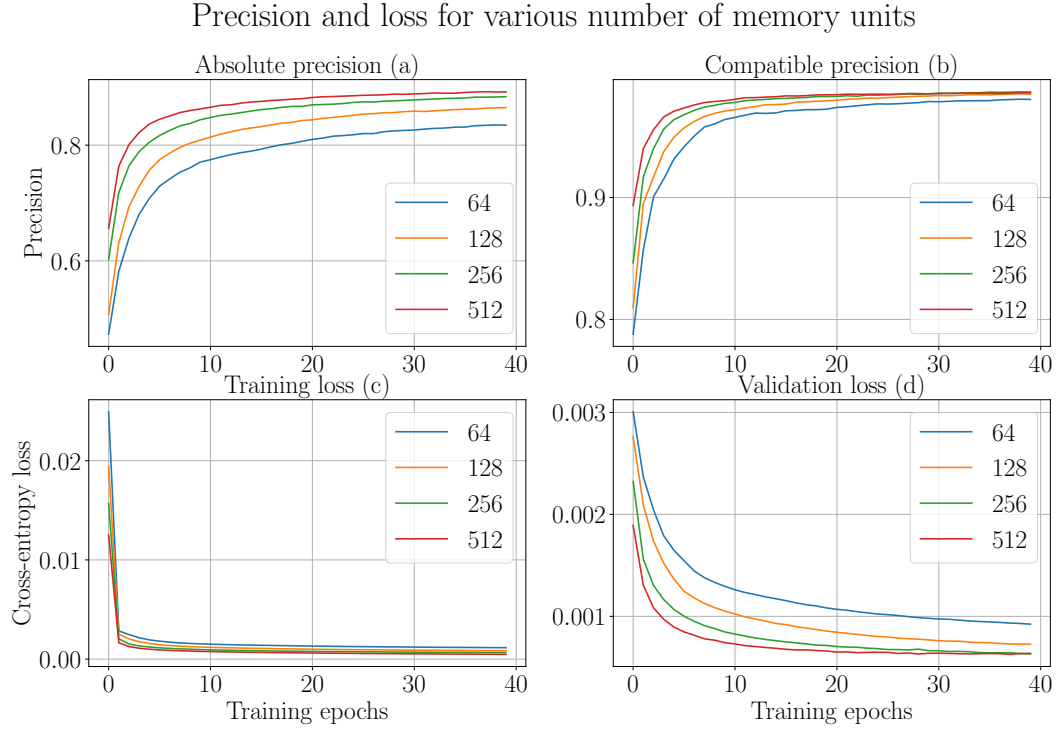


Figure 42.: Performance of multiple values of memory units: The plot shows the performance of different memory units. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. 64 memory units for each recurrent layer do not perform well compared to a larger number like 256 or 512. The training time increases with the number of memory units.

sizes of embedding layer. All these sizes perform close to one another. A larger size performs slightly better than a lower size. The size 1,024 performs the best while 64 also performs close especially for the training loss (figure 44c). The training time increases with the size of embedding layer. For size 64, it took ≈ 30 minutes for training one epoch while with 1,024, it took ≈ 45 minutes.

10.3.8. Accuracy (top-1 and top-2)

Absolute and compatible top-k accuracies are computed for training and test paths (figure 45). k is an integer and satisfies $k \geq 1$. All predicted next tools are sorted in descending order of their scores and then, top-k tools are extracted. For example, the absolute top-k accuracy computes how many of the k predicted next tools are

Precision and loss for various dropout values

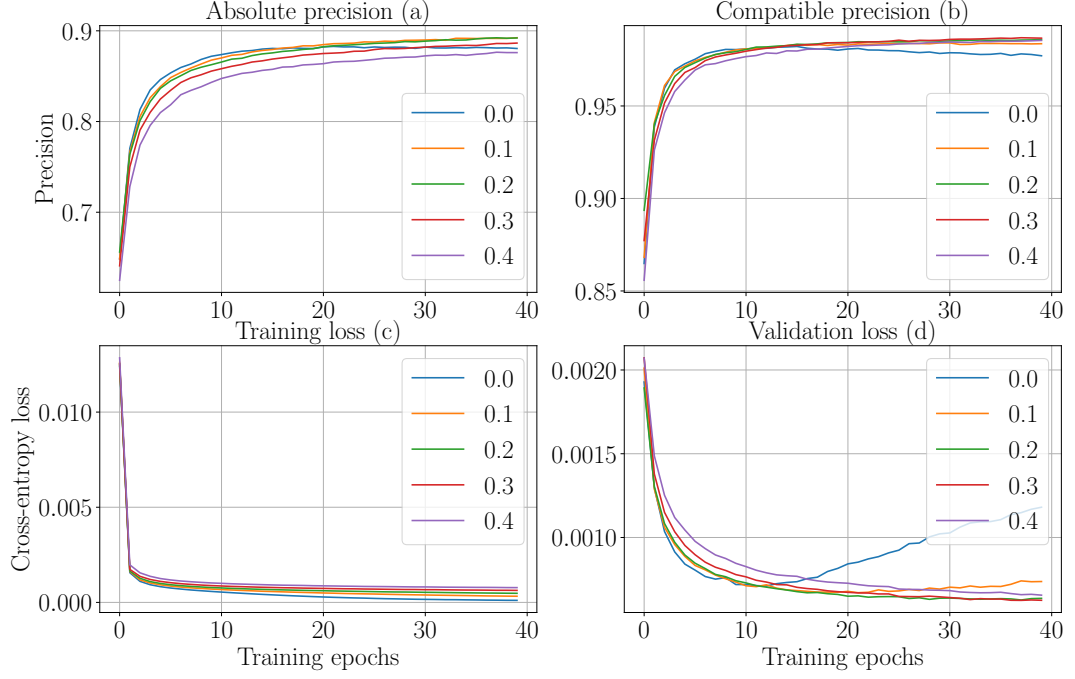


Figure 43.: Performance of multiple values of dropout: The plot shows the performance of different values of dropout. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. No dropout shows overfitting as the validation loss increases during training (d) while a higher dropout (0.4) achieves lower precision.

present in the set of actual next tools of a path. Top-1 (absolute and compatible) and top-2 (absolute and compatible) accuracies are computed and they averaged for training and test paths. A similar approach is followed for computing compatible top-k accuracy.

Figure 45 shows that the performances of training and test paths remain similar for the absolute and compatible top-1 and top-2 accuracies. To compute absolute top-1 accuracy for a path, the next tool with the highest score is selected and checked whether it is present in the set of actual next tools. This accuracy is averaged for all the paths. To compute compatible top-1, the next tool with the highest score is checked if it is present in the set of compatible next tools (of the last tool of a path). The compatible top-1 and top-2 accuracies are higher than that of the absolute top-1 and top-2, respectively. It shows that the recurrent neural network learns tool connections from multiple paths and use them in prediction. Top-1 achieves a

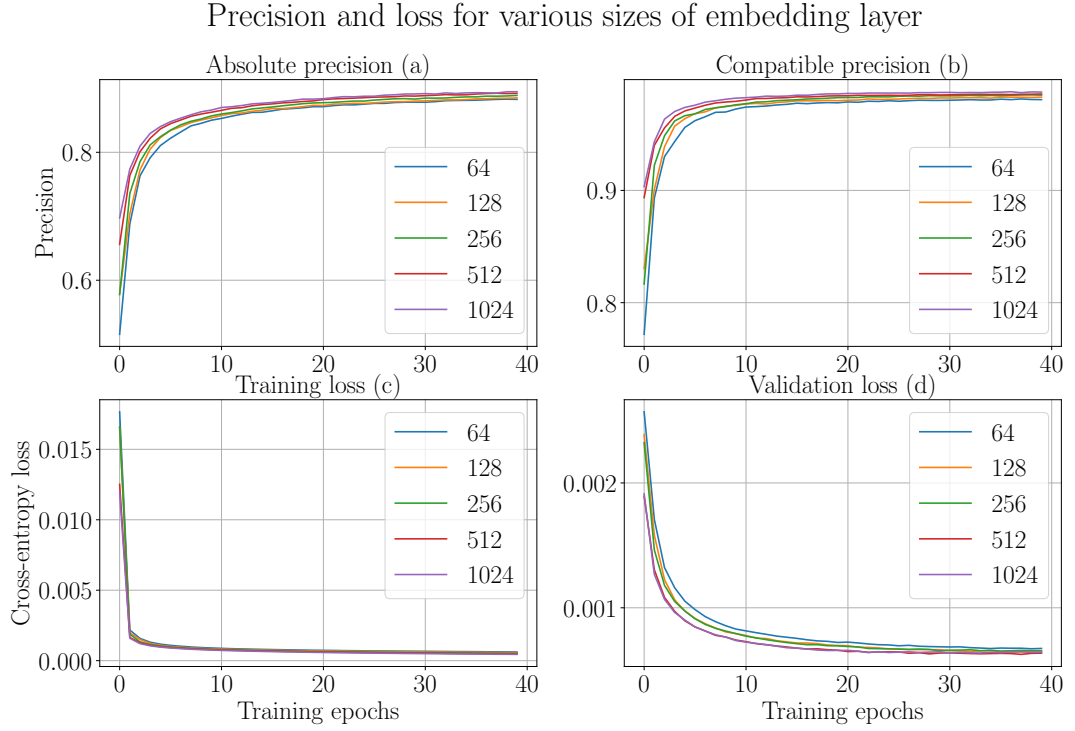


Figure 44.: Performance of different dimensions of embedding layer: The plot shows the performance of different dimensions of embedding layer. The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively. All the sizes perform close to one another.

higher accuracy than top-2. There is a severe drop in the absolute top-2 accuracy for training and test paths. It is because many paths have just one next tool (actual next tool) in workflows. The compatible top-1 and top-2 accuracies remain high ($\geq 90\%$) for the training and test paths.

10.3.9. Precision with the length of paths

The variation of precision (absolute and compatible) with the length of paths is shown in figure 46. The precision of predicting next tools for paths containing the same number of tools is averaged for training and test paths.

As the maximum length of a path is fixed to 25, the plot shows the maximum length as 24 and the last tool (at 25th position) is used as the next tool for each path of length 25. It is concluded from the plot (figure 46) that as the length of paths increases, precision becomes better. This increase is more dominant for compatible

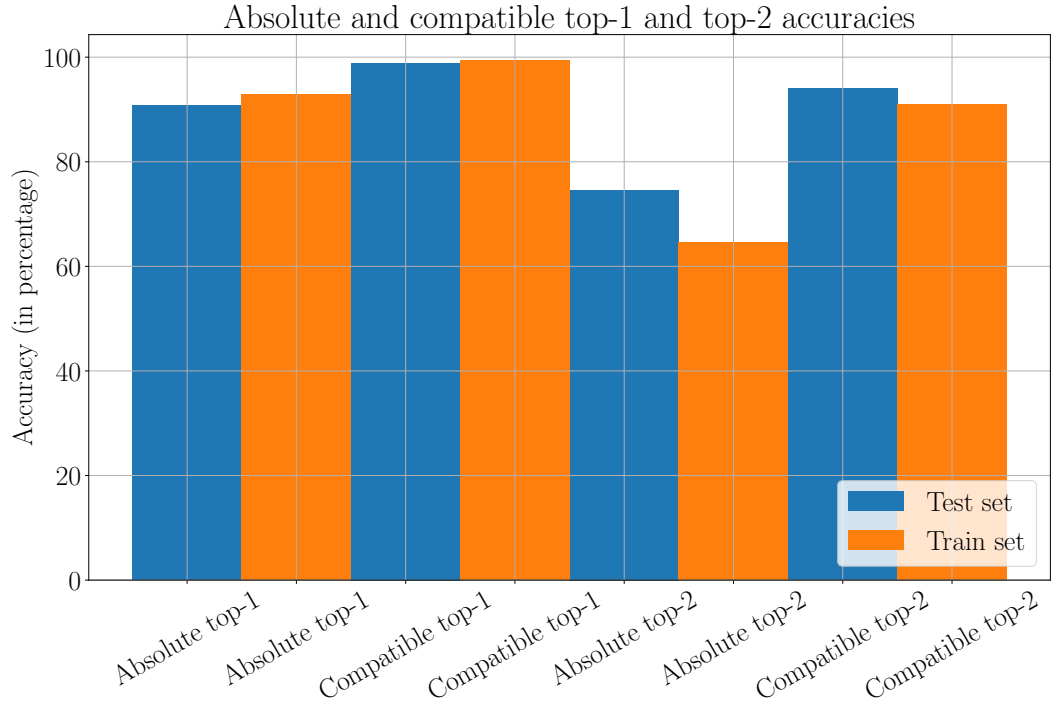


Figure 45.: Absolute and compatible top-1 and top-2 accuracies: The bar plot shows the absolute and compatible top-1 and top-2 accuracies. It also shows that the training and test paths achieve comparable performance.

precision than for absolute precision. As the length of paths increases, they have more tool connections and thereby more features. A higher number of features present in longer paths helps in predicting next tools more robustly. This is achieved even though the number of paths with length ≥ 20 is lower compared to the number of shorter paths.

10.3.10. Performance with a small number of workflows

An analysis is done with a small number of workflows ($\approx 9,000$ paths). Figure 47 shows the results. The subplots (a) and (b) show that the absolute and compatible precision are not comparable to what is achieved using a large number of workflows with $\approx 167,000$ paths (figure 37). The absolute and compatible precision saturate around $\approx 70\%$ and $\approx 85\%$ respectively. As the number of workflows is less, a weaker recurrent neural network with 256 memory units is used. A dropout of 0.2 is used to prevent overfitting. The analysis is executed for 100 epochs. After the 50th epoch,

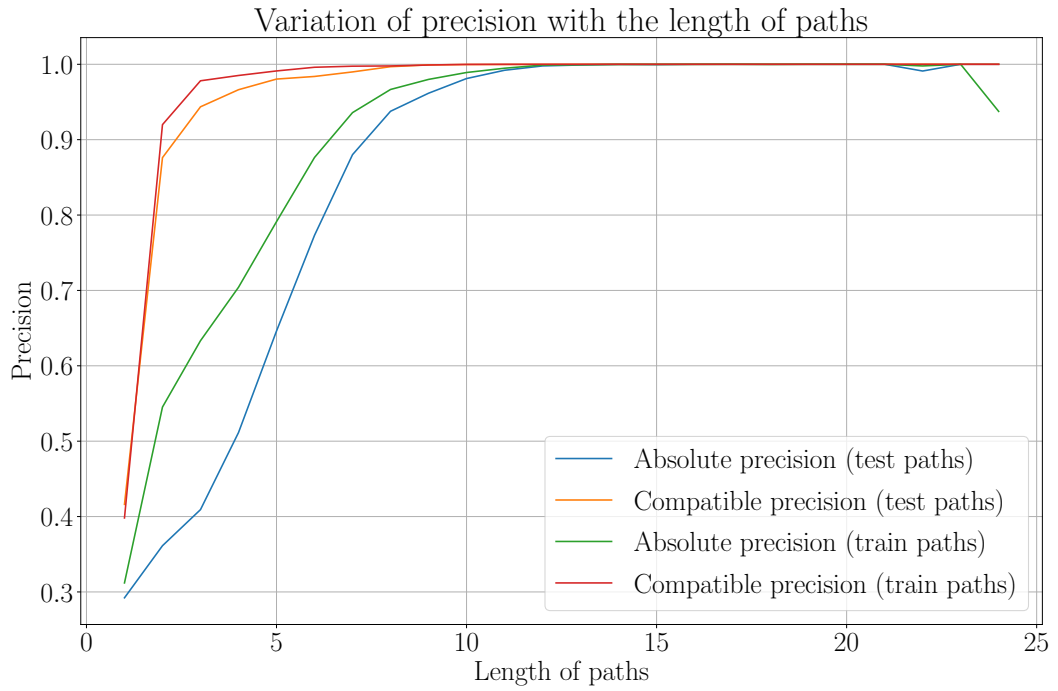


Figure 46.: Variation of precision with the length of paths: The plot shows how the length of paths affects precision. As the length of paths increases, the precision becomes better. This improvement is more dominant for compatible precision compared to absolute precision.

the validation loss starts increasing which proves that the network is overfitting. A recurrent neural network with 128 memory units is also used but the performance is worse than shown in figure 47. It is concluded that a strong recurrent neural network is required to achieve better precision, but it needs a large amount of data.

10.3.11. Neural network with dense layers

A neural network with only dense layers is used as a classifier to compare the performance of learning on sequential data with the recurrent neural network. Two hidden layers are used with 128 neurons each. The first layer is an embedding layer and the last (output) layer is a dense layer. The dropout (0.05) is applied to combat overfitting. Rest all parameters like the optimiser, learning rate, batch size, embedding dimension, activations and loss function remain the same as for the recurrent neural network. The training and test paths are decomposed in the same way as explained in section 10.2.3. The network is trained for 40 epochs. The

Precision and loss using less data

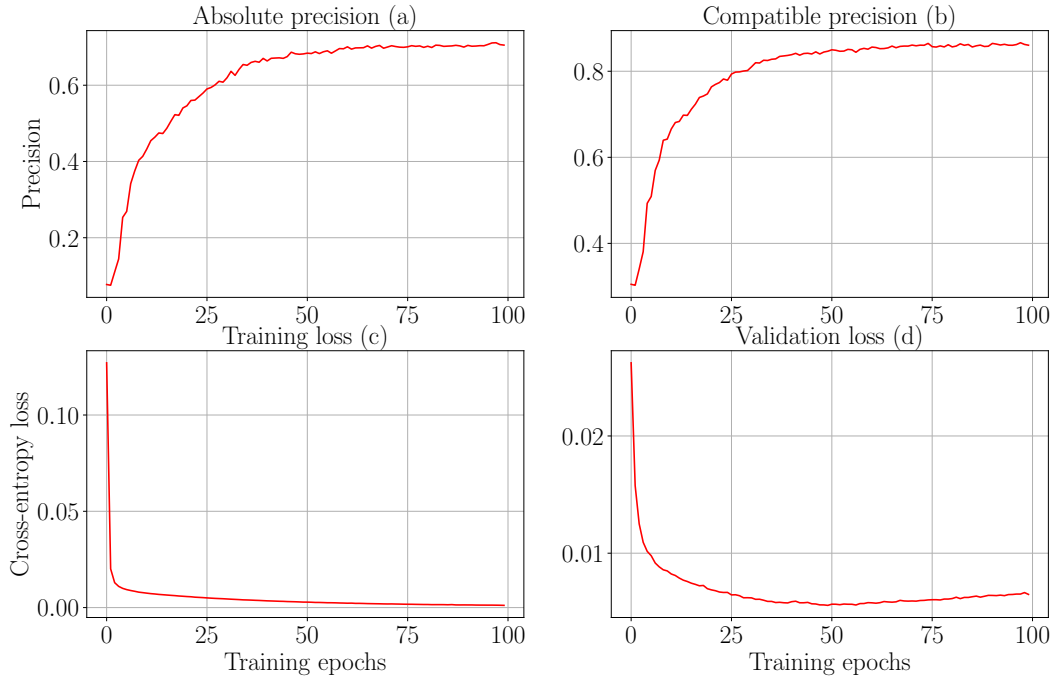


Figure 47.: Performance with a small number of workflows: The plot shows the performance of the recurrent neural network with a small number of workflows. The paths are decomposed keeping the first tool fixed (as described in section 10.2.3). The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

precision (absolute and compatible) is computed after each training epoch. The training and validation losses are also noted. Figure 48 shows the performance of this network.

Subplots 48a and 48b suggest that this network with dense layers also starts performing well (earlier in the epochs). It reaches an absolute precision of $\approx 83\%$ and compatible precision of $\approx 97\%$ around the 15th epoch. The learning seems to be good in the beginning, but it starts becoming worse towards the end of training. The precision starts becoming worse. Moreover, the training and validation losses (figure 48c and 48d) start increasing. They collectively suggest that the network is overfitting. The neural network with recurrent layers performs better than this neural network with dense layers. It is concluded that the neural networks with recurrent layers are more suited for learning on sequential data than the neural networks with only dense layers.

Precision and loss using neural network with dense layers

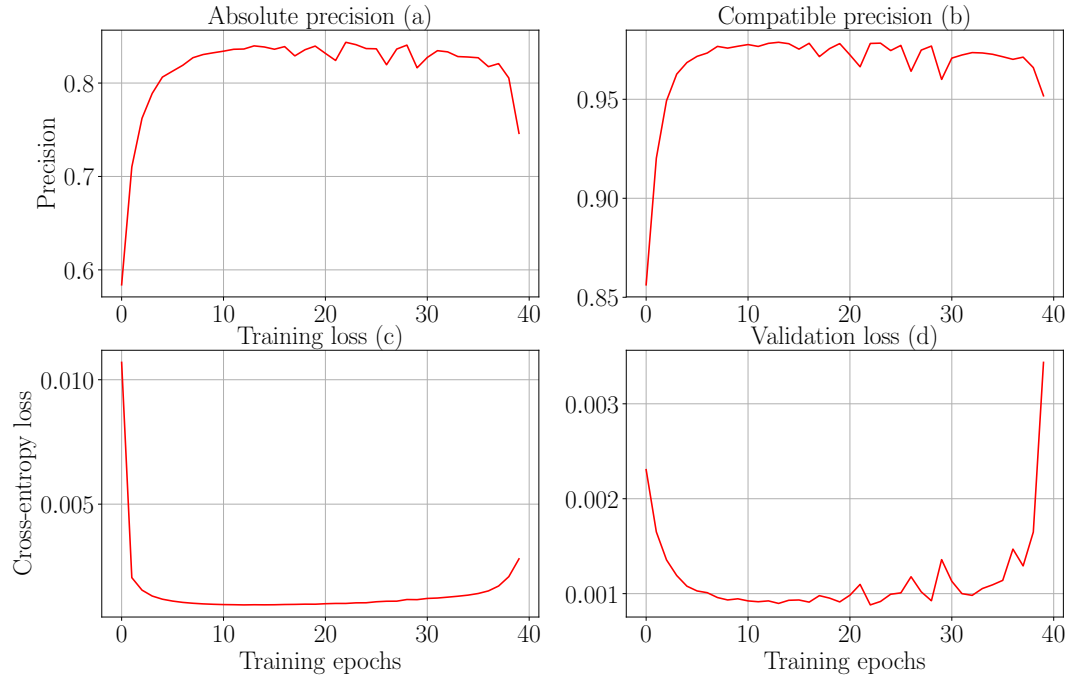


Figure 48.: Performance of a neural network with dense layers: The plot shows the performance of a neural network with dense layers. The paths are decomposed keeping the first tool fixed (as described in section 10.2.3). The subplots (a) and (b) show the absolute and compatible precision, respectively. The subplots (c) and (d) show the training and validation losses, respectively.

11. Conclusion

The aim of the work was to predict next tools for workflow paths (tool sequences). The workflows were first divided into paths and these paths were treated as sequences of tools. The paths in a workflow were considered independent. The recurrent neural network was used as a classifier to learn tools connections from these paths.

11.1. Network configuration

Many different configurations of the recurrent network were used to achieve a higher precision without overfitting. The baseline configuration of the recurrent neural network performed well. Applying dropout was found to be beneficial to reduce overfitting. A larger number of memory units and larger size of embedding layer and mini-batch improved the precision too. The optimisers with adaptive strategies performed well compared to the non-adaptive ones. A comparison was made to ascertain the best learning rate and activation.

11.2. The amount of data

A large number of workflows played a significant role to improve the precision (absolute precision $\approx 89\%$ and compatible precision $\approx 99\%$). With a large number of workflows, it was ensured that the number of paths increased for each feature (tool connection). Moreover, a more complex network with 512 memory units and 512 dimensional embedding layer could be used without overfitting it. Collectively, they ensured a higher precision. But, with an increased number of workflows and a more complex network, the running time of the analysis also increased.

11.3. Decomposition of paths

For the idea of decomposing only test paths, the recurrent neural network was trained on longer paths. It failed to learn the semantics of smaller paths (present in test paths). But, for the other ideas, where the training and test paths were decomposed identically, it performed well achieving $\approx 89\%$ absolute precision.

11.4. Classification

There were multiple tools as next tools for many paths. Therefore, multilabel classification approach was required. Compatible precision was higher than the absolute precision for all the approaches of path decomposition. It may be because only the knowledge of next tools present in training paths was used for prediction. Therefore, the predicted next tools did not match the actual next tools for some paths.

12. Future work

12.1. Train on longer and test on shorter paths

A poor performance was noted for the idea of decomposing only test paths. Detailed reasons should be found out and corrected to achieve a good precision for this approach too. Different configurations of the recurrent neural network can be used to check whether they can improve precision.

12.2. Restore original distribution

While taking unique paths into training and test sets, the original distribution of paths was not taken into consideration. The original distribution of paths should be restored for training paths only. Then, the recurrent neural network would assume that a path that repeats many times is more important. Moreover, the frequency of paths which are rare can be increased too [41].

12.3. Use convolution

The recurrent neural network achieved a precision of $\approx 89\%$. Using convolutional layers along with the recurrent layers, classification can be improved [14, 42]. The convolutional layers can be stacked above the recurrent layers to learn sub-features from the smaller parts of tool sequences. Convolution is well-suited to learn features irrespective of their positions.

12.4. Use other classifiers

The recurrent neural network was used as a classifier for this approach. *Bayesian network* and *markov random fields* can be used as classifiers to predict next tools.

They may provide different insights. Multiple configurations of a neural network with dense layers can be tried out to check whether they improve precision.

12.5. Decay prediction based on time

The tools which are not used regularly in Galaxy should be given less importance in the analysis. The prediction of the next tools should include more popular tools. For this, usage statistics of tools is required. To achieve that, the following two ideas can be used:

- Exclude those tools which are rarely used for a certain amount of time while processing the workflows.
- Assign less importance (small weights) to these tools while training the recurrent neural network.

Bibliography

- [1] E. Afgan, D. Baker, M. Van Den Beek, D. Blankenberg, D. Bouvier, M. Cech, J. Chilton, D. Clements, N. Coraor, C. Eberhard, B. Grüning, A. Guerler, J. Hillman-Jackson, G. Von Kuster, E. Rasche, N. Soranzo, N. Turaga, J. Taylor, A. Nekrutenko, and J. Goecks, “The Galaxy platform for accessible, reproducible and collaborative biomedical analyses: 2016 update,” *Nucleic Acids Research*, vol. 44, pp. W3–W10, July 2016.
- [2] S. Robertson and H. Zaragoza, “The probabilistic relevance framework: Bm25 and beyond,” *Found. Trends Inf. Retr.*, vol. 3, pp. 333–389, Apr. 2009.
- [3] C. E. Shannon, “A mathematical theory of communication,” *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 3–55, Jan. 2001.
- [4] P. W. Foltz, “Latent semantic analysis for text-based research,” *Behavior Research Methods, Instruments, & Computers*, vol. 28, pp. 197–202, Jun 1996.
- [5] 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.
- [6] 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.
- [7] J. Yang, “Notes on low-rank matrix factorization,” *CoRR*, vol. abs/1507.00333, 2015.
- [8] G. Shabat, Y. Shmueli, and A. Averbuch, “Missing entries matrix approximation and completion,” vol. abs/1302.6768, 2013.
- [9] Q. V. Le and T. Mikolov, “Distributed representations of sentences and documents,” *CoRR*, vol. abs/1405.4053, 2014.

- [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] 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.
- [17] 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.
- [18] N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, “Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription,” 2012.
- [19] A. McCarthy and C. K. Williams, “Predicting patient state-of-health using sliding window and recurrent classifiers,” 2016.
- [20] H. Jia, “Investigation into the effectiveness of long short term memory networks for stock price prediction,” *CoRR*, vol. abs/1603.07893, 2016.
- [21] F. J. Ordóñez and D. Roggen, “Deep convolutional and lstm recurrent neural networks for multimodal wearable activity recognition,” *Sensors*, vol. 16, no. 1, 2016.

- [22] S. Karan and J. Zola, “Exact structure learning of bayesian networks by optimal path extension,” *CoRR*, vol. abs/1608.02682, 2016.
- [23] P. Spirtes, C. Glymour, R. Scheines, S. Kauffman, V. Aimale, and F. Wimberly, “Constructing bayesian network models of gene expression networks from microarray data,” 02 2002.
- [24] D. M. Chickering, D. Heckerman, C. Meek, and D. Madigan, “Learning bayesian networks is np-hard,” tech. rep., 1994.
- [25] G. F. Cooper, “The computational complexity of probabilistic inference using bayesian belief networks (research note),” *Artif. Intell.*, vol. 42, pp. 393–405, Mar. 1990.
- [26] D. M. Chickering, D. Heckerman, and C. Meek, “Large-sample learning of bayesian networks is np-hard,” *J. Mach. Learn. Res.*, vol. 5, pp. 1287–1330, Dec. 2004.
- [27] A. Sarkar and D. B. Dunson, “Bayesian nonparametric modeling of higher order markov chains,” vol. abs/1506.06268, 2015.
- [28] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” *CoRR*, vol. abs/1409.1259, 2014.
- [29] R. Pascanu, T. Mikolov, and Y. Bengio, “Understanding the exploding gradient problem,” *CoRR*, vol. abs/1211.5063, 2012.
- [30] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, pp. 1735–1780, Nov. 1997.
- [31] M. Hermans and B. Schrauwen, “Training and analysing deep recurrent neural networks,” pp. 190–198, 2013.
- [32] D. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus),” *CoRR*, vol. abs/1511.07289, 2015.
- [33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

- [34] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent neural network regularization,” *CoRR*, vol. abs/1409.2329, 2014.
- [35] Y. Gal and Z. Ghahramani, “A theoretically grounded application of dropout in recurrent neural networks,” pp. 1027–1035, 2016.
- [36] S. Ruder, “An overview of gradient descent optimization algorithms,” *CoRR*, vol. abs/1609.04747, 2016.
- [37] M. Li, T. Zhang, Y. Chen, and A. J. Smola, “Efficient mini-batch training for stochastic optimization,” in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’14, (New York, NY, USA), pp. 661–670, ACM, 2014.
- [38] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *CoRR*, vol. abs/1412.6980, 2014.
- [39] A. Graves, “Generating sequences with recurrent neural networks,” *CoRR*, vol. abs/1308.0850, 2013.
- [40] S. Wang and C. Manning, “Fast dropout training,” vol. 28, pp. 118–126, 17–19 Jun 2013.
- [41] M. Buda, A. Maki, and M. A. Mazurowski, “A systematic study of the class imbalance problem in convolutional neural networks,” *CoRR*, vol. abs/1710.05381, 2017.
- [42] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, and W. Xu, “CNN-RNN: A unified framework for multi-label image classification,” *CoRR*, vol. abs/1604.04573, 2016.

A. Appendix

A.1. Visualiser

To showcase similar tools, visualisers are created. For *latent semantic analysis* approach, there are two websites and for *paragraph vectors* approach, there is one website. In addition to showing similar tools for the selected tool, visualisers also show a few plots for the error, gradient, learning rates and the selected tool's similarity scores with all other similar tools. Links to the websites are as follows:

- Full-rank document-token matrices¹.
- Document-token matrices reduced to 5% of the full-rank².
- Paragraph vectors³

A.2. Code repository

The following sections provide the locations of the codebase (*github* repositories) used for this work. All the repositories are under MIT license.

A.2.1. Find similar scientific tools

There are separate branches for two approaches, (*latent semantic analysis* and *paragraph vectors*). For *latent semantic analysis* approach, there are two branches:

- Full-rank document-token matrices⁴.

¹https://rawgit.com/anuprulez/similar_galaxy_tools/lsi/viz/similarity_viz.html

²https://rawgit.com/anuprulez/similar_galaxy_tools/lsi_005/viz/similarity_viz.html

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

⁴https://github.com/anuprulez/similar_galaxy_tools/tree/lsi

- Document-token matrices reduced to 5% of the full-rank⁵.

Both of these branches differ only in their ranks of corresponding document-token matrices. There is a separate branch for *paragraph vectors* approach⁶.

A.2.2. Predict next tools in scientific workflows

The Galaxy workflows are represented as directed acyclic graphs and they can be visualised in a website⁷. Workflow chosen from the drop-down is displayed as a cytoscape⁸ graph. Separate code repositories are maintained for the ideas discussed in section 9.5.1. They are listed as follows:

- No decomposition of paths⁹
- Decomposition of only test paths¹⁰
- Decomposition of the training and test paths¹¹

⁵https://github.com/anuprulez/similar_galaxy_tools/tree/lsi_005

⁶https://github.com/anuprulez/similar_galaxy_tools/tree/doc2vec

⁷https://rawgit.com/anuprulez/similar_galaxy_workflow/master/viz/index.html

⁸<http://js.cytoscape.org>

⁹https://github.com/anuprulez/similar_galaxy_workflow/tree/train_longer_paths

¹⁰https://github.com/anuprulez/similar_galaxy_workflow/tree/train_long_test_decomposed

¹¹https://github.com/anuprulez/similar_galaxy_workflow/tree/extreme_paths

