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
Faculty of Engineering
Department of Computer Science
Bioinformatics Group Freiburg

July 09, 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 the indispensable means. Being precise, his pragmatic suggestions concerning the Galaxy tools and workflows helped me discern them better and improve the overall quality of the work. His advice to create a 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 for the German translation of the abstract and her continued support and help. 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 the other members of the Freiburg Galaxy team for their continued support and help. I appreciate and thank Tonmoy Saikia for proofreading the thesis.

Zusammenfassung

Die Arbeit erforscht zwei Konzepte, um ein Empfehlungssystem für Galaxy‘s wissenschaftliche Datenverarbeitungs-Tools und Workflows zu entwickeln. Die Konzepte werden in dieser Arbeit separat behandelt und beinhalten das Auffinden von Ähnlichkeiten in Tools und die Vorhersage nachfolgender Tools in Workflows. Der erste Teil untersucht die Methode zum Auffinden von Ähnlichkeiten bei Tools, einschließlich der Existenz ähnlicher Tools für ein jeweiliges Tool in einem Tool-Bündel. Darüber hinaus wird die Ähnlichkeit zwischen Tools quantifiziert durch das Zuweisen eines Ähnlichkeitswertes zu jedem Tool-Paar. Die Ähnlichkeit von Tools wird durch Ansätze aus der natürlichen Sprachverarbeitung und Optimierung berechnet. Für diese Aufgabe wurden über 1.000 Tools verwendet. Die Daten der Tools wurden aus verschiedenen Eigenschaften, einschließlich Name, Beschreibung, Eingabe- und Ausgabe-Datentypen und Hilfstexten gesammelt. Der zweite Teil der Arbeit entwirft ein Vorhersage-System, welches die möglichen nächsten Tools in Workflows darlegt. Diese wissenschaftlichen Workflows sind komplex und werden mit mehr als erstellt 4.000 Tools in der Bioinformatik. Die Kenntnis möglicher nächster Tools kann es weniger erfahrenen Galaxy-Benutzern leichter machen, Workflows zu erstellen. Außerdem kann die zur Erstellung eines Workflows benötigte Zeitdauer verkürzt werden. Die einzelnen, aus Workflows (gerichtete azyklische Diagramme) entnommenen Wege (Tool-Sequenzen) werden in periodische neuronale Netzwerke eingebracht, um die Semantik von Tool-Verbindungen zu ermitteln. Aus der Erfahrung von höheren, in diesen Tool-Verbindungen vorherrschenden Abhängigkeiten, werden die Vorhersagen gemacht. Um das Vorhersage-Modell in Erfahrung zu bringen wurden über 167.000 Pfade verwendet.

Abstract

The thesis enquires into two concepts to develop a recommendation system for the Galaxy's scientific data-processing tools and workflows. The concepts incorporate finding similarity in tools and predicting next tools in workflows and are dealt with separately in the thesis. The first part explores a way to find similarity in tools. It implies which similar tools exist for each tool within a cluster of tools. In addition, it quantifies the similarity among tools by assigning a similarity score for each pair of tools. The similarity in tools is computed using the approaches from natural language processing and optimisation. Over $1,000$ tools have been used for this task. The metadata of tools is gathered from multiple attributes including name, description, input and output data types and help text. The second part formulates a prediction system to display the 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 the 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 (tools sequences) extracted from the workflows are fed to the recurrent neural networks to learn the semantics of tools connections. The predictions are made by learning higher-order dependencies prevalent in these tools connections. More than $167,000$ paths have been utilised to learn the predictive model.

Contents

List of Figures

List of Tables

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 extensively used biological data formats like *fasta*, *fastaq*, *gff*, *pdb* and many more[1]. The Galaxy offers scientific tools and workflows to transform these datasets (figure 1). Each tool has an exclusive way to process the datasets. A simple example of this processing is to merge two compatible datasets to make one. Another example is to reverse complement a sequence of nucleotides[2].

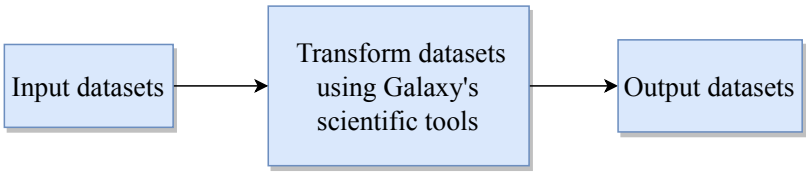

Figure 1.: Dataset transformation: The image shows a general flow of data transformation using the Galaxy's scientific tools.

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 the *text manipulation* category. Similarly, there are many tool categories like *imaging*, *convert formats*, *genome annotation* and many others[3]. These tools are the building blocks of a workflow. A workflow is data processing pipeline where a set of tools are joined one after another. The connected tools in the 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 variant (specifically single and multiple nucleotide polymorphisms and insertions and deletions) detection following a bayesian approach.

[1] `https://galaxyproject.org/learn/datatypes/`
[2] `https://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, *trimmomatic* tool trims a *fastq* file and produces *fastqsanger* file which is used as an input file to another tool like *fastqc*. A tool has multiple attributes which include its input and output file types, name, description, help text and many more[4]. These attributes constitute the metadata of a tool. The metadata shows that some tools have similar functions while some share similarities in their input and output file types. For example, a 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 the similar file type. Similar tools can also be found by analysing other attributes like the name or description.

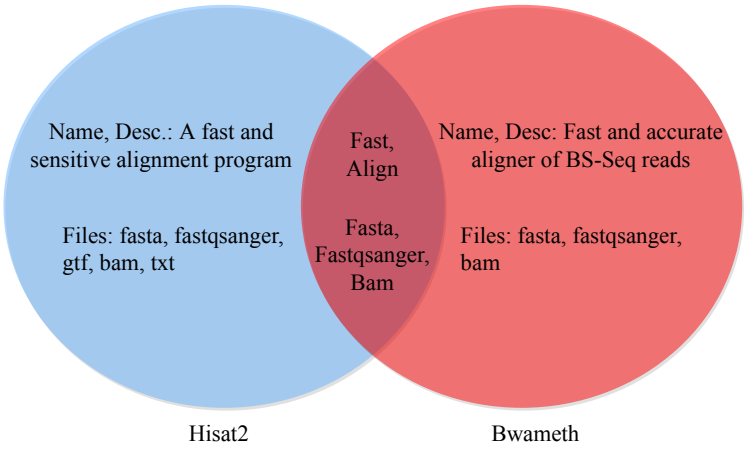

Figure 2.: Common features of two tools: The venn diagram shows the common features of two tools - *hisat2* and *bwameth*. Based on their common features, shown in the middle of the venn diagram, the similarity between them is assessed.

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. These tools share common file types (*fasta*, *fasaqsanger*, *bam*). Moreover, they share a similar function of aligning. By extrapolating this method of finding similar features among tools, a set of similar tools for each tool can be prepared.

[4] `https://docs.galaxyproject.org/en/master/dev/schema.html`
[5] `https://usegalaxy.eu/?tool_id=toolshed.g2.bx.psu.edu/repos/bgruening/hicexplorer_hicpca/hicexplorer_hicpca/2.1.0&version=2.1.0`

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 keep 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 using the Galaxy.

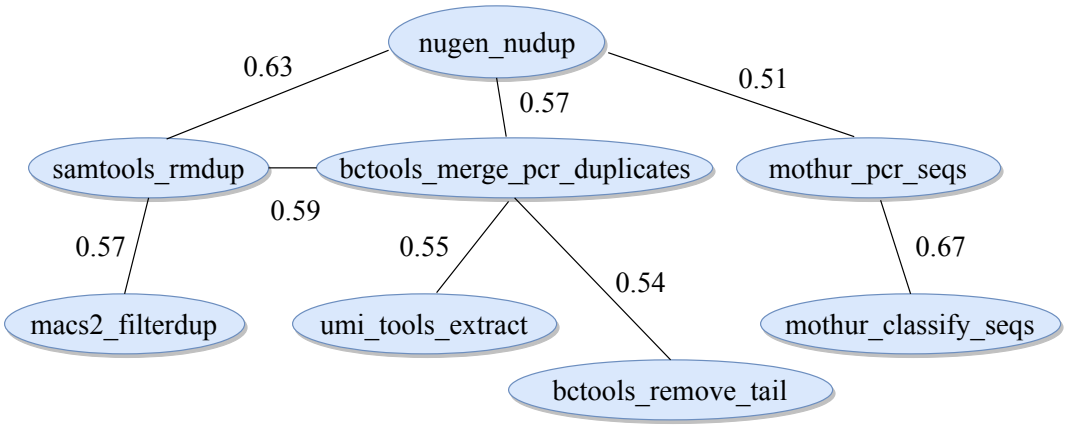

Figure 3.: Similarity graph of tools: In the graph, the nodes are represented by the tools and the edges show similarity scores for each pair of tools. Higher the similarity score, more similar a pair of tools are.

To elaborate more, a tool *nugen nudup*[6] (figure 3) is taken. It is used to find and remove PCR duplicates. A few of its similar tools are found. Similar 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 network of the related tools. This network shows *connectedness* among tools and can help a user to find multiple ways to process data. A continuous representation of similarity (a real number between 0.0 and 1.0) is learned between each pair of tools. Figure 3 shows how this similarity graph can be created after computing the similarity among tools.

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

2. Approach

This section gives a comprehensive description of the approach to compute similarity among tools. First of all, the metadata of all the tools is extracted from the github's tool repositories. The metadata 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. To create a vector for each tool, the words from all the tools are merged and the size of this collection of words gives the size of the vector. Each word gets an index in the vector. In a tool's vector, the positions of the respective words (words from the tool's set) contain the frequency of their occurrence. The indices which belong to the words not present in the set for a tool contain zero. These vectors are used to compute similarity scores for each pair of tools using the similarity measures. The similarity scores for each tool with all the other tools create a similarity matrix. For each attribute, a similarity matrix is computed. These similarity matrices are combined using the optimisation to get a weighted average similarity matrix (figure 4).

2.1. Extract tools data

The Galaxy's scientific tools are stored at *github* across multiple repositories. One such repository is *Galaxy tools maintained by IUC*[1]. A Galaxy tool is defined in an extensible markup language (XML) file which can be parsed efficiently to gather the metadata from multiple attributes. These attributes include the input and output file types, name and description and help text. The XML files for tools start with a *tool* tag.

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

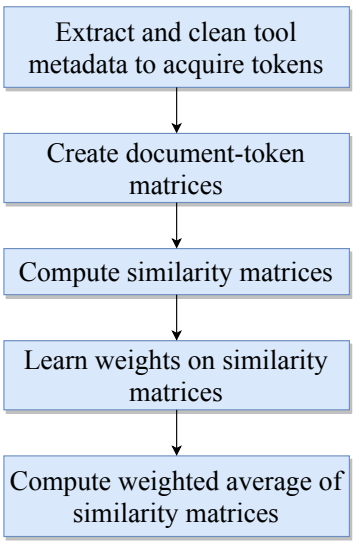

Figure 4.: Sequence of steps to find similar tools: The flowchart shows a series of steps to establish similarity among tools using the approaches from natural language processing to compute similarity matrices and optimisation to compute the weighted average of the similarity matrices.

2.1.1. Tool's attributes

A tool has multiple attributes which include input and output file types, help text, name, description, citations and many more[2]. But, not all of these attributes are 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 for 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

[2] `https://docs.galaxyproject.org/en/master/dev/schema.html`

types) and its functionality (name and description). Further, the help text attribute is also included in the metadata to compute similarity. 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 the detailed explanation of a tool's functions and the format of an input data to the tool[3]. Much of the information contained by this attribute is not important to clearly distinguish a tool. Therefore, only the first few lines of the text of this attribute, which illustrate a tool's core functionality, are considered. The rest of the information is discarded.

2.1.2. Gather useful metadata

Remove duplicates and stop-words

The metadata of the tools collected from multiple attributes is raw. It contains lots of noisy and duplicate items which do not add value. These items should be removed from the metadata to get tokens which are unique and useful. For example, a tool *bamleftalign* has *bam* and *fasta* as the 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. In this case, the *bam* and *fasta* are considered as a set of file types for the tool. This set does not maintain any order of the file types. The attributes like the name and description and help text contain sentences (complete or partially complete) in the English language. Therefore, to process these, the strategies from the natural language processing[4] are needed. A sentence contains many words and can have many different parts of speech. The parts of speech are the subject, object, preposition, interjection, verb, adjective, adverb, article and many more[5]. For this text-processing, only those tokens (words) are required which categorise a tool uniquely. For example, the tool *tophat* has a name and description which is read as *tophat for illumina find splice junctions using RNA-seq data*. The words like *for*, *using* and *data* cannot not distinguish the tool as they can be present in the description for many other tools. These words are called as *stop-words*[6] and are discarded. In addition, the numbers are also removed. All the remaining tokens are converted to the lower case.

[3] `https://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`

Use stemming

After removing the duplicates and stop-words, the metadata becomes clean and contains tokens which can identify the tools uniquely. The different forms of a word are used in the sentences due to the rules of grammar. For example, the word *regress* has multiple forms as *regresses* or *regression* or *regressed*. All these forms share the same root and point towards the same concept. Therefore, it is beneficial to converge all the different forms of a word to one basic form. This process is called stemming[7]. The NLTK[8] package is used for stemming. It reduces the size of the metadata and keeps the meaning of the tokens (words) in the metadata same across all the tools. Figure 5 shows a distribution of tokens in the metadata for all the three attributes of the tools. These tokens are used for computing the similarity in tools. The help text attribute contains more number of tokens compared to the input and output file types and name and description.

Learn relevance for words

After stemming the tokens from the metadata of tools, the sets of meaningful tokens are collected for all the three attributes - input and output file types, name and description and help text. These sets are called as *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. The importance factors are found out for all the tokens 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 BM25 (*bestmatch25*) [2] algorithm is used. The variables used in implementing this algorithm are as follows:

- Token frequency[9] (tf)
- Document frequency (df) and inverted document frequency (idf)
- Average document length ($|D|_{avg}$)
- Number of documents (N)

[7] `https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html`
[8] `http://www.nltk.org/`
[9] `https://nlp.stanford.edu/IR-book/pdf/06vect.pdf`

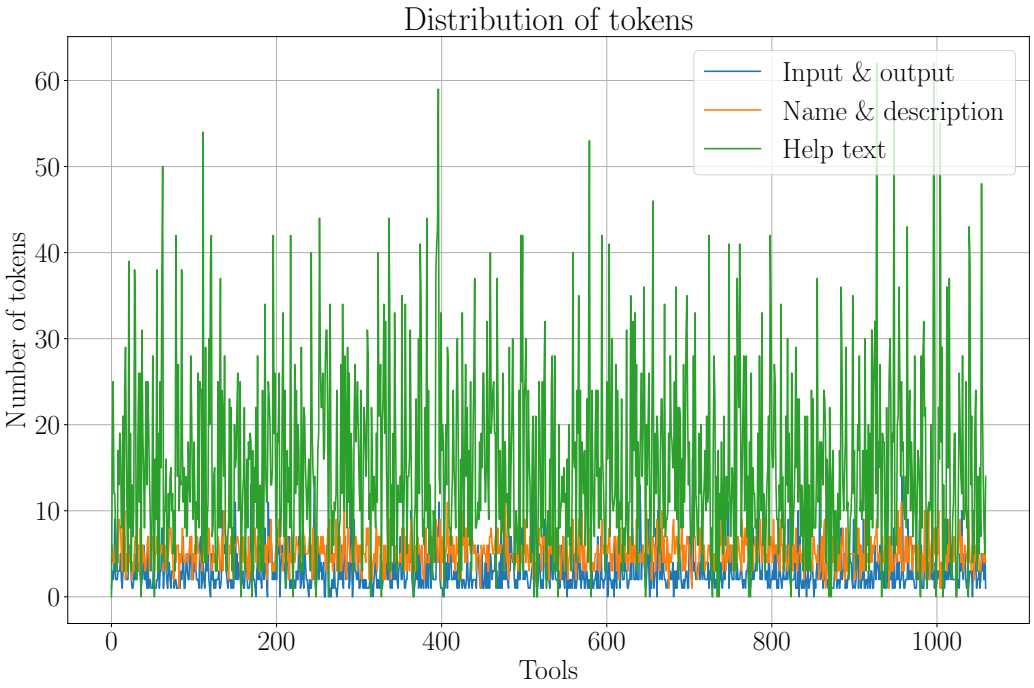

Figure 5.: Distribution of tokens (words) for all the tools: The plot shows a distribution of the number of tokens for the input and output file types, name and description and help text attributes of the tools. The number of tokens from the help text attribute is more than the number of tokens from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

- Size of a document ($|D|$)

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

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

where df is the count of the documents in which this token is present and N is the total number of documents. If a document is randomly sampled from a set of documents, the probability of this token to be present in this document is $p_i = \frac{df}{N}$. From the information theory [3], the information contained by this event is computed by $-\log p_i$. The idf is higher when a token appears only in a few documents. It means that this token is a good candidate for representing that document. The tokens which appear in many documents are not good representatives of those documents.

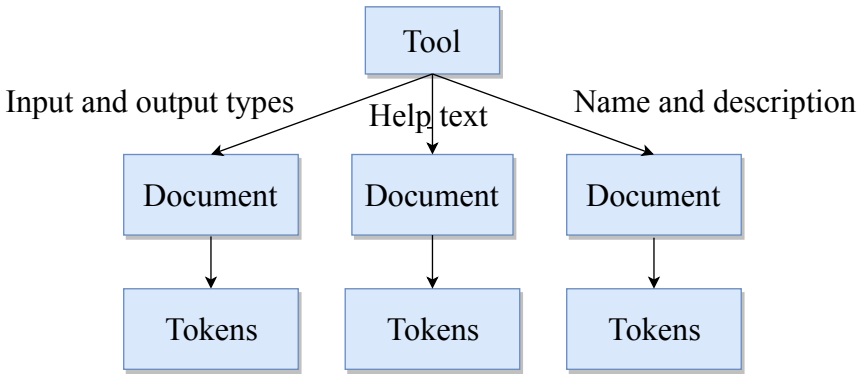

Figure 6.: Relationship between a tool, its documents and their tokens: The image shows that a tool has three documents corresponding to each attribute and each document contains tokens. For all the tools, the documents are equal to the number of tools for each attribute. The number of tokens in each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

The average document length ($|D|_{avg}$) is the average number tokens computed over all the documents. The size of a document ($|D|$) is the count of all the tokens for that document.

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

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

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

where k and b are the hyperparameters for the *bestmatch25* and their values should be tuned according to the data. k is always a positive number. When k is zero, the $BM25$ score for a token in a document is defined only by its idf ($tf^* = 1$). When k is a large number, the $BM25$ score is computed using the raw term frequency ($tf^* = tf$). b is a real number between zero and 1 ($0 \leq b \leq 1$). 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 $|D| >> |D|_{avg}$ 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 the equation 4, the relevance score for each token is computed for all the documents. Table 1 shows the $BM25$ scores for the four

documents (rows) along with five tokens (columns). Following this approach, the document-token matrices are computed for all the three attributes of the tools. For the input and output file types, the entries will have only two values, one if a token is present for a document and zero if not. For the other attributes, the $BM25$ scores are positive real numbers. This method is called the vector space model as each document (a row) is represented by a vector of tokens (table 1).

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

Table 1.: A sparse document-token matrix: This table shows a matrix of documents (tools) arranged along the rows and tokens along the columns. Each value in the matrix is a $BM25$ score assigned to a token for all the documents. This matrix is sparse and contains mostly zeros. This is 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 in a document. This representation is important to know which tokens are better representatives, but which are not for a document. They do not give any information about the co-occurrence of tokens in a document. A token is important for a document if the $BM25$ score is high but 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, *New York City*, 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 the hidden concepts within the documents and find correlation among multiple tokens, two approaches are explored:

- Latent semantic analysis[10]
- Paragraph vectors

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

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

2.2. Learn dense vector for a document

2.2.1. Latent semantic analysis

A concept is a relationship among few tokens. The *latent semantic analysis* is a mathematical way to learn these hidden concepts in the documents. It is achieved by computing a low-rank representation of the document-token matrix [4, 5, 6]. The singular value decomposition (SVD) is used to achieve it. This decomposition follows the equation:

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

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

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

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

Mathematically, the singular values[11] from the 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 present as the columns of the matrix U and the eigenvectors of $X^T \cdot X$ are the columns of the matrix V.

Figure 7 explains[12] how the SVD of a matrix is computed. The matrix U contains the information about how the tokens, arranged along the columns, are mapped to the concepts. The matrix V stores the information about how the concepts are mapped to the documents which are arranged along the rows.

[11] http://web.mit.edu/be.400/www/SVD/Singular_Value_Decomposition.htm
[12] http://theory.stanford.edu/~tim/s15/l/l9.pdf

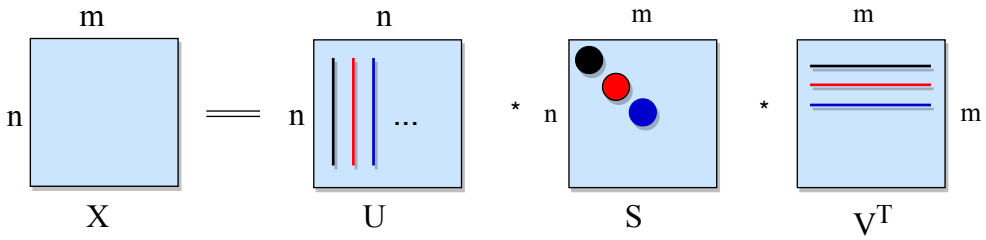

Figure 7.: Singular value decomposition: The image shows how a matrix is decomposed using the singular value decomposition. The matrix X is factorised into three matrices - U, S and V using equation 5. These factor matrices (U, S and V) are used for the low-rank estimation of the matrix X.

Low-rank approximation

The low-rank approximation of a matrix is important to discard the features which do not repeat. The document-token matrices suffer from the sparsity and show no relation among tokens. The low-rank approximations of these matrices deal with these issues. The features with smaller singular values (the last entries of the matrix S along the diagonal in equation 5) represent noise and are discarded and the features with larger singular values (the top entries of the matrix S along the diagonal in equation 5) are retained. The resulting low-rank matrices are dense and contain only larger singular values [7]. The low-rank approximation X_k ($k < m$) is computed using the equation 8. The size of the reconstructed matrix X remains the same but its rank reduces to k.

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

where U_k is the first k columns of U, V_k is the first k rows and S_k is the first k singular values in S. X_k is called as the rank-k approximation of the full-rank matrix X. Figure 9 shows how the fraction of the sum of singular values changes with the fraction of the ranks of the document-token matrices. The percentage rank is $k \div K$ where $1 <= k <= K$ and K is the original (full) rank of a matrix. From figure 9, if the ranks of matrices are reduced to 70% of the full-rank, it is still possible to capture $\approx 90\%$ of the sum of singular values. The reduction to half of the full-rank achieves $\approx 80\%$ of the sum of singular values. This variation is also shown for the input and output file types in figures 8 and 9 but its rank is not reduced. Its plot is added only for completeness.

The ranks of the document-token matrices for the name and description and

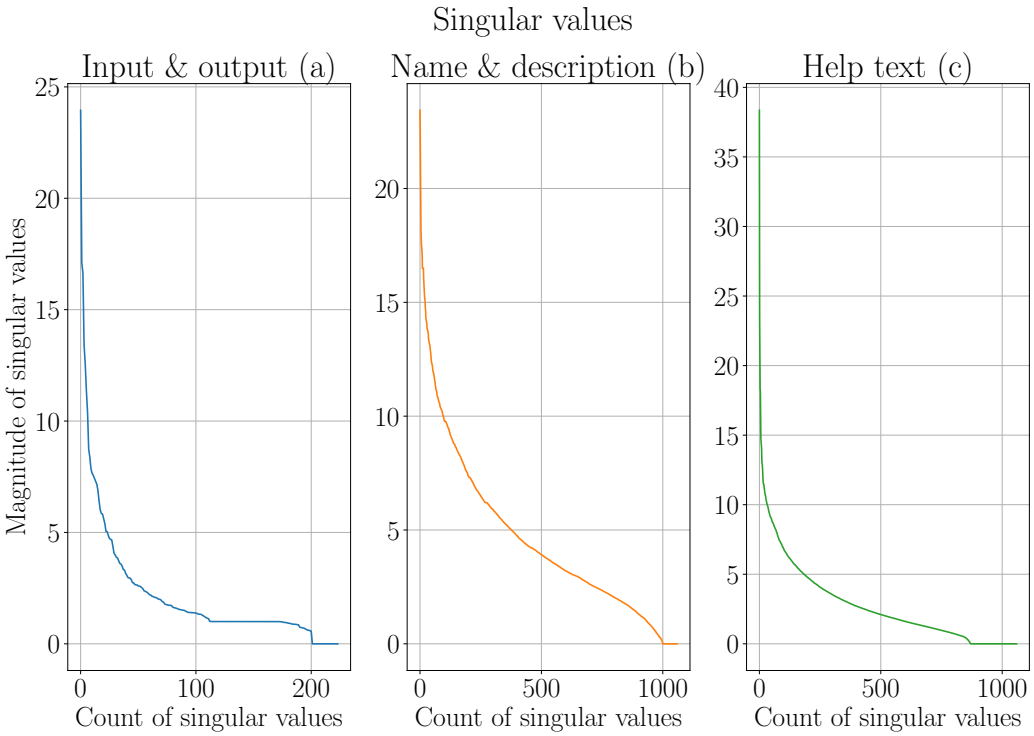

Figure 8.: Singular values of the document-token matrices: The plot shows the singular values computed using the singular value decomposition (equation 5) for the document-token matrices of the three attributes (input and output file types (a), name and description (b) and help text (c)). The diagonal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

help text attributes are reduced 5% of the corresponding full-rank. It gives the dense, low-rank approximations of the 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 the similarity measures for each pair of document vectors. There are many similarity measures which can be used like the *euclidean distance*, *cosine similarity*, *manhattan distance* or *jaccard index*. In this work, the *cosine angle similarity* is used for the name and description and help text attributes and the *jaccard index* for the input and output file types to compute the similarity between each pair of the document vectors. A positive real number between 0.0 and

1.0 is assigned as the similarity score for a pair of documents. The higher the score, 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 square and symmetric matrix (also known as the 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).

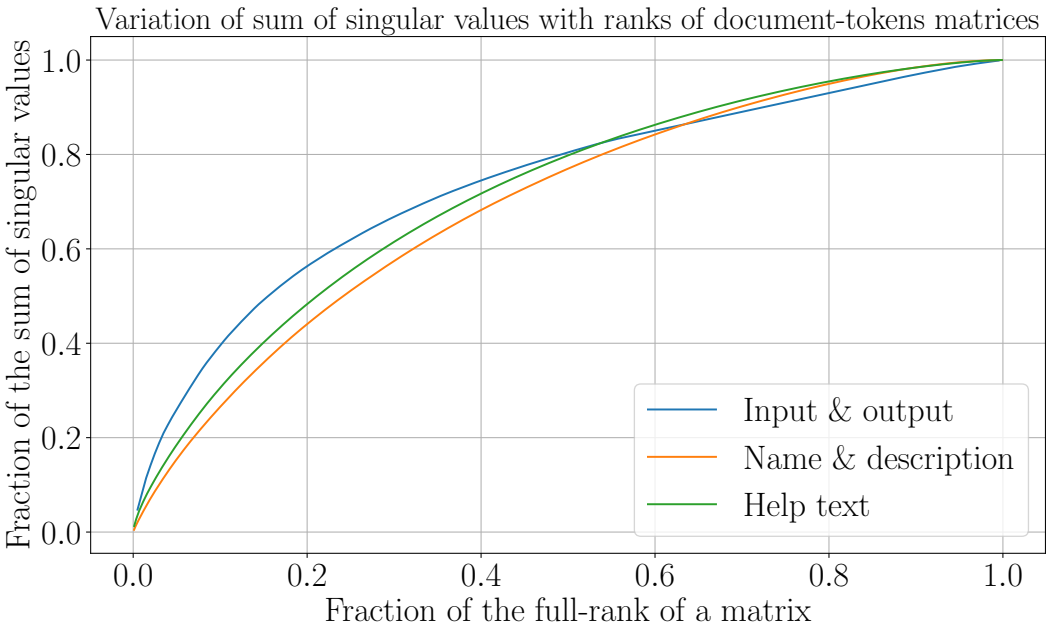

Figure 9.: The variation of the fraction of ranks of document-token matrices with the fraction of the sum of singular values: This plot merges the singular values from the three different matrices from figure 8 into one plot. As the ranks and sum of singular values of all the document-token matrices vary, they are converted to respective percentages. 0.2 on the rank axis (x-axis) means 20% of the original rank of a matrix. In the 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 the singular values for all the attributes. In the equation $sum_{fraction} = \frac{\sum_{i=1}^{k} s_i}{\sum_{i=1}^{K} s_i}$, K is the number of all singular values, s_i is the i^{th} singular value and 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 the reduction in the rank 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*, the 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 the document-token matrices in order to achieve the good document vectors. This factor can be different for different document-token matrices. There are ways to find the optimal rank by optimisation using the *frobenius norm* (as a loss function). But, it is not simple [8]. The weighted similarity scores are dominated by the similarity scores shared by the input and output file types. 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 a dense, fixed-size vector for each document using the neural networks. These vectors are unique in the way they capture the semantics present in the documents. The documents which share similar context are represented by similar vectors. When the cosine distance is computed between these vectors sharing similar context, a higher similarity score is observed. Moreover, it allows the documents to have variable lengths.

Approach

It learns vector representations for all the words and documents (sets of words). The words which are used in a similar context have similar vectors. For example, the words 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:

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

where T is the total number of words (w_i) in a document, k is the window size. The window size 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 the backpropagation[14] is used to compute the gradient.

[13] `http://cs231n.github.io/linear-classify/#softmax`
[14] `http://colah.github.io/posts/2015-08-Backprop/`

The stochastic gradient descent is used as an optimiser. The paragraph vectors are used to learn the probability of the next words in the context. The dense vectors are learned for each word as well and are known as the word vectors. The paragraph and word vectors are averaged or concatenated to make a classifier which predicts the next words in the 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 the words in this context. The words vectors are shared across all the paragraphs (documents) and paragraph vector is unique to each paragraph. Figure 10 explains this approach.

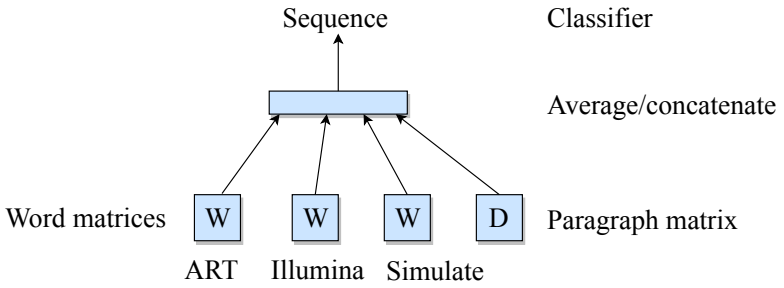

Figure 10.: Distributed memory approach for paragraph vectors: This image shows a mechanism for learning paragraph (document) vectors. W is a word matrix where each word is represented by a vector. D is a paragraph matrix where each paragraph (document) is represented by a vector. The word vectors are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

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

The original work, distributed representations of sentences and document[15], inspires the figures 10 and 11. The second form of learning paragraph vectors (distributed bag-of-words) is simple and is used to learn document (paragraph) vectors for the

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

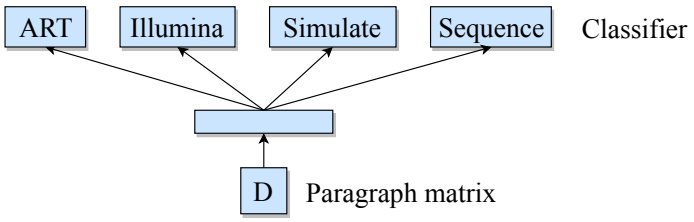

Figure 11.: Distributed bag-of-words approach for paragraph vectors: This image shows how the paragraph vectors are learned by predicting a random word chosen from a randomly selected set of words. D is a paragraph matrix where each paragraph (document) is represented by a vector. In this approach, the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

name and description and help text attributes. Only the paragraph vectors are learned (and not the word vectors) which makes it computationally less expensive [9]. The number of parameters is less than the distributed memory model which learns word vectors as well in addition to the paragraph vectors. The number of parameters for the distributed bag-of-words is $\approx N \times z$ where N is the number of paragraphs (documents) and z is the dimensionality of each vector.

2.3. Similarity measures

Using *latent semantic analysis* and *paragraph vectors* approaches, documents vectors are learned. To find similarity between a pair of document vectors, a similarity measure is applied to get a score. This score quantifies how much similar a pair of documents are. In this work, two similarity measures, *cosine similarity* and *jaccard index*, are used. Both of these similarity measures 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:

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

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

where $|x|$ is the norm of the vector x. If the norm is zero, $\cos\theta$ is set to zero. $x \cdot y$ is the dot product of vectors x and y. If the documents are dissimilar, then it is zero and if they are completely similar, it is one. For all the other cases, it stays between zero and one. This score is understood as the probability of similarity between a pair of documents[16].

2.3.2. Jaccard index

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

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

where A and B are two sets. $\cap$ is the number of entities present in both the sets and $\cup$ is the count of unique entities present in both the sets. [10]. This measure is used to compute the 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. The *jaccard index* for this pair of tools would be:

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

2.4. Optimisation

The similarity matrices are computed by applying the 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 of similarity scores from the matrices. By doing this, the combined similarity scores of one document (tool) with all the other documents (tools) are achieved. Iterating this process for all the documents would result in a similarity matrix. It contains the similarity scores of the documents with all the other documents. The diagonal entries of this matrix would be one and all the other entries would be a positive real number between zero and one. Another way to compute the combination of the similarity matrices is to learn the weights on the corresponding rows from the three matrices. These weights are used to compute the

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

weighted average of the similarity matrices (equation 14). The weights are positive real numbers between zero and one. For the corresponding rows in the similarity matrices, they sum up to one. Instead of using the fixed weights of $1/3 = 0.33$ (3 is the number of similarity matrices), an optimisation technique is employed to find these weights.

$$SM^k = w^k_{io} \cdot SM^k_{io} + w^k_{nd} \cdot SM^k_{nd} + w^k_{ht} \cdot SM^k_{ht} \tag{14}$$

where weight (w) is a positive, real number and satisfies $w^k_{io} + w^k_{nd} + w^k_{ht} = 1$. SM^k_{io}, SM^k_{nd} and SM^k_{ht} are the similarity scores (matrix rows) for the k^{th} document for the input and output file types, name and description and help text attributes respectively. The similarity scores, SM, has a dimensionality of $1 \times N$ where N is the number of documents. 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 true similarity values are set based on the similarity measures to define the error function. The maximum similarity between a pair of documents can be one as the values from the *cosine distance* and *jaccard index* can be at most one. The mean squared error is computed between the true and computed similarities.

$$SM^k_{ideal} = [1.0, 1.0, ..., 1.0]_{1 \times N} \tag{15}$$

where SM^k_{ideal} is the ideal similarity scores for the k^{th} document against all the other documents and N is the number of documents. Using the ideal and computed similarity scores, the mean squared error is computed (equation 16). After computing the error, it is important to verify which attribute is closer to the ideal score and which is not. 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 an objective function with respect to its parameters. In this work, the weights are learned by minimising the mean squared error function:

$$Error_{io}(w^k_{io}) = \frac{1}{N-1} \cdot \sum_{j=1, j \neq k}^{N} (w^k_{io} \cdot SM^j_{io} - SM_{ideal})^2 \tag{16}$$

$$Error_{nd}(w_{nd}^k) = \frac{1}{N-1} \cdot \sum_{j=1, j\neq k}^{N} (w_{nd}^k \cdot SM_{nd}^j - SM_{ideal})^2 \tag{17}$$

$$Error_{ht}(w_{ht}^k) = \frac{1}{N-1} \cdot \sum_{j=1, j\neq k}^{N} (w_{ht}^k \cdot SM_{ht}^j - SM_{ideal})^2 \tag{18}$$

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

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

$$w^k = w_{io}^k + w_{nd}^k + w_{ht}^k = 1 \tag{21}$$

In the equations 16-19, the subscripts *io*, *nd* and *ht* refer to the input and output file types, name and description and help text respectively. Each weight term in equation 21 is a real number between zero and one. To minimise the equation 20 under the constraint given by equation 21, the gradient of the error function is calculated (equation 22). The gradient specifies the rate of change of the error with respect to the weights.

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

The *Gradient* is a vector $(Gradient_{io}, Gradient_{nd}, Gradient_{ht})$. Using this gradient vector, the weights are updated in each iteration. The update equation (23) needs the learning rate η.

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

where η is the learning rate.

2.4.2. Learning rate decay

Finding a good learning rate is an important part of the gradient descent optimisation. If the learning rate 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 these situations

are undesirable and should be avoided by starting out with a small learning rate and gradually decrease it over the iterations. The decay is important because as the learning saturates, only the small steps towards the minimum are needed to converge. This technique is called time-based decay [11]. Figure 12 shows the decay of learning rate. Equation 24 is used to decay the learning rates shown in figure 12.

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

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

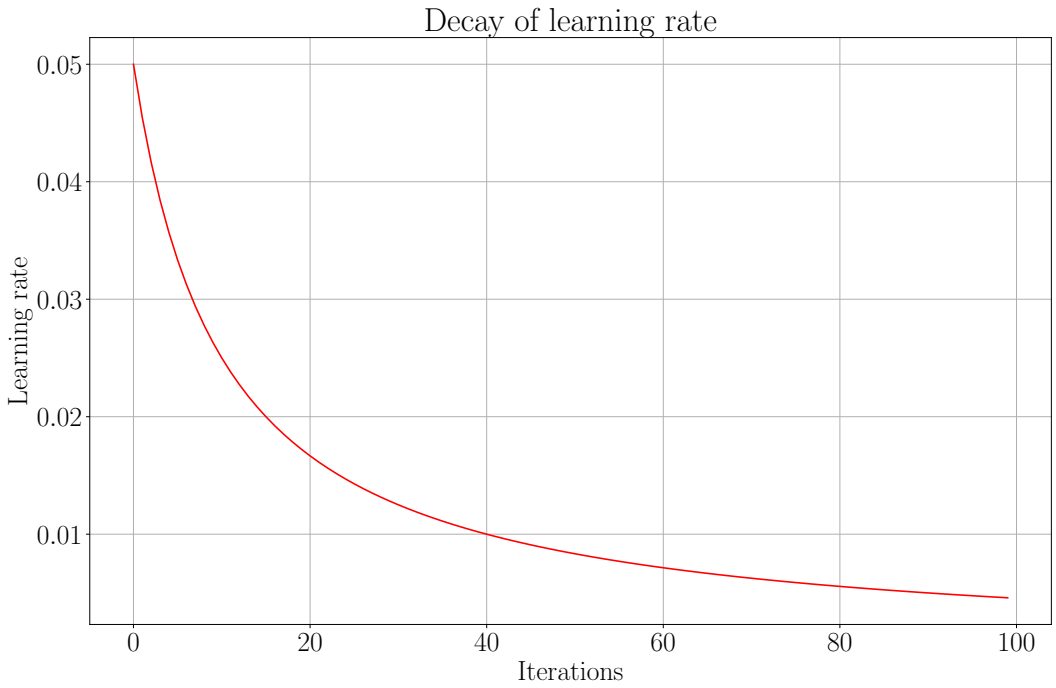

Figure 12.: Decay of learning rate for gradient descent optimiser: The plot shows how the learning rate for gradient descent changes with the iterations. It starts with a small value and decreases gradually over time. It is essential to have the learning rates which neither drops too quickly nor drops too slowly. Both 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 the saddle points. At these saddle points, the derivative of a function is zero. Adding a momentum term to the weights avoids these saddle points and the optimiser converges to the lowest point quickly. It gives the 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:

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

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

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

Nesterov's accelerated gradient

The inclusion of the momentum is useful to get necessary advance towards finding the minimum of the error function. However, the speed of going down the slope of the error function should become less if there is a possibility of change in the 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:

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

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

Gradient verification

The gradient is computed using equation 22 and used to update the weights. To verify that the computed gradient is correct, the approximated gradient computed

using the error function should be close to the actual gradient (equation 29):

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

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

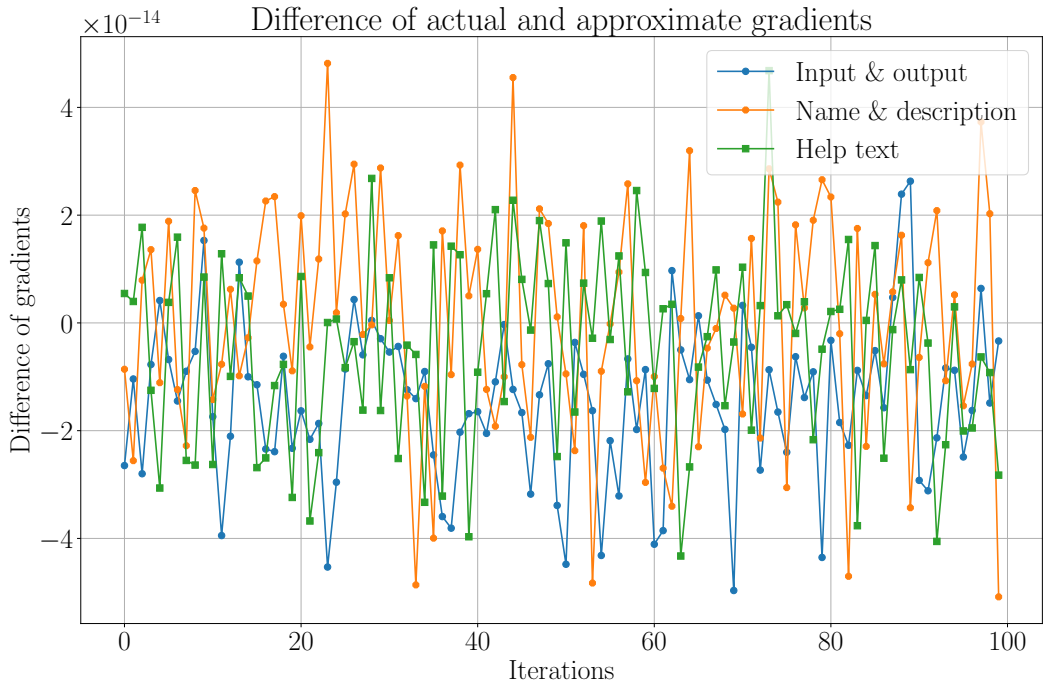

Figure 13.: Verification of the gradients for the error function: The plot shows the difference between the actual and approximated gradients. They are computed for the three attributes and averaged for all the tools over 100 iterations of the optimisation. The difference of the gradients is consistent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

3. Experiments

3.1. Number of attributes

Three different attributes, input and output file types, name and description and help text, are considered for compiling the collection of tokens. These attributes are different from each other. Using them together to make one set of tokens for each tool is not beneficial. Instead, the similarities are computed using the tokens from these three different attributes separately. These similarities are combined using optimisation.

3.2. Amount of help text

Help text attribute is noisy and contains text which is not useful for finding similarity in tools. Therefore, only the first four lines of text are considered from this attribute.

3.3. Similarity measures

For calculating the similarity scores based on the input and output file types attribute, the *jaccard index* is used. The *cosine similarity* is used for the name and description and help text attributes. Both these similarity measures give a real number between zero and one as a similarity score for a pair of tools.

3.4. Latent semantic analysis

Using *latent semantic analysis*, dense vectors are learned for each tool by reducing the ranks of the document-token matrices. The rank reduction factor is important to compute the correct similarity among tools. The ranks are reduced to 5% of the full-ranks of the document-token matrices. The resulting document-token and similarity matrices are analysed. The document-token and similarity matrices are expected

to be denser compared to no rank reduction. Only the document-token matrices of the name and description and help text attributes are considered for the low-rank estimation. The input and output document-token matrix is left in its full-rank state. The singular value decomposition method is used from its implementation in *numpy*[1] linear algebra package.

3.5. Paragraph vectors

A fixed-length dense vector (also called as paragraph vector) is computed for the name and description and help text attributes of each tool. When the number of tokens for a tool is lower, a lower number of dimensions is used for the dense vectors. Figure 5 shows that the number of tokens is higher for the help text attribute compared to the name and description attribute. Therefore, the dimension is set to 100 for the name and description and 200 for the help text. The window size for sampling the text is five for the name and description and 15 for the help text. The network runs over 10 epochs and each epoch has 800 iterations. In each epoch, all the tools are used for the training. A python implementation of this approach, *gensim*[2], is used to learn these vectors.

3.5.1. Distributed bag-of-words

Out of the two approaches (*distributed memory* and *distributed bag-of-words*) to learn the paragraph vectors, the *distributed bag-of-words* approach is applied to learn vectors. This approach does not compute the word vectors which makes it faster and computationally less expensive.

3.6. Gradient descent

It is used to optimise the combination of similarity scores computed from the three attributes by learning the importance weights on the similarity scores. Based on the similarity measures, an error function is defined for the optimiser. The optimiser runs for 100 iterations to stabilise the drop in the error. The *nesterov's accelerated gradient* is used to ensure the steady drop in the error during optimisation.

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

[2] `https://radimrehurek.com/gensim/models/doc2vec.html`

3.6.1. Learning rates

A time-based decay strategy is implemented to decay the learning rate after each iteration of the gradient descent. It starts off with a value of 0.05 and gradually decreases over the iterations. The drop in the learning rate is kept steady. The initial learning rate is set to 0.05.

4. Results and analysis

4.1. Latent semantic analysis

It is found that as the ranks of the document-token matrices of the name and description and help text attributes are reduced, they become denser. This caused the similarity matrices for the name and description and help text to become denser as well. Due to this, larger weights are learned on them by the optimiser and the distribution of weights changes. The rank reduction is not applied to the document-token matrix of the input and output file types. It was not beneficial to find out the correlation among the file types.

4.1.1. Full-rank document-token matrices

Figure 14 shows the similarity matrices computed using the full-rank document-token matrices for the input and output (14a), name and description (14b) and help text (14c) attributes. Using these similarity matrices, the weight for each is learned using the gradient descent optimiser and then combined to get a weighted average similarity matrix (14d). Figure 15 shows the distribution of these weights for three attributes. The magnitude of weights estimated for the input and output file types is larger than that of the other two attributes. The larger weights are associated with the larger values captured for the similarity matrix of the input and output file types (14a) than the other two attributes in subplots 14b and 14c.

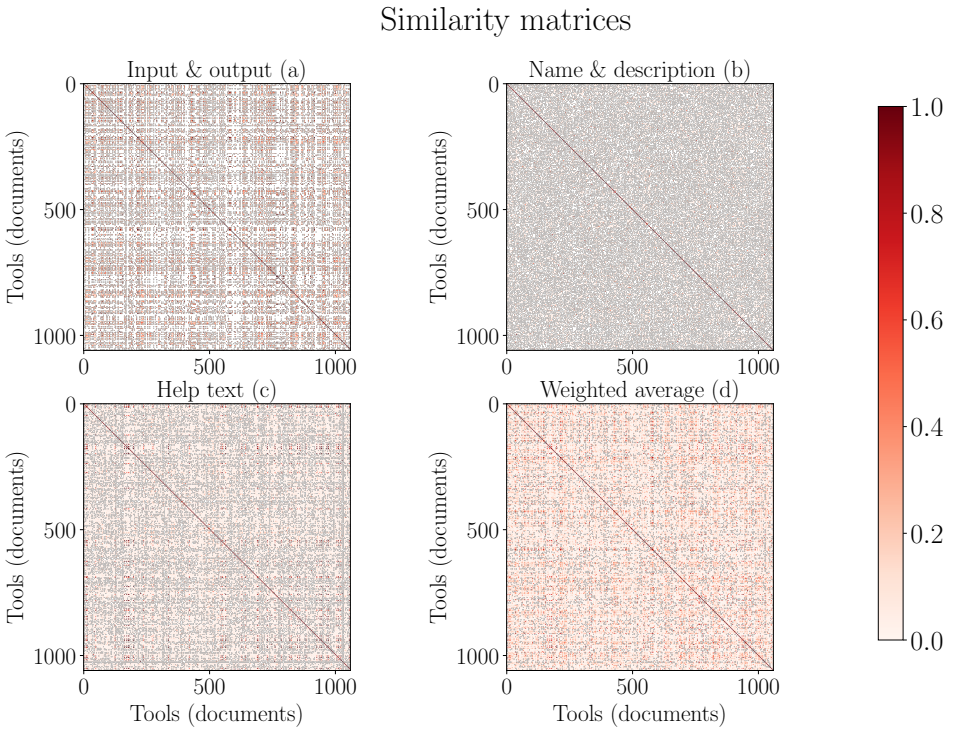

Figure 14.: Similarity matrices computed using full-rank document-token matrices: The heatmaps show the similarity matrices for the input and output file types (a), name and description (b) and help text (c) attributes. The subplot (d) shows the similarity matrix which is the weighted average of the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

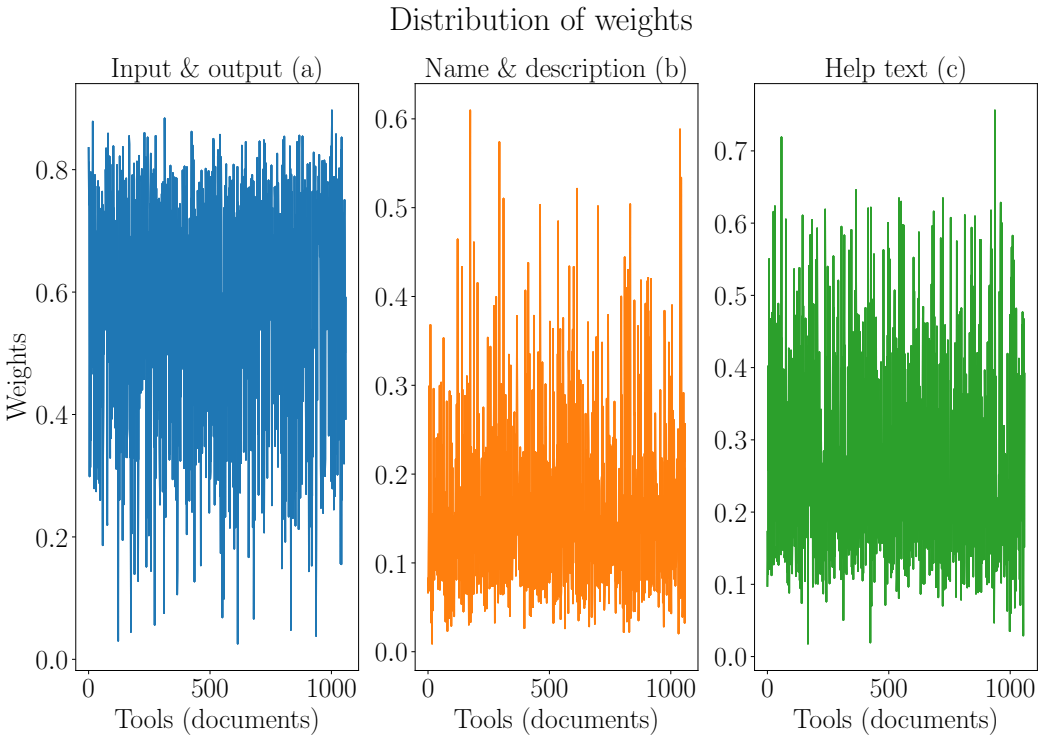

Figure 15.: Distribution of weights learned for similarity matrices computed using full-rank document-token matrices: The plot shows the distribution of weights learned for the similarity matrices (14a, 14b and 14c) by the gradient descent optimiser for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding document-token matrices contain their full-ranks. The larger weights are learned for the input and output file types attribute compared to the name and description and help text attribute.

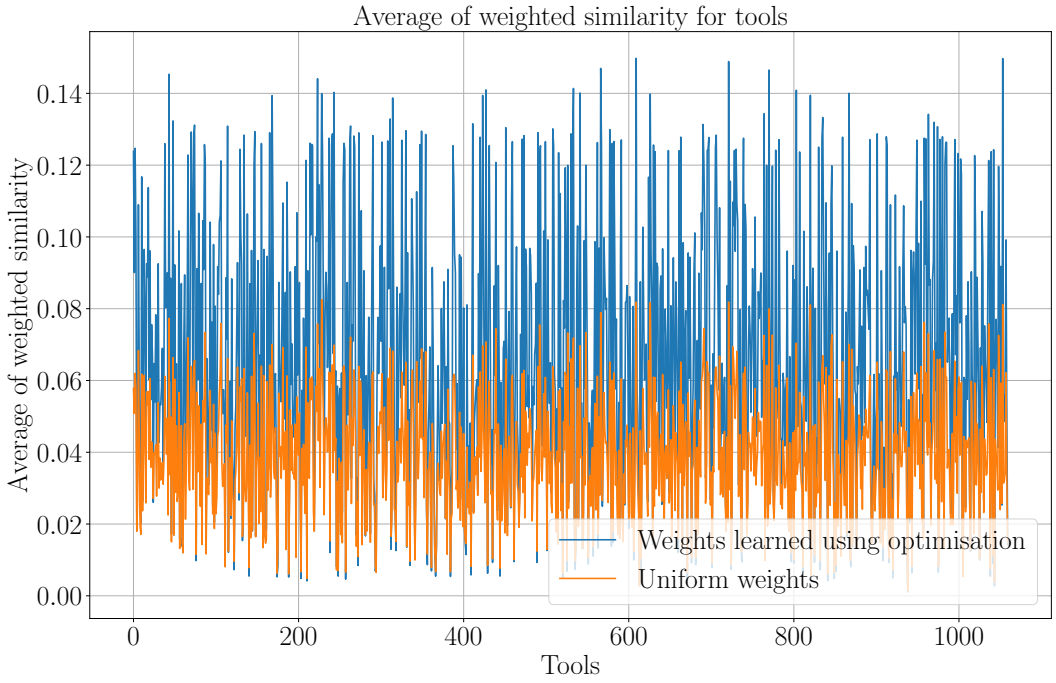

Figure 16.: Average similarity computed using uniform and optimal weights: The plot shows the average weighted similarity computed using the uniform and optimal weights (learned by the optimiser). The full-rank document-token matrices are used. The average similarity computed using the optimal weights is larger than using the uniform weights.

4.1.2. 5% of full-rank

Ranks of the document-token matrices for the 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 corresponding to the name and description and help text attributes become denser compared to figure 14. Due to this, the distribution of weights also changes (figure 18). Larger weights are learned for the name and description (18b) and help text (18c) than the weights for the input and output file types (18a).

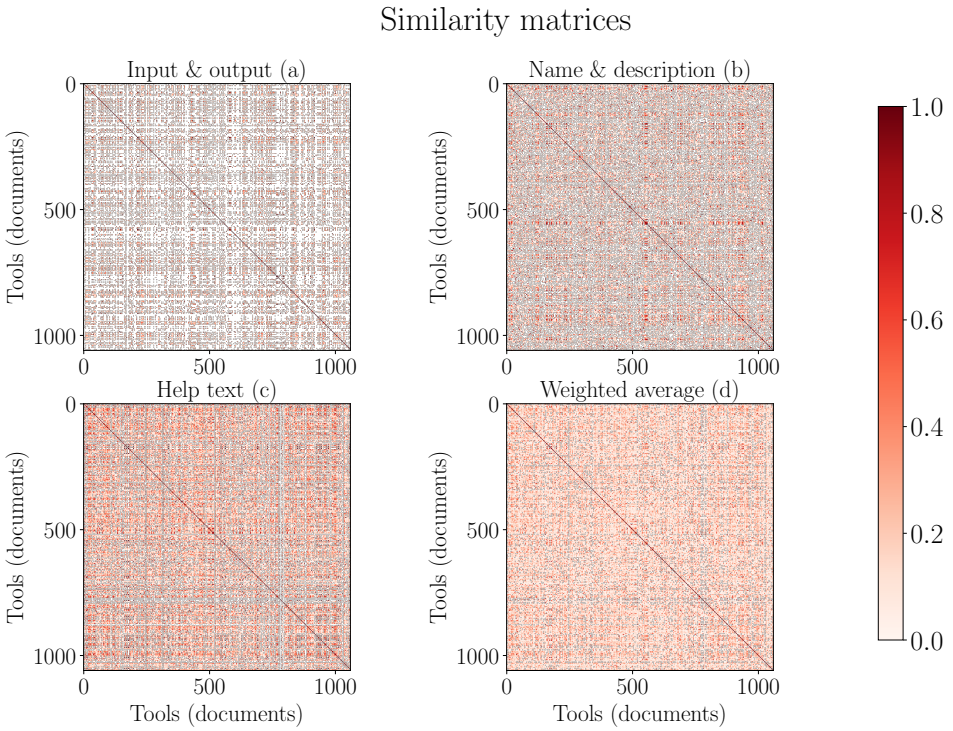

Figure 17.: Similarity matrices computed using document-tokens matrices reduced to 5% of their full-rank: The heatmaps show the similarity matrices for the input and output file types (a), name and description (b) and help text (c) attributes. The subplot (d) shows the similarity matrix which is the weighted average of the similarity matrices computed in (a), (b) and (c). The document-token matrices are reduced to 5% of their corresponding full-ranks. The similarity matrices in (b) and (c) are denser compared to that of in 14b and 14c respectively. The weights used in computing (d) are shown in figure 18.

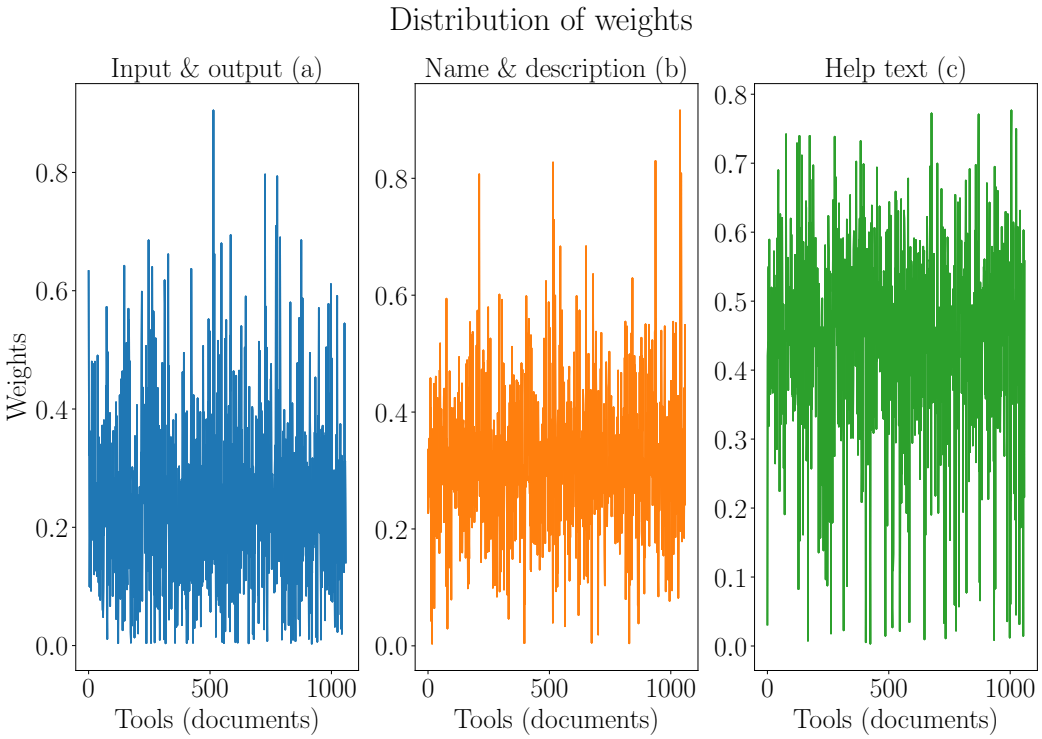

Figure 18.: Distribution of weights learned for similarity matrices computed using document-token matrices reduced to 5% of their full-rank: The plot shows the distribution of weights learned for the similarity matrices (17a, 17b and 17c) by the gradient descent optimiser for the input and output file types (a), name and description (b) and help text (c) attributes. The corresponding document-token matrices contain 5% of their full-ranks. The larger weights are learned for the name and description and help text attributes compared to figure 15b and 15c. The weights decrease for the input and output file types attribute compared to 15a.

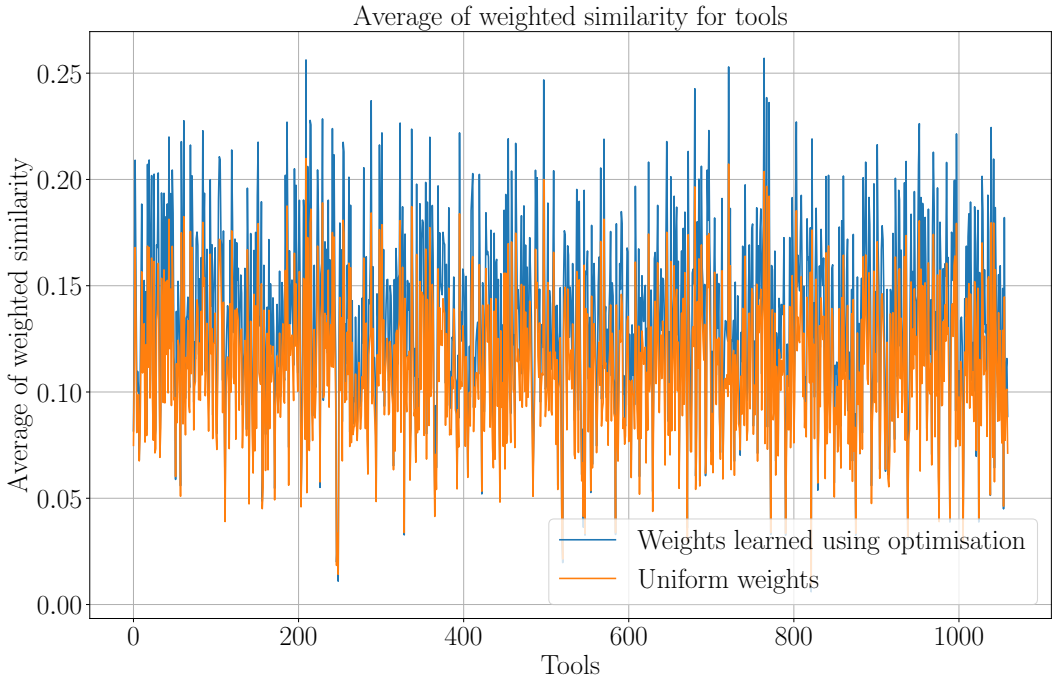

Figure 19.: Average similarity computed using uniform and optimal weights: The plot shows the average weighted similarity computed using the uniform and optimal weights. The document-token matrices are reduced to 5% of the full-rank. The average similarity scores computed using the optimal weights are larger than using the uniform weights. Compared to figure 16, it measures larger similarity scores with the uniform and optimal weights.

4.2. Paragraph vectors

The paragraph vectors approach is used to learn a fixed-length, multi-dimensional vector for each document from the name and description and help text attributes. Using the document-token matrices, the similarity matrices are computed for the attributes (figure 20). These similarity matrices are symmetric and dense. The weight distribution changes (figure 21) compared to figures 15 and 18. Larger weights are learned for the name and description (21b) and help text (21c) compared to the input and output file types (21a). Figure 22 shows that the average similarity scores across all the tools are also larger compared to that of the *latent semantic analysis* approaches (figures 16 and 19). The average similarity increases to ≈ 0.30 using the optimal weights (learned by the optimiser). For the uniform weights as well, the

average similarity is ≈ 0.25 (figure 22).

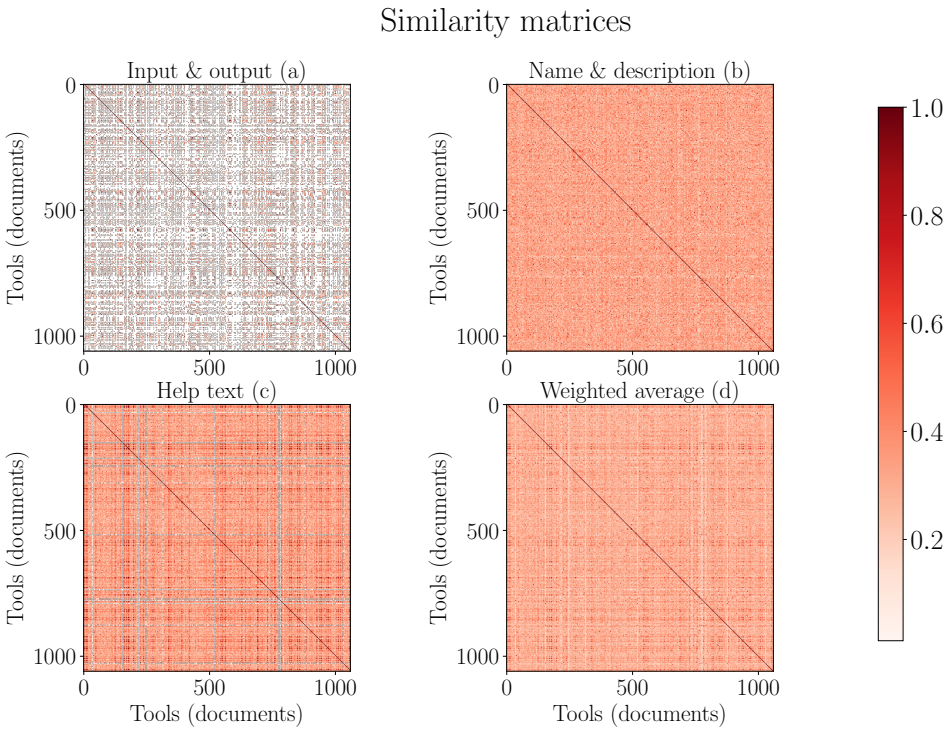

Figure 20.: Similarity matrices using paragraph vectors approach: The heatmaps show the similarity matrices for input and output (a), name and description (b) and help text (c) attributes. The subplot (d) shows a similarity matrix which is the weighted average computed using (a), (b) and (c). The weights used in computing the weighted average similarity matrix in (d) are shown in figure 21. The similarity matrices (b), (c) and (d) are denser compared to those from the figures 14 and 17.

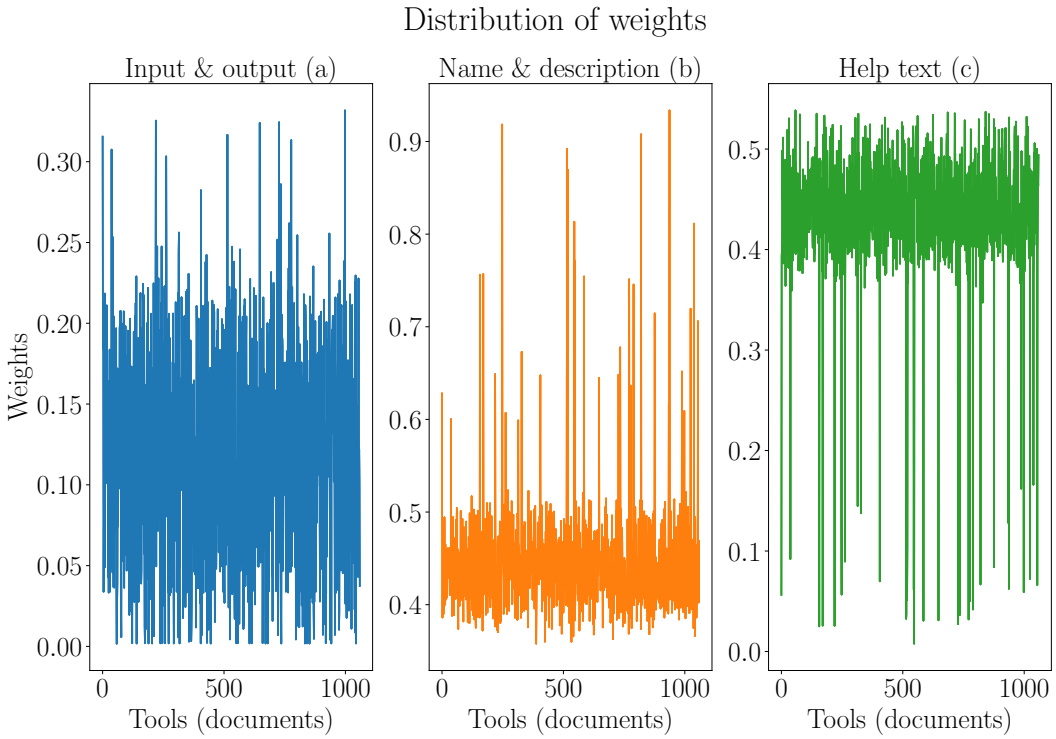

Figure 21.: Distribution of weights for similarity matrices computed using paragraph vectors approach: The plot shows the distribution of weights learned by the gradient descent optimiser on the similarity matrices (20a, 20b and 20c) for the input and output file types (a), name and description (b) and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

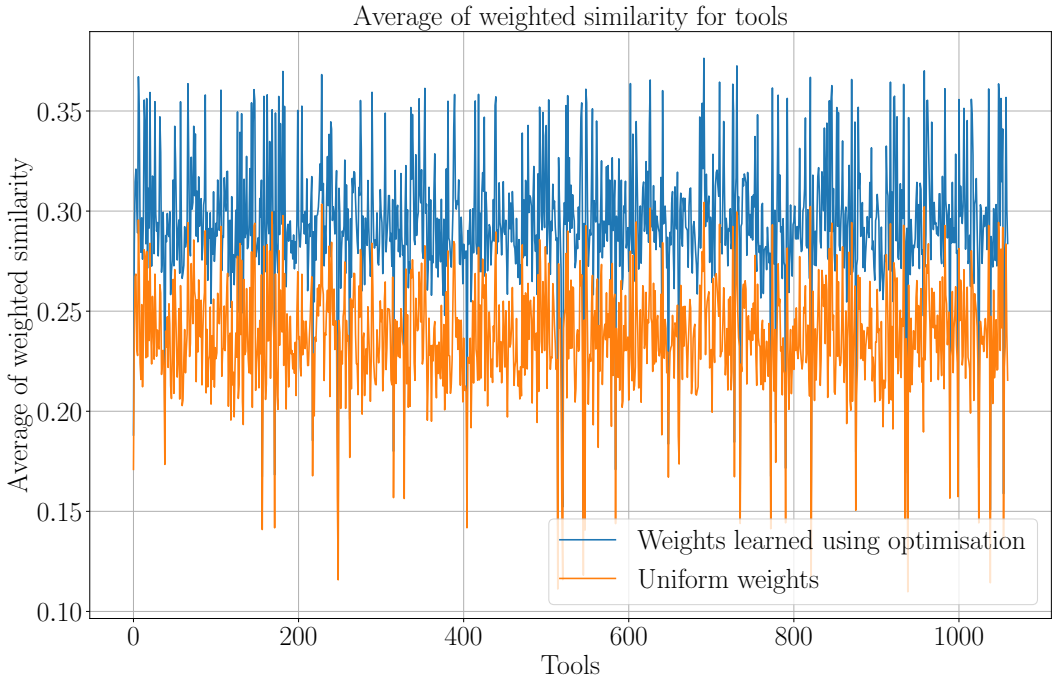

Figure 22.: Average similarity computed using uniform and optimal weights: The plot shows a comparison of the average weighted similarity scores computed using the optimal and uniform weights across all the tools. The average similarity scores using the optimal weights are larger than using the uniform weights. It learns larger similarity scores using uniform and optimal weights than the *latent semantic analysis* approach (figure 16 and 19).

4.3. Comparison of latent semantic analysis and paragraph vectors approaches

A comparison is done between the similar tools assessed by the two approaches used in this work. The similar tools for *hisat* are computed. Two similar tools for *hisat* are computed, once with the full-rank document-token matrices (table 2) and again with 5% of the full-rank document-token matrices (table 3). Moreover, to compare with the *paragraph vectors* approach, two similar tools for *hisat* are computed (table 4). Using the tables 2, 3 and 4, the similar tools for *hisat* are analysed with their respective similarity scores for the different attributes. The text below each table gives the values of the optimal weights (learned by the optimiser). For example, the

weighted similarity score in the first row of table 2 is calculated using the following equation (equation 30). The weights are given at the end of the description.

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

From the tables 2, 3 and 4, it is concluded that the *paragraph vectors* approach works better than the *latent semantic approach*. In table 2, for the name and description column, the similarity values are too less (0.07 and 0.12) for *hisat2* and *srma_wrapper*. In table 3, the similarity scores for the name and description are not correct as they measure 1.0 even though their descriptions are not exactly same. These incorrect interpretations can lead to wrong similarity assessment. As the low-rank estimation of the input and output file types document-token matrix is not done, its score remains the same in table 2 and 3. The *hisat2* has the same score for the 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 in table 4 is more relevant than that from the tables 2 and 3.

Similar tools	Input & output	Name & desc.	Help text	Weighted similarity
hisat2	0.38	0.07	1	0.42
srma_wrapper	0.5	0.12	0.01	0.4

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

Similar tools	Input & output	Name & desc.	Help text	Weighted similarity
hisat2	0.38	1	1	0.83
srma_wrapper	0.5	1	0.64	0.69

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

Similar tools	Input & output	Name & desc.	Help text	Weighted similarity
hisat2	0.38	0.46	1	0.67
tophat	0.25	0.73	0.54	0.58

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

5. Conclusion

5.1. Tools data

The 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 the other attributes. Only the first four lines of text were taken for the analysis from the help text. The metadata from the input and output file types and name and description was helpful. The extraction of metadata from the *github's* multiple repositories was slow[1]. Therefore, the metadata from the XML files was read into a tabular file and it was used as the data source for this analysis.

5.2. Approaches

Two approaches were investigated to find similar tools. The *latent semantic analysis* approach which relied on matrix rank reduction of the documents-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 risked. During optimisation, more importance was given to the input and output file types which were many times undesirable. But, this approach was simple and took less time (≈ 350 seconds for $\approx 1,050$ tools) to complete. The low-rank estimations of the sparse document-token matrices were easily computed using singular value decomposition.

The *paragraph vectors* approach worked in a more robust way than the *latent semantic analysis* approach to find similar tools. The vectors it learned for the name and description and help text were dense. The documents which were similar in context learned similar vectors. However, this approach was slow as it took $\approx 1,000$ seconds to finish. Most of the time was spent to learn the document vectors. This

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

approach achieved higher average similarity scores than the *latent semantic analysis* approach (figure 19 and 22).

5.3. Optimisation

Learning weights on similarity scores worked in a stable way and reached the saturation point already before the 50^{th} 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 to the true distribution (based on the similarity measures) as possible. The risk of getting stuck at saddle points or local minima was reduced by using momentum with an accelerated gradient to update the weights.

6. Future work

6.1. Get true similarity values

To quantify the improvements in the ranking of the similar tools for a tool, it is crucial to set the true similar values among tools. For example, a dictionary of 10 similar tools can be created for each tool. 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. This should be done to verify whether the approaches actually work in finding the similar tools. At the same time, it is not an easy task to create these logical categories and set similarity scores for thousands of tools.

6.2. Correlation

The similarity matrices were dense. The low scores from the similarity matrices can be excluded to retain only the high scores. It can be done by finding the median of the scores for a tool and those scores are set to zero which are lower than the median. Then, the optimisation would consider the important scores.

6.3. Other error functions

Mean squared error is used as a loss function for the optimiser. Other error measures like the *cross-entropy* can be used. With a new error function, it is necessary to redefine the true similarity value (it was set to one as the best similarity score between a pair of tools. With the *cross-entropy* error, one term would always be zero). The different similarity measures like the *euclidean distance* can be used as well.

6.4. More tools

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

6.5. 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 the similar tools are used for similar kind of data processing. This concept can be used to compute similar tools. The distributed memory approach learns word vectors for words in a document. The document can be replaced by the paths in the workflows and the tools would become the words. It can then learn the tool vectors. The tools which are used in the similar context would learn similar vectors.

Part II.

Predict next tools in scientific workflows

7. Introduction

7.1. 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. 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.

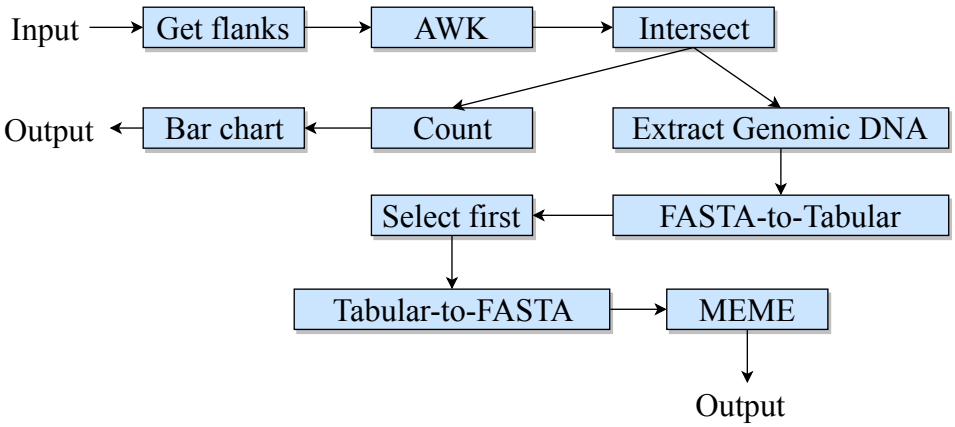

Figure 23.: A workflow: The image shows a workflow with two paths. It takes input data, processes it through multiple steps involving many tools and gives two outputs.

To create a workflow, a tool is chosen from the cluster 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 several factors like the kind of input data or data-processing required to attain the desired output.

7.1.1. 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 that should come next. These next tools must be compatible with the previous tool. No two incompatible tools can connect to each other on the canvas of Galaxy workflow editor. This knowledge of compatibility 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.

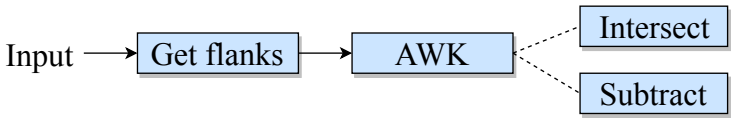

Figure 24.: Multiple next tools: The image shows a part of a workflow where one tool can connect to multiple tools.

In figure 24, a tool *AWK* can connect 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 tool or a tool sequence) 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 a 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.

8. Related work

The tools are connected one after another in workflow paths (sequences of tools). In order to predict next tools, all the previous connected tools should be taken into account. To learn how to make predictions, it is important to explore few works from machine learning or related fields which analyse similar data. Interestingly, learning from sequential data is a popular task in many other fields like natural language processing [14, 15], clinical data analysis [16] and speech processing [17, 18].

In natural language processing, deep learning models are used to learn sequences of words. They aim to categorise a corpus based on sentiments and learn part-of-speech tagging and [illegible]se vector for each word. The categorisation of sentiments involves learning contexts present in the sentences (sequences of words). The part-of-speech tagging task divides a sentence into multiple parts-of-speech which include adjective, adverb, [illegible]junctions. For sentiment analysis, [14] achieves an accuracy of $\approx 85\%$ using a recurrent neural network (gated recurrent unit). For part-of-speech tagging, the accuracy goes up to $\approx 93\%$.

For clinical data too, learning on long sequences of data proves to be beneficial [16]. In this work, the health states of patients recorded at the different points of time are analysed by accessing their electronic health records (EHR). Using a learned model, the future health states of a patient are predicted using sequences of his/her health states in the past. 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 regularisation techniques like dropout and weight normalisation.

The research presented in [17, 18] use a recurrent neural network to model music and speech signals. The performance of traditional recurrent units, long-short term memory (LSTM) and gated recurrent units (GRU) are compared. It is concluded from the comparison that the traditional recurrent units do not learn semantics present in the sequences, but the LSTM and GRU do. In this study, the sequences of 20 continuous samples (20 time-steps of speech) are learned and the next 10

continuous samples (next 10 time-steps of speech) are predicted. It is also found that the GRU performs better than the LSTM in terms of accuracy and running time. For musedata[1], a collection of piano classical music, the performances (average of the negative log-likelihood) on a test set are noted - GRU: 5.99, LSTM: 6.23 and traditional recurrent units: 6.23. Six different types of musical and speech data are tested. For four of these 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 networks. Learning tool connections in workflows to predict next possible tools is inspired by these approaches.

[1] `www.musedata.org`

9. Approach

This work proposes to predict a set of next tools at each step of designing a workflow. A learning algorithm is needed which can learn tool connections in a path and predict its next tools. This learning algorithm is a classifier. There are two coveted features of the 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 space complexity and its accuracy to predict next tools are noted for doing a comparison.

9.1. Steps

Preprocessing is necessary to make workflow paths available for analysis by a classifier. There are multiple steps involved to preprocess them. Figure 25 highlights the sequence of steps. First, all the 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 smaller tool sequences. Section 9.5.1 describes these approaches in detail. The tool sequences are divided into training and test paths. The classifier consumes the training paths to learn their characteristics along with their next tools. The robustness of learning is verified on the test paths.

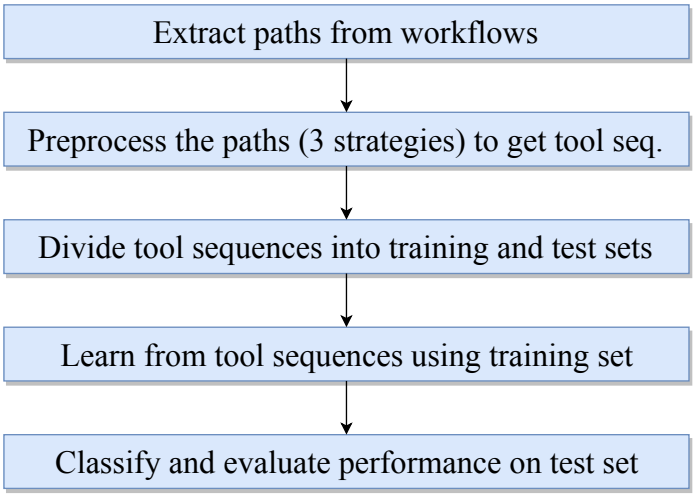

Figure 25.: Sequence of steps to predict next tools in workflows: The flowchart shows a sequence of steps to predict next tools in workflows. They are decomposed into paths. Further, these paths of varying lengths are decomposed to have tool sequences by following three different strategies explained in section 9.5.1. A classifier learns connections from tool sequences in the training set. The learning is evaluated on the test set.

9.2. 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 tool or a tool sequence can connect to more than one tool. A classifier needs data in the form of a tool or a tool sequence and its next tools (a tool or a set of different tools).

A tool or tool sequence	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 tool sequences and their respective next tools extracted from a workflow shown in figure 23. The next tools of a tool sequence are the actual next tools present in the workflow.

A workflow is decomposed into paths. Using the technique explained able 5,

these paths are further decomposed into smaller tool sequences and their respective next tools. This idea is considered because of the necessity of getting predictions in the same way. It means that when a 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 a tool *intersect*. To this sequence, when a tool *intersect* is added, the classifier should predict two tools *count* and *extract genomic DNA* (figure 23). Following this approach of decomposition of paths, tool sequences and their respective next tools are created. The next tools that are assigned to a tool or a tool sequence are its actual next tools. It means that the tool or tool sequence is connected to these next tools in workflows.

9.3. 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 tool sequence, *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 tool sequences ending in *intersect*. When a classifier predicts next tools for a tool sequence 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.

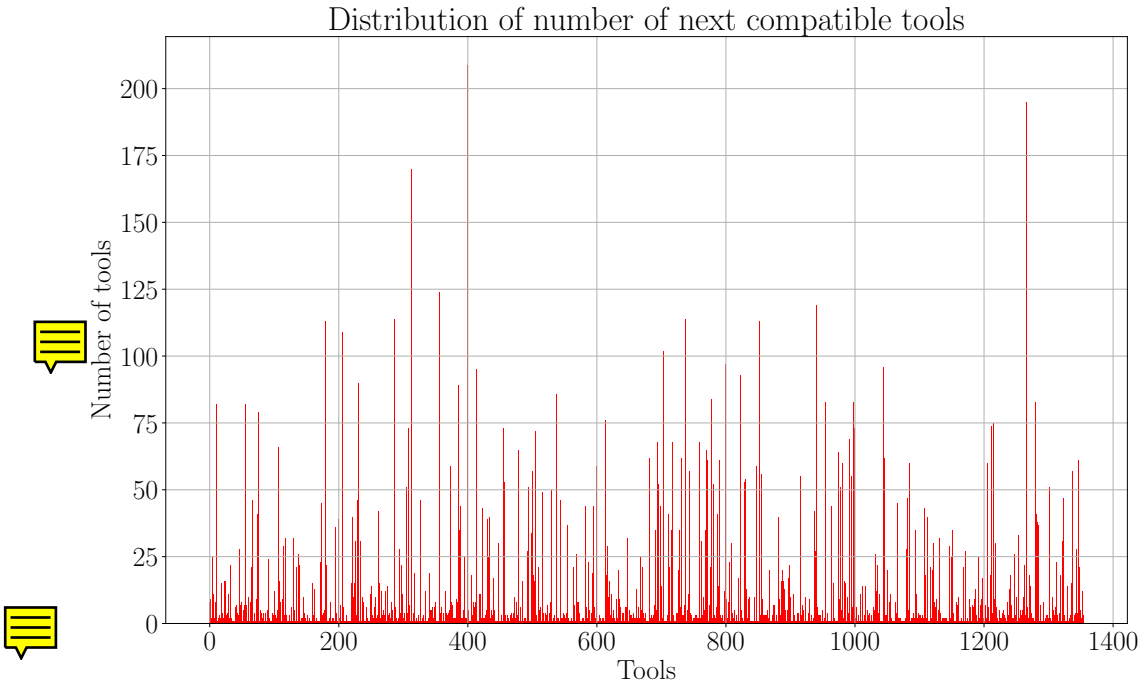

Figure 26.: Distribution of the number of compatible next tools: The bar plot shows a distribution of the number of compatible next tools for each tool. The x-axis shows tools which have at least one compatible next tool. The y-axis shows the number of compatible next tools.

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. This set of compatible next tools is computed from all the workflows analysed in this work.

9.4. 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 the workflow paths and whether the classifier is sensitive to these sizes.

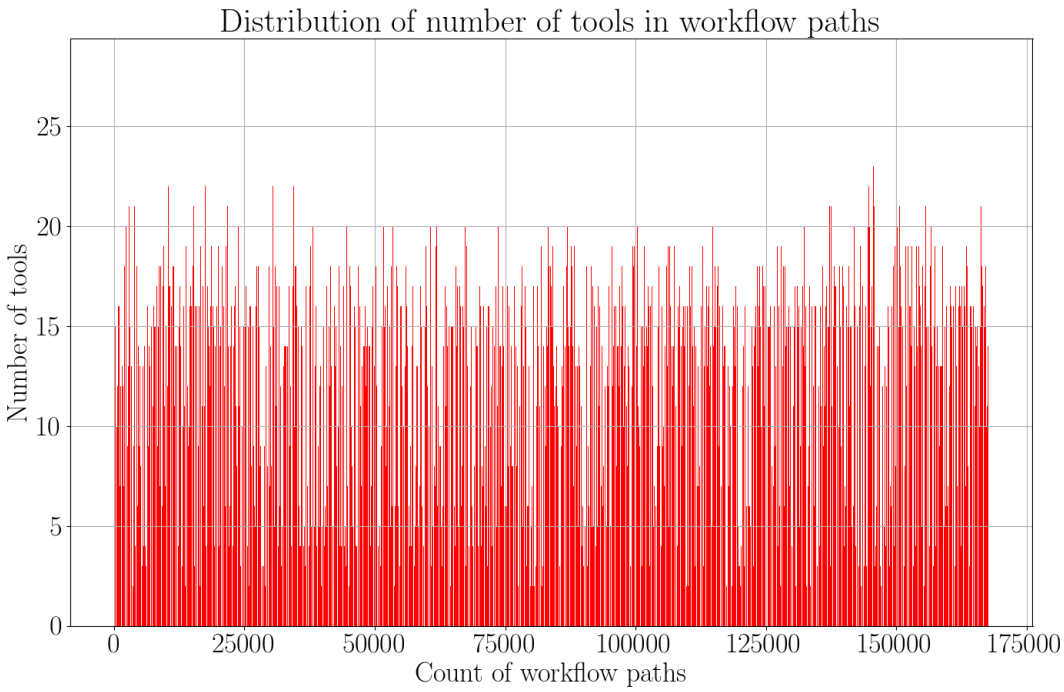

Figure 27.: Distribution of sizes of workflow paths: The plot shows a distribution of the sizes of workflow paths. A workflow path consists of multiple tools. This distribution shows how many tools are there for each path. Most of the paths contain less than 20 tools.

9.5. Learning

Figure 25 delineates the steps to predict next tools in workflows. They are decomposed into paths and then into tool sequences. Each tool sequence consists of a set of tools connected one after another.

9.5.1. 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 sequences are used. The decomposed tool sequences are further divided into training and test paths. The classifier uses the training paths to learn features. The amount of learning is evaluated on the test

paths. It is ensured that the intersection set of the training and test paths is empty. Otherwise, the evaluation of the learned model would be biased [19, 20, 21]. Different methods 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.

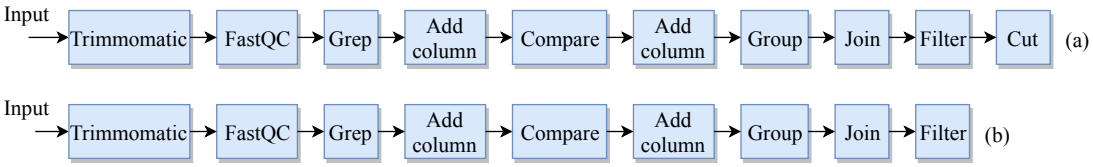

Figure 28.: No path decomposition: The image shows a workflow path. For a path of length n (a), a tool sequence of length $n-1$ (b) is taken out and the last tool is assigned as its next tool. For example, the tool *cut* is a next tool of the previous tool sequence.

Decomposition of test paths

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

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

Decomposition of test and train sets

By decomposing paths keeping the first tool fixed, learning task is made easier for a classifier. In the previous section, only the test paths are decomposed. But, in this approach, all the paths are decomposed. The strategy explained in figure 29 is used to obtain a mixture of shorter and longer tool sequences. For this approach as well, a dictionary of paths is created where each key represents a tool sequence and its value is a set of next tools. This dictionary is shuffled and divided into training and test paths.

9.5.2. 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 the nodes in a graph and also the conditional probabilities among them. When the number of nodes increases, it would 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, bayesian network is not used in this work for predicting next tools in workflows.

9.5.3. Recurrent networks

A workflow may have multiple paths and they can share a few tools. These paths are extracted from the workflows and the duplicates are removed. They are assumed to be independent of one another which keeps the analysis simple yet 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 tool or a tool sequence. Figure 30 shows these dependencies for a tool sequence. The tool *text reformatting* is not the only cause of tool *sort* but tools *datamash* and *concatenate datasets* as well. A classifier should model these dependencies for variable length tool sequences.

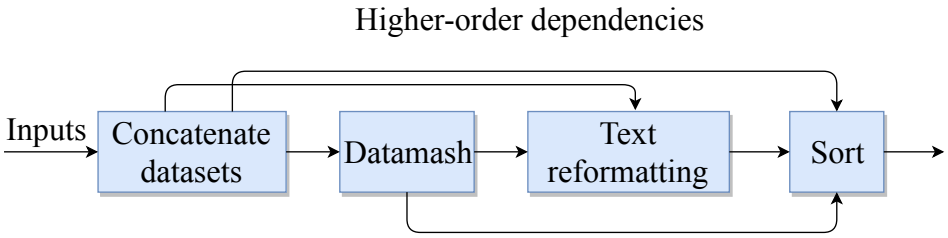

Figure 30.: Higher-order dependency: The image shows a small tool sequence where four tools are arranged in a sequential manner. Each tool depends not only on its immediate parent tool, but all the previous tools in a path.

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 tool sequences. It combines information from previous tools and the current tool. It is computed as:

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

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:

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

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:

$$p(x_1, x_2, ..., x_T) = \prod_{t=1}^{T} p(x_t | x_{t-1}, ..., x_1) \tag{33}$$

where x_t is the input at step t, $p(.)$ is the probability distribution and T is the length of a tool sequence (total time-steps). From equation 33, the input at step t is reliant on the previous inputs in a sequence. To predict next input (or next tool in a tool sequence), this conditional probability distribution 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). At each step, the final gradient is the product of gradients until that step. If the gradients are small which inherently means that the recurrent units are not capturing the higher-order dependencies, then the gradient can quickly slip towards 0 and disappear (the product of small numbers). However, in another scenario when the gradients are large numbers, then the product can easily explode to become a large number. This situation is known as vanishing/exploding gradients problem. The traditional recurrent units are prone to exhibit this behaviour [29]. In order to avoid these situations which hamper learning, two variants of the recurrent units are proposed:

- Long-short term memory units (LSTM) [30]

- Gated recurrent units (GRU) [28]

In this work, the gated recurrent units (GRU) is explored which is recently proposed and is simpler than the LSTM and contains less parameters. The performance of these two variants is comparable [17].

Gated recurrent units (GRU)

The GRU has gates which control the flow of information (figure 31). The reset gate r checks how much information from the previous time steps should be carried forward. If it is 0, then all the information from the previous time steps is discarded. If it is 1, all the information accumulated over the previous time steps is taken forward. It enforces how much of the information from the previous time steps would be forgotten. The update gate controls how much of the unit's own content would be used when computing the current memory content.

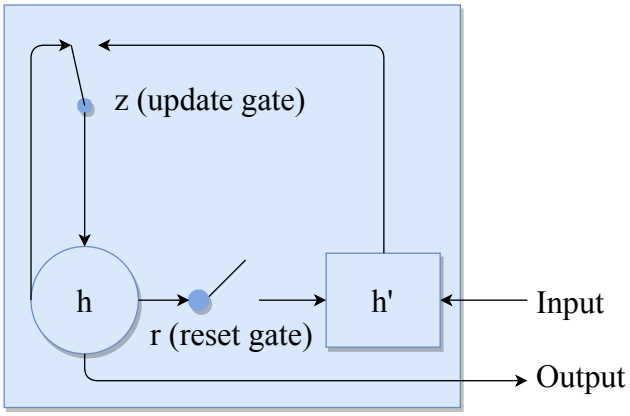

Figure 31.: Gated recurrent unit: The image shows a gated recurrent unit with two gates, r as a reset gate and z as an update gate. The activation of the GRU is h and the proposed activation is $h^{'}$ [17].

$$h_t = (1 - z_t) \times h_{t-1} + z_t \times h_t^{'} \tag{34}$$

where h_t and h_{t-1} are the current and previous step activations, 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:

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

where $\sigma(.)$ is the *sigmoid* function and W and U are the input and recurrent weights matrices. The current proposed activation is computed using:

$$h_t^{'} = \tanh(W \times x_t + U \times (r_t \odot h_{t-1}) \tag{36}$$

where $h_t^{'}$ is the proposed current 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 activation. The symbol $\odot$ is an element-wise multiplication between r_t and h_{t-1}. The *tanh* is an activation function and is given as:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{37}$$

The reset gate is given as:

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

where r_t is reset gate which controls how much of information from previous activations should be used at step t. W_r and U_r are the input and recurrent weights matrices. x_t is the current input at step t and h_{t-1} is previous state activation.

There are few important points to note here from equations 34-38:

- The output does not only depend on the current input but on the previous inputs as well. This is well managed by maintaining the memory of the previous inputs. This enables the network to extract the latent features present in the long sequences which lead to a specific output.
- The nonlinear activation functions, *tanh* and *sigmoid*, enable the values of the update gate (z_t), reset gate r_t and current state (h_t) to lie between 0 and 1. Using these values, a specific percentage of information is extracted by the update and reset gates.

9.6. Pattern of predictions

The classifier is designed in such a way so as to generate a probabilistic prediction of the next possible tools for a tool sequence. There are n unique tools which are used to create all the workflows. For each tool, the classifier should assign a probability score of being the next tool for a tool sequence. The dimensionality of the output should be n.

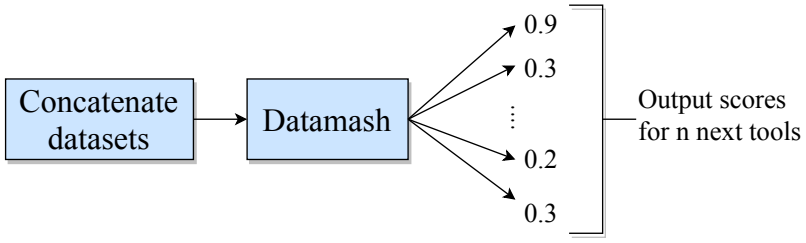

Figure 32.: Predicted scores for next tools: The image shows how the scores are assigned by the classifier to a set of n tools. These scores assigned to each tool varying between 0 to 1. Each value contains an importance of each tool to become the next tool of a given tool sequence. In the given example, there are two tools in the tool sequence. For the next tool, the importance scores of each tool of being the next tool are learned by the classifier. The predicted set of next tools are sorted in the descending order of their scores and a few top tools are selected.

9.7. The classifier

The recurrent neural network is used to classify tool sequences and predict next tools. This network contains several layers and they form a stack. The first layer consumes the tool sequences and the output layer gives out the predictions. The hidden recurrent layers do all the computations required to do the classification of input tool sequences. While classifying, the tool sequences are mapped to their respective next tools (classes). Figure 32 shows how the classifier assigns scores to all the tools of being the next tool of a tool sequence.

9.7.1. Embedding layer

The first layer in the network is an embedding layer. It represents a dense, fixed-length vector for each tool (figure 34e). An integer is assigned to each tool and it is converted to a vector (also called an embedding) by the embedding layer. An embedding is a vector-representation of an integer (tool's index). This embedding is not just a dense vector but represents features associated with that tool. It is a weight vector of a tool. It means that the embedding for a tool index encodes the information about the context in which the tool is being used. The tools which are used in similar context, their embedding vectors are similar.

9.7.2. Recurrent layer

Two hidden recurrent layers containing the gated recurrent units are used. These layers are responsible for doing all the computations given in the equations 34-37. The two hidden layers are stacked one after another with an equal number of the recurrent units in each layer. The stacking of layers allows the learning of deeper structures (features) that exist in the tool sequences. The hidden states of the memory units in the first layer become the input to the units of the next hidden recurrent layer. Figure 33 shows the recurrent network.

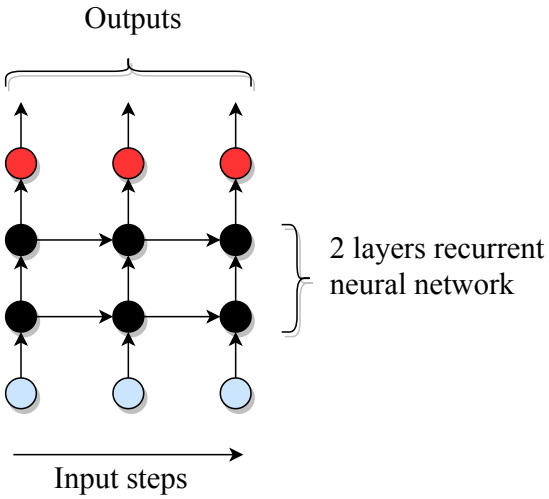

Figure 33.: Recurrent neural network: The image shows the stacked gated recurrent units layers and how they pass information from the input layer to the output. The output of the first recurrent layer is used as an input to the next recurrent layer. The blue circles denote the input steps (different tools in a tool sequence), the black ones form the two recurrent layers and the red circles form the output layer [31].

9.7.3. Output layer

The output is computed from the last hidden recurrent layer by applying a nonlinear function to its hidden states. The importance scores for all the tools for being the next tool for a tool sequence are computed (figure 30). To achieve that, the output layer has the same dimensionality as the number of tools. Each dimension holds a real number between 0 and 1 which can be understood as a probability of that tool to be the next tool for that sequence. The *sigmoid* function (equation 39) is used as an activation for this layer:

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

9.7.4. Activations

The 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 more[1]. It is chosen based on the kind of output that is required for the further evaluation. The output of a neuron is given by:

$$y^j = \phi(\sum_{i=1}^{n} w_i \times x_i + b)^j \tag{40}$$

where y is the output of the neuron j, w_i is the weight of the i^{th} input connection to the neuron j, x_i is the input and b is the bias for neuron j. ϕ is an activation function. The *sigmoid* is used as an output activation because all the outputs (next tools) are independent of one another and their importance factors should be learned separately. The *softmax* activation can replace the *sigmoid* activation, but it normalises the output values which is undesired. The *sigmoid* gives a positive real number between 0 and 1 which is understood as a probability of being the next tool. Each tool is assigned a real number between 0 and 1 at the output layer. Another activation is the exponential linear unit (ELU). It is used as the activation for the recurrent layers. In equation 36, the *tanh* is replaced by the ELU activation. It is given by:

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

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

[1] `https://keras.io/activations/`

9.7.5. Regularisation

Overfitting is a common phenomenon in the machine learning algorithms. It occurs when a classifier starts memorising the training data without learning the general features from the data. It leads to an increased learning on the training data but no learning on the test data. The error on the train data decreases but the error on the test data either stops decreasing or sometimes increases. The classifier stops generalising and would predict new data with an increased uncertainty. The neural network is used in this work for the classification task. It is prone to overfitting as it tends to derive a complex model. If the size of the training data is not enough, the network starts overfitting. To generate a model which learns the general features from the training data, it is important to apply measures to remove or reduce overfitting. The regularisation is a technique to overcome this common problem in the neural networks. 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, as the name suggests, removes connections momentarily from the neural network and thereby changing the network structure during each weight update. It randomly sets the output of some connections to 0. It leads the network to behave in a novel manner as some of the randomly chosen connections stop their emissions. Due to this, the network becomes less powerful and it stops picking bias from the training data. A real number between 0 and 1 is specified which sets the percentage of neurons or units to be dropped. These units are chosen randomly. The dropout is applied to the input, output and recurrent connections of the 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 drop in the training and validation losses remains stable and close to each other [34, 35]. It depends on the complexity of the network and the amount of data.

9.7.6. Optimiser

An optimiser is used to minimise the loss computed by a loss function. Mini-batch RMSProp optimiser is employed to find the optimal weights for the features which minimise the error on the training data. RMSProp is a variant of the stochastic

gradient descent with an adaptive strategy for the learning rates [36]. It adapts the learning rate according to the gradients. It keeps a track of the previous gradients and updates the learning rate by dividing with an average of the square of the previous gradients. This average decays over time in an exponential manner.

$$MeanSquare(w,t) = 0.9 \times MeanSquare(w,t-1) + 0.1 \times \frac{\partial E}{\partial w}(t) \tag{41}$$

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

where $MeanSquare(w,t)$ is the mean of the squared gradients until time t, $\frac{\partial E}{\partial w}$ is a partial derivative of the error function (gradient) with a parameter w at time t and η is the learning rate.

Learning rates

Learning rate is an important parameter for an optimiser which determines the amount of update at each time step. If the learning rate is high, there is a risk of an optimiser divergence. Instead of going to the minimum of the error surface, which is the expected behaviour, the optimiser keeps oscillating on the error surface bypassing the minimum. On the other hand, if the learning rate is low, the minimum may never be reached as the learning steps become too small. The optimiser does not converge to the minimum or take a large amount of time to converge. Due to these reasons, setting a good learning rate is the key to find the optimal weights in a reasonable amount of time.

Loss function

Binary cross-entropy[2] is used as the loss function for the optimiser. It is well-suited for the multi-label classification. The multi-label classification is a kind of classification of data with multiple outputs. For this work, a tool sequence can connect to more than one tool and due to this, there can be multiple next tools for a tool sequence. This loss function is given by:

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

[2] `https://github.com/keras-team/keras/blob/master/keras/backend/tensorflow_backend.py`

where N is the total number of next tool positions, y_i is the i^{th} tool's actual value, p_i is the predicted value for the i^{th} tool. The prediction each next tool is independent of the other next tools. The values in actual next tools vector y is either 0 or 1. The predicted vector p_i contains positive real numbers between 0 and 1 for each position (due to the *sigmoid* activation). The mean loss will be low when the vector of the predicted next tools is comparable to the actual next tools vector. The summation is always negative or 0 which makes the loss to be a positive real number or 0.

9.7.7. Precision

The last tool of each workflow path is treated as its next tool. For a path of length n (with n tools), the tool sequence has a length of $n-1$ and its next tool is the n^{th} tool. In this way, the tool sequences and their next tools are arranged. For many of these tool sequences, there is more than one next tool. The performance of each training epoch is assessed by computing all the predicted next tools (which are as many as the actual next tools). All the predicted tools are matched either with the actual next tools or compatible next tools to compute how many of the actual or compatible next tools are predicted correctly by the classifier for a tool sequence. The precision is computed for each tool sequence in the test data (equation 43). It is averaged over all the tool sequences in the test data to give average precision for each training epoch.

$$precision^j = \frac{1}{N} \times \sum_{i=1}^{N} y_i \tag{44}$$

where the $precision^j$ is computed for the j^{th} tool sequence, N is the total number of next tools for the j^{th} tool sequence, y_i is the accuracy for the i^{th} predicted tool. The prediction vector y is sorted in the descending order. It takes 1 if the i^{th} predicted tool is present in the set of the actual or compatible next tools and 0 if not. The precision computes the fraction of the correctly predicted next tools in the actual or compatible next tools.

10. Experiments

The workflows are collected from the Galaxy's main[1] and Freiburg servers[2]. There are $\approx 900,000$ paths in these workflows. Out of all these paths, $\approx 167,000$ paths are unique. These paths are sequential in which a tool is dependent on all its previous tools (figure 30). The duplicate paths are removed. They are divided into the training and test paths. The recurrent neural network is trained on the training set (80%) and evaluated on the test set (20%). A small part of the training set (20%) is reserved as a validation set to find the right values of hyperparameters. This set is used during training to compute loss on an unseen set of paths. The *bwForCluster*[3] provides the computing resources for processing the workflows and training and evaluating the recurrent neural network.

10.1. Decomposition of paths

The paths are decomposed following the strategies explained in section 9.5.1 and the performance is measured separately for each approach. In one approach, no path is decomposed into the smaller tool sequences. The classifier is trained and tested on the actual paths from the workflows. In the second approach, only the paths from the test set are decomposed as described in figure 29. The classifier is trained on the actual paths. In the third approach, both the training and test paths are decomposed as described in figure 29. The classifier is trained using a mixture of smaller and longer paths. The configuration of the classifier is kept same for all the three approaches for the training and testing phase.

[1] `https://usegalaxy.org/`
[2] `https://usegalaxy.eu/`
[3] `https://www.bwhpc-c5.de/wiki/index.php/Main_Page`

10.2. Dictionaries of tools

The names of tools present in paths cannot be used by any classifier to learn on. They need to be converted into numbers. To do that, a dictionary is created where each tool is assigned an integer. The tool names are replaced in the paths by their respective indices in the dictionary. Each path becomes a sequence of integers (figure 34a and 34c). In addition, a reverse dictionary is created as well to replace any tool index by the corresponding tool name.

10.3. Padding with zeros

The paths divided into the training, test and validation sets have variable sizes. Some of the paths are short while the others are long. But, the classifier takes only a fixed-size input. To deal with this issue, a maximum length of the paths is set to 25 which captures all the paths from the workflows. Figure 27 shows that the number of tools contained by all the paths is less than 25. For the shorter paths, an extra padding is added with zeros in the beginning to ensure that all the paths have the same length. Moreover, it is undesirable that the classifier learns any feature from these padded zeros. To avoid this, a flag is set (in the implementation of the embedding layer of the network) to mask these streams of zeros present in the paths. The classifier considers only the useful indices in a path. As the zero is chosen for the padding, it does not represent any tool in the dictionary.

Figure 34 shows that how a tool sequence and its possible next tools are transformed into the respective vectors. These vectors are used by the classifier in the form of the training and test paths. Figure 34a shows a tool sequence with three tools arranged in an order as it would appear in any workflow. Figure 34b shows the next tools for this tool sequence. The section 10.2 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 with zeros is also shown followed by the corresponding indices of the tools in the tool sequence (figure 34c). In each vector, the padding of zeros precedes the sequence of tool indices. Each index is further transformed into the fixed-length vector (embedding) as shown in figure 34e. The length of this embedding vector remains the same for all the tools and is defined by the size of the embedding layer (512). 512 dimensional vector is learned for each tool's index and arranged as shown in figure 34e. The order of the tools is maintained as present in the tool sequence

in figure 34a. Figure 34d shows how the next tools vector is arranged. A vector of zeros with the size equal to the number of tools is taken. The corresponding indices of the next tools are set to one in this vector. For example, "AddValue" tool has an index of 4 in the dictionary. Therefore, the fourth position of the next tool vector (figure 34d) is set to one. This is repeated for all the possible next tools for a tool sequence. Figure 34d contains two positions set to one representing the next tools for the tool sequence (figure 34a).

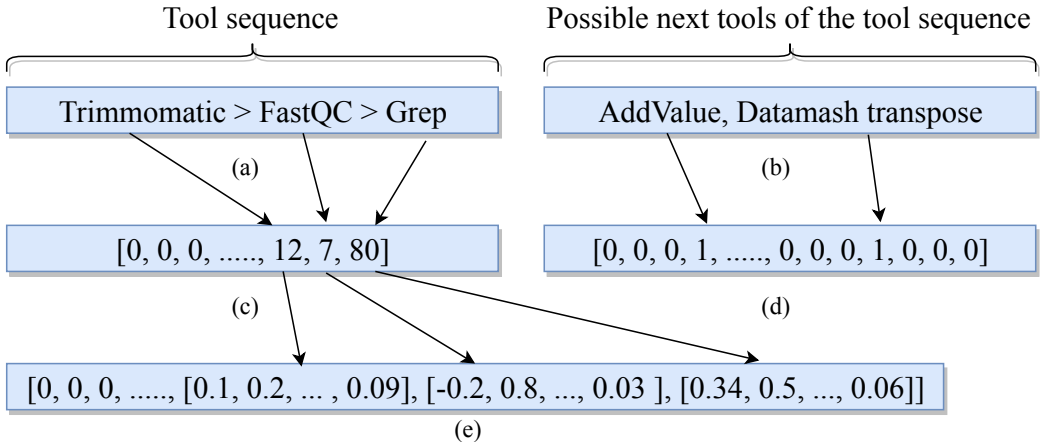

Figure 34.: Vectors of tool sequence and its next tool: The image shows how the vectors are created for a tool sequence and its next tools in order to make them available for the classifier. Figure 34a shows a tool sequence with three tools. 34b shows its next tools. Figure 34c shows the arrangement of padding 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 network. The size of the embedding layer defines the length of this embedding vector.

10.4. Network configuration

The gated recurrent unit, a variant of the recurrent neural networks, is used as the classifier. The recurrent neural network has several layers and many hyperparameters. An embedding layer is used as an input layer which learns a fixed-size vector (128) 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. The dropout is applied to the output of this layer in order to reduce overfitting. Two hidden recurrent layers with the gated recurrent units are used to model the paths. Each layer has a fixed number of memory units. The number of memory units specifies the dimension of the hidden state (equation 34). The dropout is applied to the inputs, outputs and recurrent connections for each hidden recurrent layer. The amount of dropout remains the same for all the different places in the network. The last layer is a dense layer (fully connected layer) and has the size equal to the number of tools. This is because, for each path, the network generates the scores for all the tools to be the next tool of a tool sequence. The tools having higher scores at the output layer possess the higher probability of being the next tools.

10.4.1. Mini-batch learning

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

10.4.2. Dropout

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

10.4.3. Optimiser

A few optimisers like the *stochastic gradient descent (SGD)*, *adam*, *rmsprop* and *adagrad* are used to find out which provides a stable learning and enables the network

to achieve the best precision. All the other parameters are kept fixed. They all minimise the cross-entropy error between the actual and predicted next tools. The default configurations of these optimisers are used as set by the keras library[4].

10.4.4. Learning rate

A range of learning rates is tried out to find the stable learning pattern in the network. It ranges from 0.0001, 0.005, 0.001 and 0.01. A smaller learning rate tends to slow down the convergence of the optimiser while a higher rate can diverge it. It is important to avoid both the situations to ensure a stable learning. All the other parameters are kept fixed.

10.4.5. Activations

There are multiple choices of activation functions like *tanh*, *sigmoid*, *relu* and *elu*. These different activations are used one by one to find which one works the best. The *tanh* is the default activation while *elu* is one of the recently proposed activations. All the other parameters are kept fixed.

10.4.6. Number of recurrent units

The hidden recurrent layers need memory units to learn from the workflow paths. This number specifies the dimensionality of the hidden state. To choose a size which expresses these hidden states to ensure better learning, several sizes like 64, 128, 256 and 512 are used one by one. All the other parameters are kept fixed.

10.4.7. Dimension of embedding layer

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

10.5. Accuracy

The classifier's accuracy is reported as the precision on the test set over multiple epochs of learning on the training set. The training is done for 40 epochs. The

[4] `https://github.com/keras-team/keras/blob/master/keras/optimizers.py#L209`

learning saturates and the training and validation losses do not decrease anymore. It means no more learning is possible and the training should be stopped. The weights saved for the last epoch is used for making the predictions for the paths present in the test set.

11. Results and analysis

In this section, the performance of the classifier is visualised and discussed on the three different approaches of decomposition of paths. Various parameters like the optimiser, learning rate, activation, batch size, number of memory units, dropout, embedding dimension and length of tool sequences are the hyperparameters of the network. The top-1 and top-2 accuracies are also compared. A slightly different neural network with only dense layers is also used to show a performance comparison with the recurrent neural network with the gated recurrent units.

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 tool sequences 25
- Test set is 20% of the complete set of paths
- Validation set is 20% of the training set
- Activation is ELU
- Output activation is sigmoid
- Loss function is binary cross-entropy

These parameters remain the same for all the three approaches (from the section 9.5.1) and define the baseline configuration of the network. By experimenting with the different values of these parameters, the possibilities to improve the classification performance are explored.

11.1. Notes on plots

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

- The x-axis shows the number of training epochs. For the subplots (a) and (b), the y-axis shows the average precision and for the subplots (c) and (d), the y-axis shows the cross-entropy loss.

- The subplot (a) depicts absolute precision. For example, if a tool sequence has five actual next tools, the top five predicted next tools are taken out from the predictions. If out of the five predicted next tools, only four of them match then the absolute precision for this tool sequence is $\frac{4}{5}$. The average precision is computed over all the tool sequences in the test set. It is done at the end of each training epoch.

- The subplot (b) shows the compatible precision. While computing the absolute precision, some false positives (wrong next tools) are also predicted which are not present in the actual next tools for a tool sequence. In this case, these false positives are matched with the compatible set of tools belonging to the last tool of the tool sequence. If some or all of the false positives are present in this compatible set, then they add up to the absolute precision to give the 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 tool sequence. If it is present, then the compatible precision is 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 the training set. It is computed using equation 43 by the classifier.

- The subplot (d) shows the cross-entropy loss on the validation set. The last 20% of the training set makes the validation set. The loss is computed by the classifier after training on the first 80% of the training set.

11.2. Performances of different approaches of path decomposition

11.2.1. Decomposition of only test paths

In this approach, the paths in the test set are decomposed keeping the first tool fixed as explained in figure 29. The paths in the training set are kept as such. The results are shown in figure 35. The motive here is to make the classifier learn the tools connections from the longer paths. The trained model is tested on the shorter paths. This approach models the real application of the work - training on the longer paths and predicting on the shorter paths. But, as the result suggests, the learning does not happen as desired. The absolute precision is worse than the compatible precision. The absolute precision saturates $\approx 22\%$ at 30^{th} epoch while the compatible precision measures $\approx 43\%$ (figures 35a and 35b). However, the cross-entropy loss for the validation set drops in a steady manner and saturates (figures 35c and 35d). The learning still happens but only for the training set (longer paths) and not for the test set (shorter paths) as the precision stops improving. The overall training and evaluation time was ≈ 28 hours. Each epoch took ≈ 20 minutes for training.

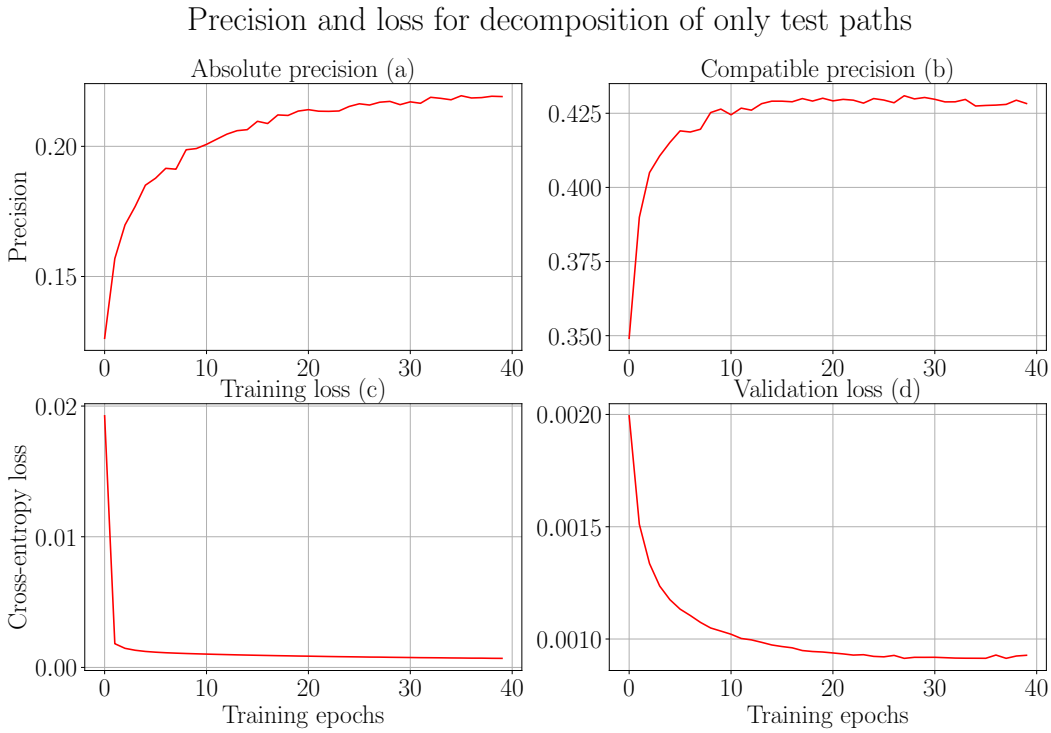

Figure 35.: Performance on the decomposition of only test paths: The plot shows the classification performance for the approach where only the test paths are decomposed keeping the first tool fixed. The paths in the training set are kept as such. The idea is to train a classifier on the longer paths and test the trained model on the shorter paths. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training.

11.2.2. No decomposition of paths

In this approach, the paths from the workflows as used as such for both, the training and test paths. The classifier is made to learn and evaluate on the longer paths. The last tool in each of these paths is used as the next tool. The results in figure 36 are encouraging. The subplot 36a reaches the precision of $\approx 89\%$ when it reaches saturation 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 the same. It took overall less time compared to the previous approach because of the smaller size

of the test set.

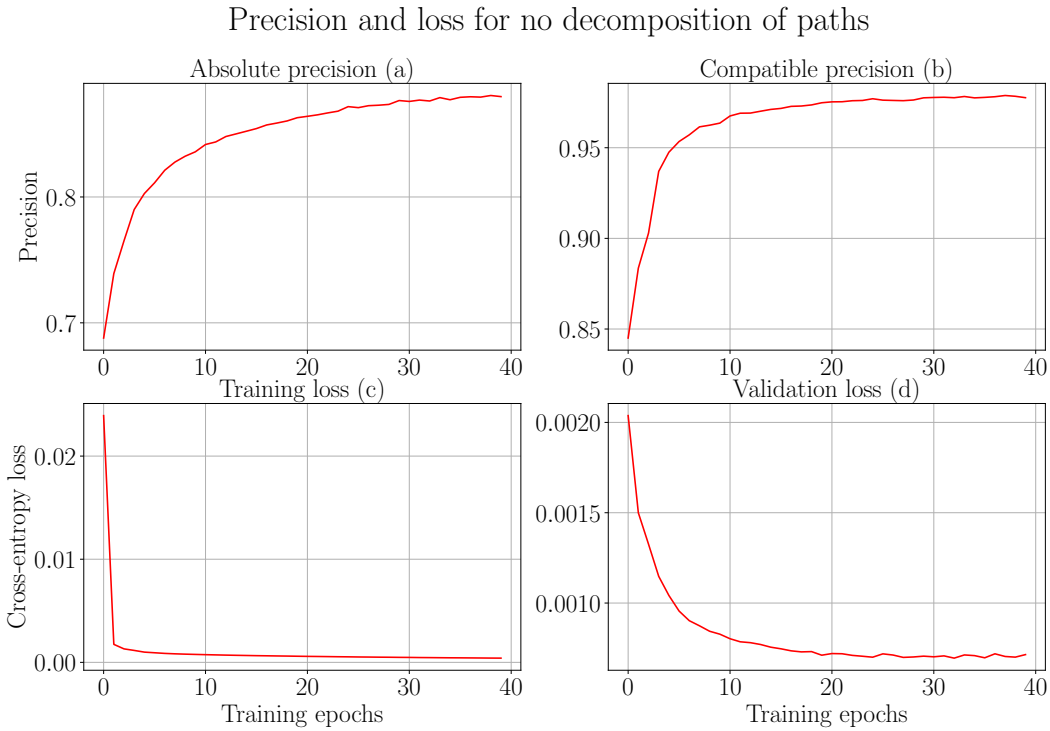

Figure 36.: Performance on no decomposition of paths: The plot shows the classification performance for the approach where no paths are decomposed. The idea is to train and test the classifier on the longer paths. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training.

11.2.3. Decomposition of the train and test paths

All the paths, in train and test paths, are decomposed keeping the first tool fixed. The last tool of each path (of length n) becomes the next tool of the $n-1$ length tool sequence. The idea here is to make the classifier learn on the shorter as well as longer paths. The results can be seen in figure 37. The absolute precision is $\approx 90\%$ while the compatible precision is $\approx 99\%$. The results are more encouraging than the previous approach. The test set is decomposed in the way how this work would be used practically. 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, the results of this approach are more practical than the previous approach. The paths are decomposed

to have many shorter paths as well. Due to this, the size of the training set increases. It implies more training time is required. The training along with the evaluation of the test set finished in ≈ 48 hours. The training for each epoch took ≈ 45 minutes.

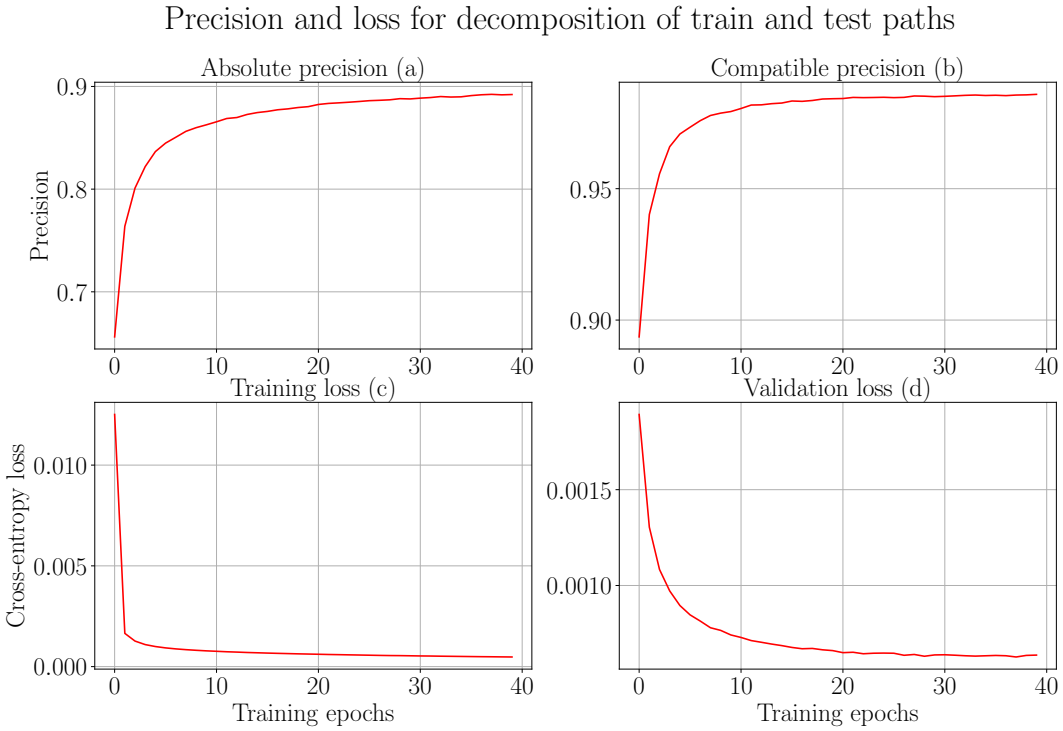

Figure 37.: Performance on the decomposition of all paths: The plot shows the classification performance for the approach where all the paths are decomposed keeping the first tool fixed. The idea is to make a classifier learn on the shorter paths. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training.

11.3. Performance evaluation on different parameters

The recurrent neural network has many hyperparameters and they need to be tuned to the amount of data and to each other so that the network learns and predicts in a reliable way. Their right combination is needed to attain a reasonable accuracy and to avoid too much learning (overfitting) and too less learning (underfitting). These hyperparameters include different optimisers, learning rates, activations, batch sizes,

number of recurrent (memory) units, dropout and the dimensions of embedding layer. There are a few metrics on which the evaluation is done. These include the top-k accuracy and the length of tool sequences. A deep network with only dense layers is also used as a classifier to compare the classification performance with the recurrent neural network.

11.3.1. Optimiser

Four optimisers are used to find out which one works best. Optimiser like the stochastic gradient descent (SGD) [36], adaptive sub-gradient (adagrad) [36], adam [38] and root mean square propagation (RMSProp) [36] are used for the analysis. All the other parameters are kept constant. From figure 38, it is concluded that the SGD performs the worst on both the metrics, precision and loss. The absolute and compatible precision do not improve and the drop in loss curve starts very late during the training. The SGD starts off with a high loss value (0.65) and does not drop much within the 40 epochs of training. Out of the remaining three optimisers, the RMSProp performs the best. The performance of the adam is comparable to the RMSProp. The adam optimiser catches up with the precision measured by the RMSProp, but slowly. Their performances converge later in the training. The plot shows that the choice of the RMSProp as an optimiser for the network is beneficial for the learning. In general, RMSProp is a good choice for the recurrent networks [39]. The bad performance of the SGD may be attributed to its non-adaptive parameter update method. The update to the parameters does not adapt to the gradients which is an important feature in the adaptive optimisers (RMSProp and adam). The adaptive optimisers adjust the parameter update with the gradients of the previous steps.

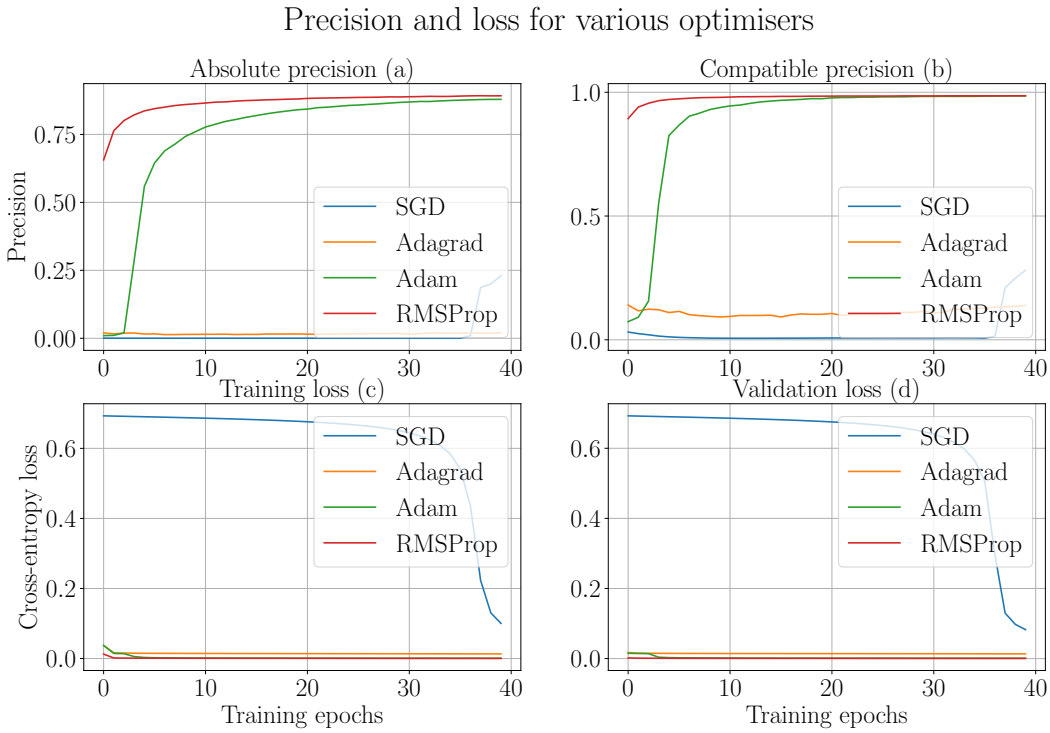

Figure 38.: Performance of different optimisers: The plot shows the performance of different optimisers. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training. The four optimisers are the stochastic gradient descent (SGD), adaptive sub-gradient (adagrad), adam and root mean square propagation (RMSProp). All the other parameters of the network are kept constant for the comparison.

11.3.2. Learning rate

Multiple values of the learning rate from as small as 0.0001 to as high as 0.01 are used to ascertain their effect on the learning. The performance obtained by using these different learning rates while keeping the other parameters constant can be seen in figure 39. A higher learning rate (0.01) diverges the learning as the precision drops during the training. The training loss increases and its curve has sharp edges. For the validation loss as well, there is not a 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 the precision drops slightly towards the end of the training. The smaller values 0.001 and 0.0001 help in learning as the precision improves and the

loss drops during the entire learning in a steady way. 0.0001 ensures learning but it is slower and would need more epochs (and more time) to converge. 0.001 works in the best way out of all these values on both the metrics. Therefore, the baseline network uses 0.001 as the learning rate for the RMSProp optimiser.

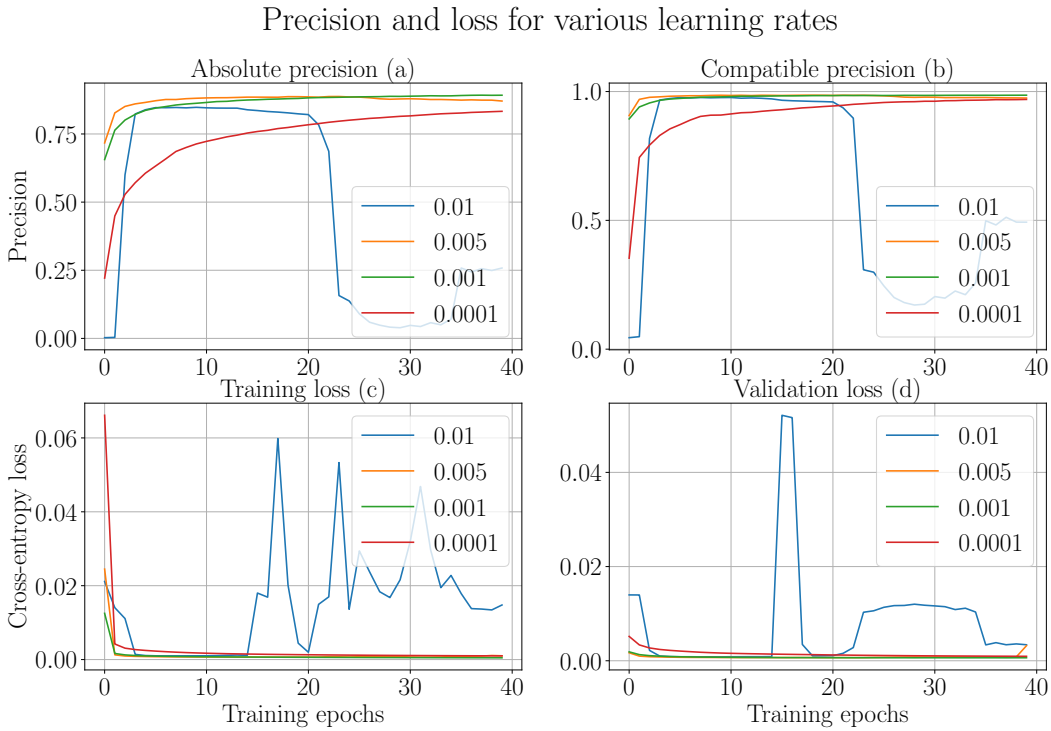

Figure 39.: Performance of multiple learning rates: The plot shows the performance of different values of learning rate. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training. The high learning rates do not provide stable learning while the smaller one slows down the learning.

11.3.3. Activation

Many activation functions are tested to find which one performs the best. Activations like *tanh*, *sigmoid*, *relu* and *elu* are utilised. Figure 40 shows the performances of these different activation functions. The activation functions *tanh*, *relu* and *elu* perform better than the *sigmoid* activation on both the metrics, precision and loss. The activation functions *relu* and *elu* perform close to each other on the precision as well as on loss. For the baseline network, *elu* is used as the activation function for the hidden recurrent layers.

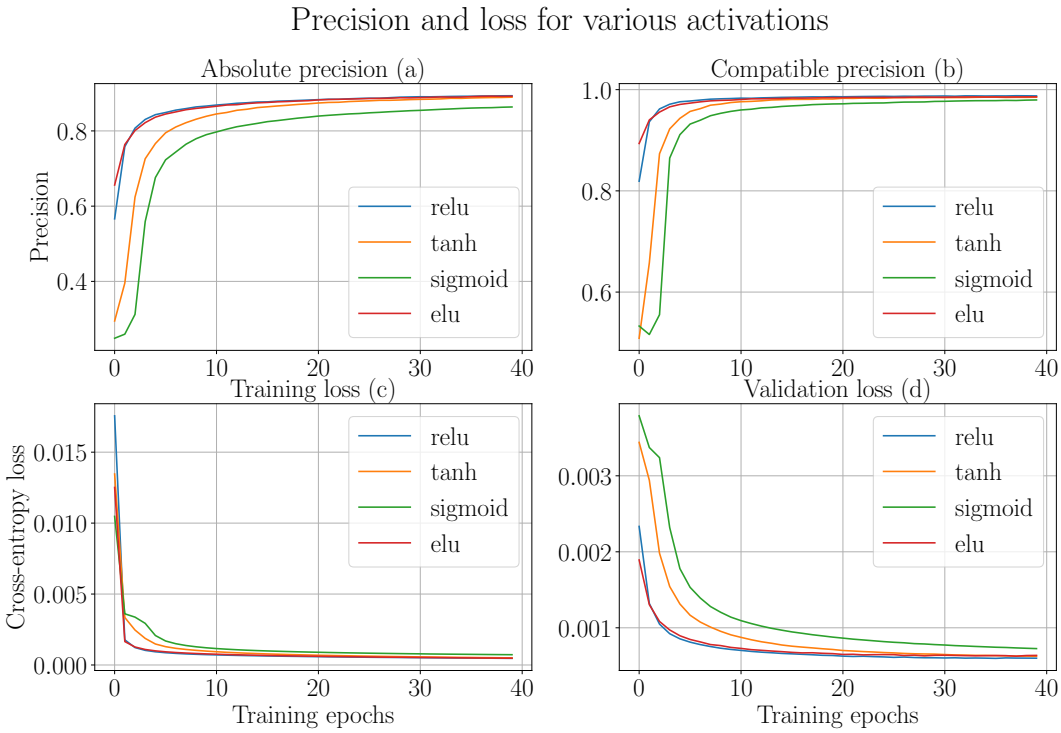

Figure 40.: Performance of multiple activation functions: The plot shows the performance of different activation functions. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training.

11.3.4. Batch size

Many batch sizes are used to ascertain which one works to provide a stable learning. Different numbers like 64, 128, 256 and 512 are tried out as the mini-batch size. In the mini-batch optimisation, a set of paths of size equal to the mini-batch size is taken and an average update is computed using these paths. This average update approximates the update for the whole set of paths. A smaller number would add more noise to the update because this smaller set would capture less variance from the whole set. The performance on various batch sizes is shown in figure 41. It is deduced from the plot that the batch size 64 is not performing as good as the other larger sizes. In figure 41c, the training loss starts to increase instead of decreasing. This states that it is not the right choice of the batch size. As the batch size is increased, the precision improves and the loss drops. The drop is higher for the batch size 512 for the validation loss compared to the training loss. The baseline network uses 512 as the batch size but the batch size 256 looks more promising and can be

used.

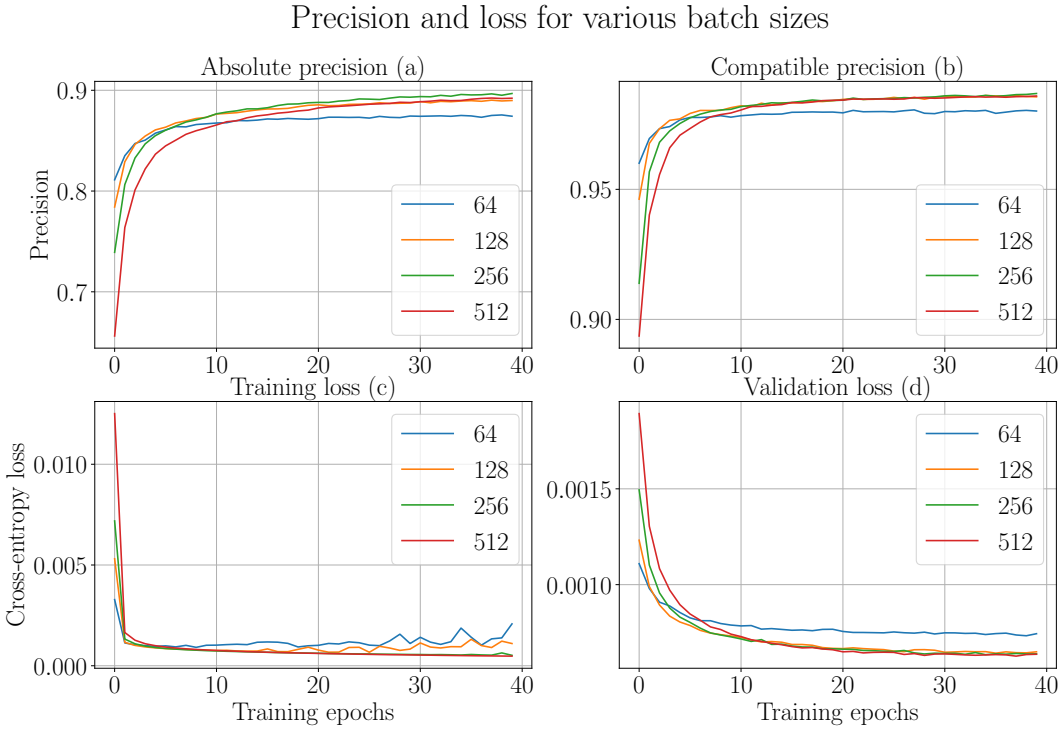

Figure 41.: Performance of different batch sizes: The plot shows the performance of different batch sizes. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training. The batch size 64 does not perform well compared to the larger sizes of the mini-batch.

11.3.5. Number of recurrent units

Three different numbers for the recurrent units are tried out to see which one achieves the best performance. This number specifies the dimensionality of the hidden state. Higher the number, more expressive the model becomes. It means that the model's prediction strength or the "memory" of the model increases. This behaviour can be seen in figure 42. As the number of units increases, the precision becomes better and the loss drop increases. 512 number of units performs the best out of the four choices taken. But, the increasingly strong model tends to overfit and tries to memorise the training set. Combating overfitting is necessary when the network becomes strong. Using a higher number of memory units also increases the training time. With 64 as the number of memory units, the training time for each epoch is ≈ 6 minutes while

with 512 memory units, each epoch takes ≈ 45 minutes.

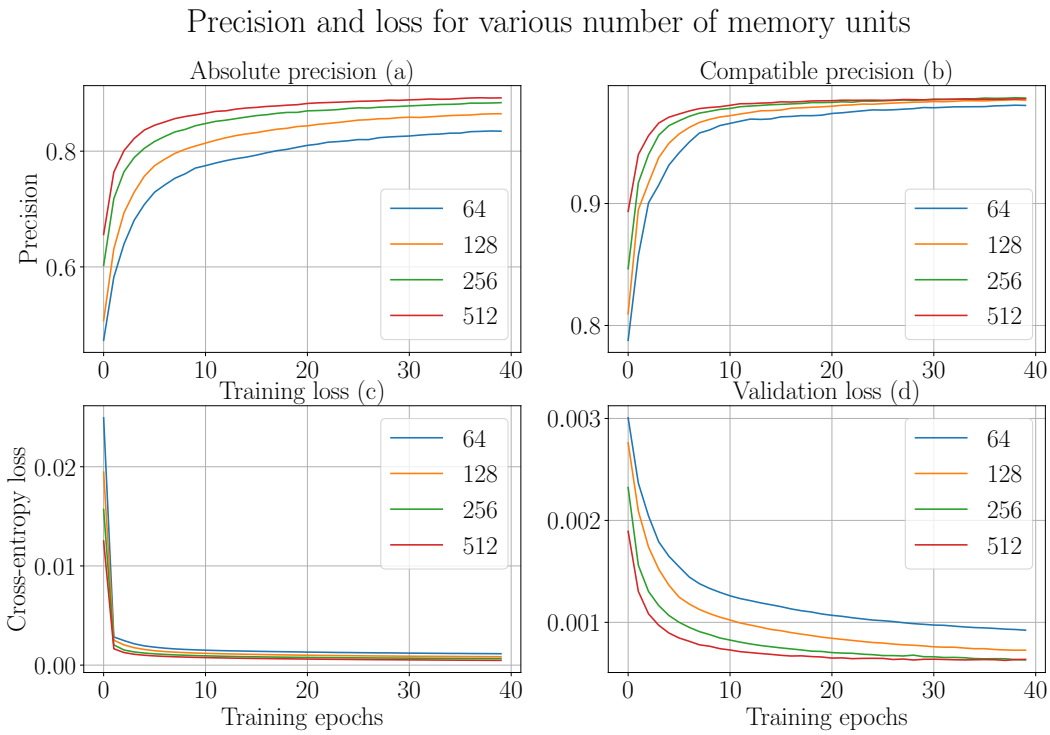

Figure 42.: Performance of multiple values of memory units: The plot shows the performance of different memory units. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training. 64 memory units for each recurrent layer do not perform well compared to the larger number like 256 or 512. The training time increases with the number of memory units.

11.3.6. Dropout

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

used, the network becomes weaker and due to this, the precision increases slowly. 0.2 gives the best choice of dropout as it achieves the best precision in both the categories and the drop in the training as well as in the validation losses is more robust out of all the dropout values chosen. The baseline network configuration uses 0.2 as the dropout. A dropout 0.3 performs close to 0.2 and can be used as well. Higher the value of dropout, larger is the training time. Using no dropout approach takes ≈ 33 minutes for training one epoch while with 0.4 dropout, it takes ≈ 37 minutes [40].

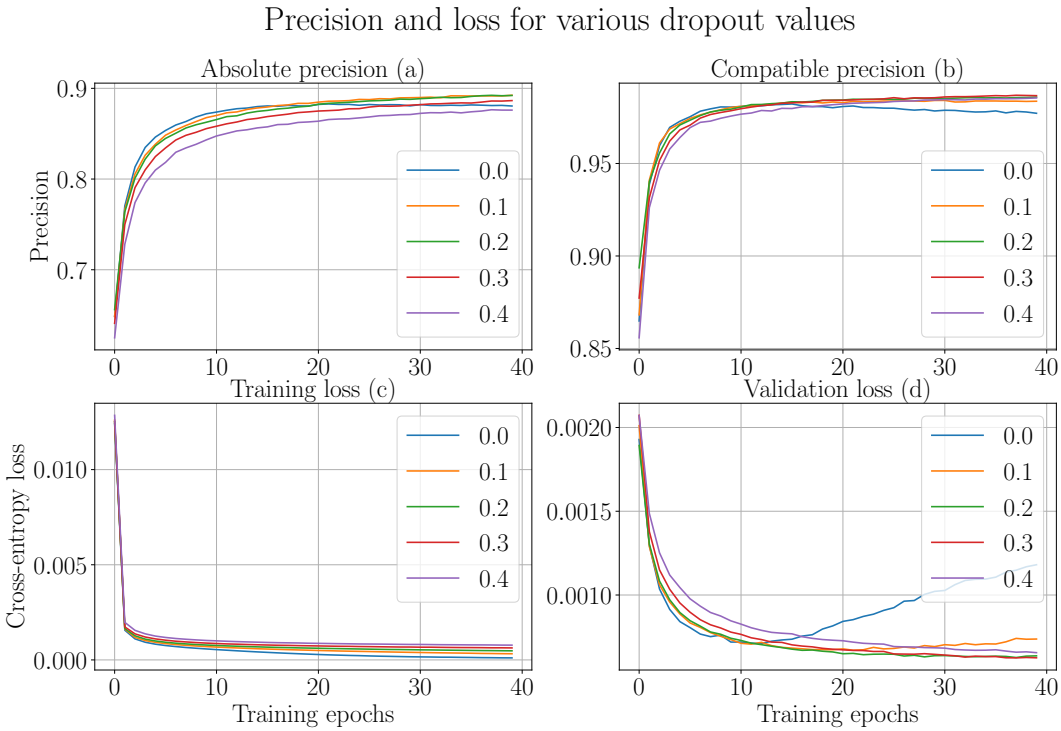

Figure 43.: Performance of multiple values of dropout: The plot shows the performance of different values of dropout. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training. Using no dropout shows overfitting as the validation loss increases during the training (d) while a higher dropout (0.4) achieves lower precision.

11.3.7. Dimension of embedding layer

Embedding layer learns a fixed-length, unique dense vector for each tool. Larger the size of this layer, higher is its expressive power to distinguish among tools. But, with

the higher dimensions, there is a risk of overfitting and with a smaller dimension, underfitting can occur. Figure 44 shows the performance with the different sizes of the embedding layer. All these sizes perform close to one another. The larger size performs slightly better than the 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 the embedding layer. For the size 64, it takes ≈ 30 minutes for the training of one epoch while with $1,024$, it takes ≈ 45 minutes.

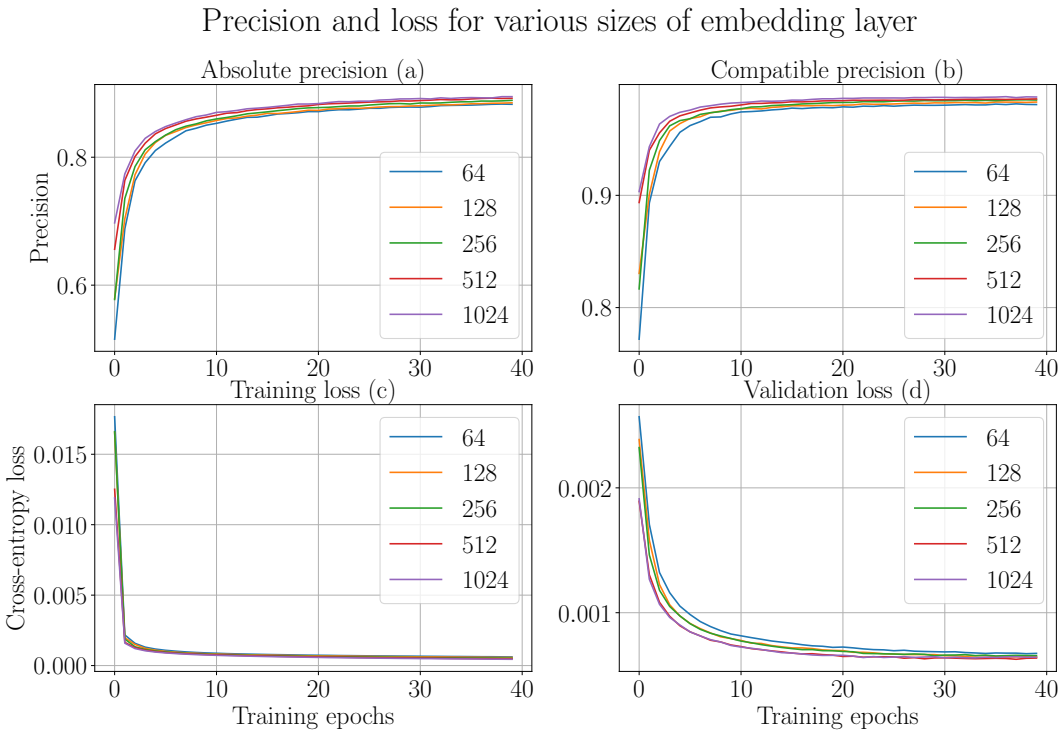

Figure 44.: Performance of different dimensions of embedding layer: The plot shows the performance of different dimensions of embedding layer. The subplots (a) and (b) show the precision while (c) and (d) show the cross-entropy loss change over multiple epochs of training.

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

The precision is given as one performance metric by the network (figure 37) by training over 40 epochs. The performance of the classifier can be verified on other metrics top-k accuracy as well. The top-k (k is an integer and satisfies $k \geq 1$) accuracy is computed (figure 45) for both the metrics, absolute and compatible (section 11.1). The accuracy metric (absolute top-k) computes how many of the k

predicted next tools are present in the set of actual next tools of a tool sequence. All the predicted tools are sorted in the descending order of their scores learned by the network for a tool sequence. Then top-k next tools are extracted. The top-1 and top-2 accuracies are computed for the absolute and compatible metrics. For example, the absolute top-2 accuracy shows that how many of the predicted two next tools are present in the actual next tools of a tool sequence. An average of the absolute top-2 is computed for all the tool sequences. A similar approach is followed for computing the compatible top-k accuracy,

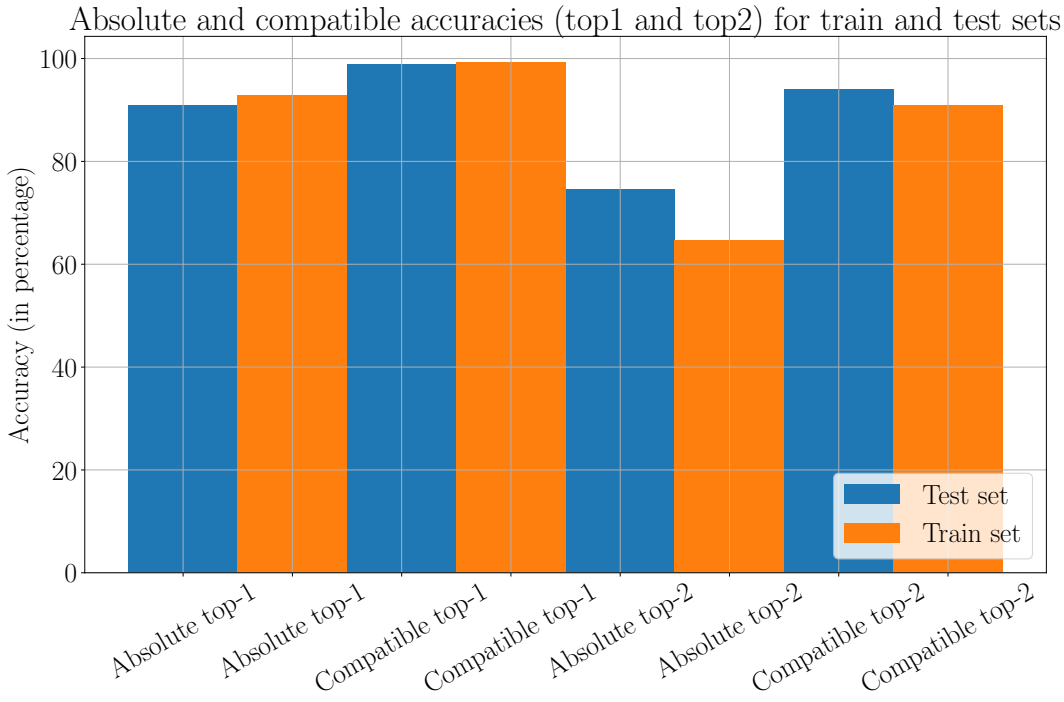

Figure 45.: Absolute and compatible accuracy (top-1 and top-2): The plot shows the top-1 and top-2 accuracies for the absolute and compatible metrics. For each tool sequence, the predicted tools are sorted in the descending order of their scores (learned by the network). The absolute top-1 accuracy measures if the predicted next tool with the highest score is present in the set of actual next tools. This accuracy is averaged over all the tool sequences. Similarly, compatible top-1 measures if the predicted next tool with the highest score is present in the set of compatible next tools. This accuracy is also averaged over all the tool sequences. The bar plot shows that the performance is comparable for the training and test paths.

Figure 45 shows that the performance remains similar for the training and test paths in the absolute and compatible accuracy metrics. Absolute top-1 selects the next tool with the highest score (computed by the network) and checks whether this next tool is present in the set of actual next tools of a tool sequence. This accuracy is then averaged for all the tool sequences to get absolute top-1. The compatible top-1 selects the next tool with the highest score (computed by the network) and checks whether this next tool is present in the set of compatible next tools (of the last tool) of a tool sequence. The accuracy of the compatible top-1 and top-2 are higher than that of the absolute top-1 and top-2 which proves that the network learns features (tool connections) from different tool sequences and use these features in prediction. The top-1 achieves higher accuracy than the top-2 for both the metrics. There is a severe drop in performance of the absolute top-2 for the training and test paths because many of the tool sequences have just one next tool in the workflow. The compatible top-1 and top-2 accuracy remains high ($\geq 90\%$) for the training and test paths.

11.3.9. Length of tool sequences

The variation of the precision (absolute and compatible) is verified with the length of the paths for training and test sets. The values of precision of the tool sequences with the same number of tools are averaged. This is repeated for all the tool sequences of different lengths. The tool sequences of the training and test sets are used to compute this variation. Figure 46 shows this variation.

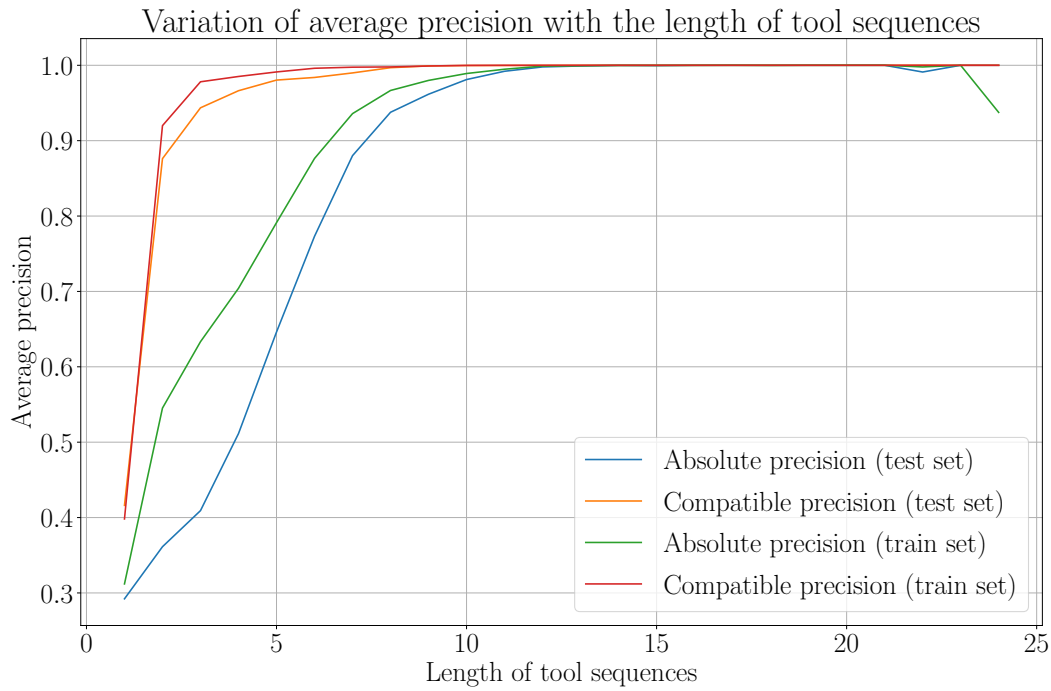

Figure 46.: Variation of precision with the length of tool sequences: The plot shows how the length of tool sequences affects the precision. Both the precision, absolute and compatible, are considered. The plot shows that as the length of the tool sequences increases, the precision becomes better. This improvement is more dominant for the compatible precision compared to the absolute precision.

As the maximum length of a tool's sequence is fixed to 25, the plot shows the maximum length as 24. The last tool is used as the next tool. It is concluded from the plot (figure 46) that as the length of the tool sequences increases, the precision also increases. This increase is more dominant for the compatible precision compared to the absolute precision for the training and test paths. As the length of the tool sequences increase, they have more tool connections and thereby have more features. The higher number of features present in the longer tool sequences helps in predicting the next tools more robustly. This is achieved even though the number of tool sequences with lengths ≥ 20 is lower compared to the number of tool sequences with shorter lengths.

11.3.10. Neural network with hidden dense layers

A network with only dense layers is also used as a classifier to compare the performance of the dense and recurrent layers on the workflow paths. Two hidden layers are used with 128 neurons each. The first layer is the embedding layer and the last layer is also a dense layer. The dropout (0.05) is applied to combat the overfitting. Rest all the parameters remain the same as for the recurrent neural network. The paths for the training and test sets are decomposed in the same way as in the section 11.2.3. The network is trained for 40 epochs. The absolute and compatible precision are computed after each training epoch. The training and validation losses are also noted. Figure 47 shows the performance of this network.

Precision and loss decomposing train and test paths (neural network with dense layers)

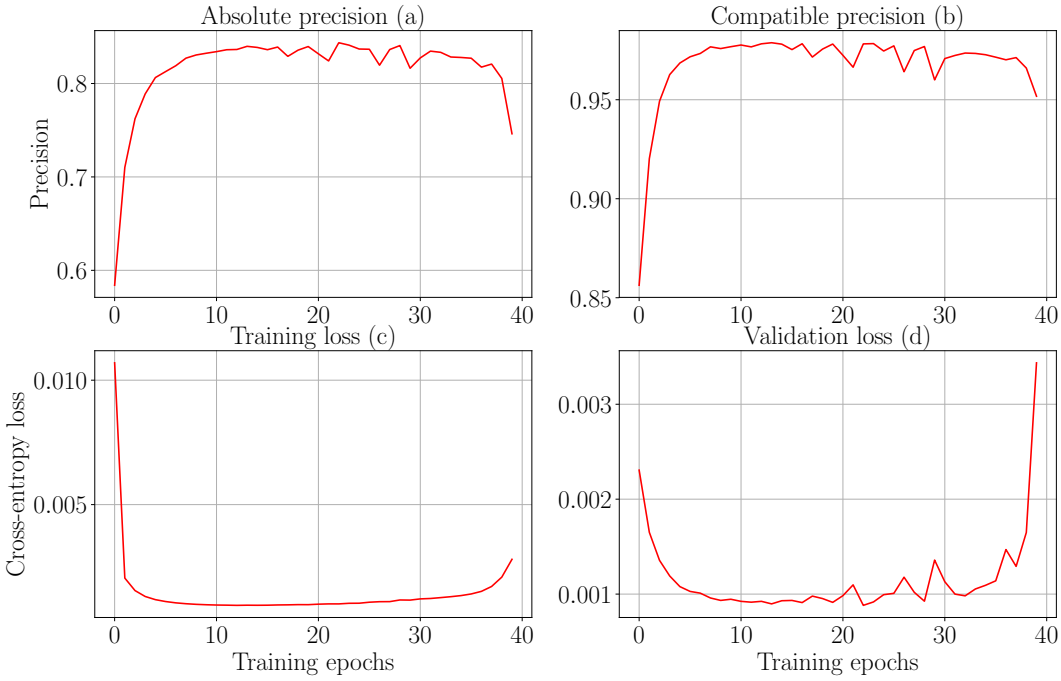

Figure 47.: Performance of a neural network with hidden dense layers: The plot shows the performance of the neural network with the hidden dense layers. The paths are decomposed keeping the first tool fixed (as described in the section 11.2.3). The subplots 47a and 47b show the absolute and compatible precision respectively. The subplots 47c and 47d show the training and validation losses respectively.

Figure 47a and 47b suggest that this network with the hidden dense layers also starts performing well (earlier in the epochs). It reaches the absolute precision of

$\approx 83\%$ and compatible precision of $\approx 97\%$ around the 15^{th} epoch. The learning seems to be good in the beginning but it starts becoming worse towards the end of the training. The precision stops increasing and starts becoming worse. Moreover, the training and validation losses (figure 47c and 47d) start increasing. They collectively suggest that the network is overfitting. The neural network with hidden recurrent layers performs better than this neural network with the hidden dense layers. It is concluded that the neural networks with the recurrent layers are more suited for learning on the sequential data than the neural networks with only the dense layers.

12. Conclusion

The aim of the work was to predict next tools for a tool or a sequence of tools. The workflows were first divided into paths and these paths were treated as sequential data. The paths in a workflow were considered independent. The recurrent neural network was used as a classifier to learn the tools connections in workflow paths. The workflow paths were decomposed into smaller tools sequences using three different approaches.

12.1. Network configuration

Many different configurations of the recurrent network were tried out to find which one achieves better precision without overfitting. Applying dropout was found to be beneficial to reduce overfitting. A larger number of memory units and a larger size of embedding layer and mini-batch increased the performance compared to their respective lower sizes. The optimisers with adaptive learning rates performed well compared to non-adaptive ones. A comparison was done to ascertain the best learning rate and the activations.

12.2. The amount of data

A large number of workflows played a significant factor to improve the absolute and compatible precision. It means that for any feature, the number of samples increased which led to higher classification performance. Moreover, a large amount of data enabled to use a more complex network with 512 memory units and 512 dimensional embedding layer. The classification rate increased with the more complex network. But, with the increased data and more complex network, the running time of the algorithm also increased. It not only increased the training time but also the evaluation time of the test set.

12.3. Decomposition of paths

for the idea of decomposing only the test paths, the classifier was trained on long tools sequences. It did not perform well as the classifier failed to learn the semantics of smaller tools sequences (present in the test data). But, for other ideas where the train and test sets were decomposed identically, the classifier performed well achieving $\approx 90\%$ precision.

12.4. Classification

There were multiple tools in the set of next tools for all the tool sequences. Moreover, there were multiple tools as next tools for a tool sequence. Therefore, this it is multilabel and multiclass classification. The compatible precision was higher than the absolute precision for all the approaches of path decomposition. Features learned from the paths in the training set were used to predict the paths in the test set. Only the knowledge of next tools present in the training set was used for prediction. Therefore, sometimes the predicted next tools did not match the actual next tools in the test set.

13. Future work

A number of improvements which can be made a part of this analysis are as follows.

13.1. Use convolution

The gated recurrent units achieved a precision of $\approx 90\%$. Using convolutional layers along with the recurrent layers, the classification performance can be improved. The convolutional layers can be stacked above the recurrent layers to learn sub-features from the smaller parts of the tool sequences. Convolution is well-suited to learn the features irrespective of their positions.

13.2. Train on long paths and test on smaller

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

13.3. Restore original distribution

While taking unique paths into training and test sets, the original distribution of paths was not taken into consideration. After dividing data into the training and test sets, the original distribution should be restored for the training set only. Then, a classifier assumes that the path repeating multiple times in the training set is important.

13.4. Use other classifiers

The recurrent neural network was used as a classifier for this approach. Bayesian networks and markov fields can also be used as the classifiers to predict next tools. They can provide a different insight. Multiple configurations of deep neural network with dense layers can also be tried out.

13.5. Decay prediction based on time

The tools which are deprecated and are not used anymore in Galaxy should be excluded from the analysis. To achieve that, the following two ideas can be used:

- Exclude the tools which are not used for a certain amount of time while extracting the workflows.
- Keep last used information for each tool. Using this, remove those tools which are not being used anymore from the predictions.

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 Merrienboer, 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.

A. Appendix

A.1. Visualisers

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

- Use full-rank document-token matrices[1].
- Use 5% of full-rank document-token matrices[2].
- Paragraph vectors[3]

A.2. Code repositories

The following sections provide the location 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 the different approaches (*latent semantic analysis* and *paragraph vectors*). For *latent semantic analysis* approach, there are two branches:

- Full-rank document-token matrices[4].

[1] `https://rawgit.com/anuprulez/similar_galaxy_tools/lsi/viz/similarity_viz.html`
[2] `https://rawgit.com/anuprulez/similar_galaxy_tools/lsi_005/viz/similarity_viz.html`
[3] `https://rawgit.com/anuprulez/similar_galaxy_tools/doc2vec/viz/similarity_viz.html`
[4] `https://github.com/anuprulez/similar_galaxy_tools/tree/lsi`

- Document-token matrices reduced to 5% of the full-rank[5].

Both these branches differ only in their ranks of corresponding document-token matrices. There is a separate branch for *paragraph vectors* approach[6].

A.2.2. Predict next tools in scientific workflows

The Galaxy workflows are represented as the directed acyclic graphs and they can be visualised in a website[7]. The workflow chosen from the dropdown is displayed as a cytoscape[8] graph. The separate code repositories are maintained for the ideas discussed in section 9.5.1. They are listed as follows:

- No decomposition of paths[9]
- Decomposition of only test set[10]
- Decomposition of test and train sets[11]

[5] https://github.com/anuprulez/similar_galaxy_tools/tree/lsi_005
[6] https://github.com/anuprulez/similar_galaxy_tools/tree/doc2vec
[7] https://rawgit.com/anuprulez/similar_galaxy_workflow/master/viz/index.html
[8] http://js.cytoscape.org/
[9] https://github.com/anuprulez/similar_galaxy_workflow/tree/train_longer_paths
[10] https://github.com/anuprulez/similar_galaxy_workflow/tree/train_long_test_decomposed
[11] https://github.com/anuprulez/similar_galaxy_workflow/tree/extreme_paths