Searching and Recommending Neural Networks

Adrian Pucher
ucvzt@student.kit.edu

Chiara Pullem
ulefw@student.kit.edu

Lucia Schuler
ujejm@student.kit.edu

Sven Koepke
uhefy@student.kit.edu

Theresa Bluemlein
udehi@student.kit.edu

1. MOTIVATION

Data analysis problems come in many different formats and occur in just as many areas of application. In recent times, the approach of artificial neural networks *(ANN)* received increased attention due to massive technological and scientific progress in this field. However, researchers and companies often face the challenge of choosing the implementation best suited to treat a data set of interest. The code hosting platform GitHub hosts a plethora of public code repositories implementing often pre-trained neural networks for re-use and application by other researchers and practitioners. However, finding the right repository and browsing through repositories which offer similar solutions for the problem at hand is made difficult by the lack of a specialized search and recommendation feature. Our research tackles this shortcoming by designing, implementing and evaluating a customized search engine and recommender system solution tailored to finding and discovering neural networks. On the basis of the rudimentary search engine *FAIRnets*, we develop a more sophisticated approach that makes use of more exhaustive repository information, intelligently extracted from GitHub. In addition to the search, we enhance *FAIRnets*' functionalities by proposing a content-based recommender system architecture which suggests relevant repositories based on readme and neural network architecture similarity. The empirical evaluation of our approaches demonstrate their effectiveness and the recommender proves superior compared to a baseline approach. The combination of both approaches is unprecedented in the application to neural networks and thus fills a gap in information retrieval literature.

The remainder of this paper is organized as follows. In Section 2, we present related literature from the domain of information retrieval research. Section 3 then examines the methodology along with the underlying concepts of our proposed architecture. In Section 4, we elaborate on the practical implementation as well as the used frame work for the web service. Section 5 documents the results of our empirical evaluation and discusses the obtained results. Finally, Section 6 concludes and points to avenues for future research.

2. RELATED WORK

Recent developments in information retrieval often refer to more individually tailored suggestions to the user of search and recommendation engines. However, the combination of both functions has not been attempted yet in extant academic literature. The subsequent paragraphs present some related approaches and bring our undertaking in relation to existing searching and recommending systems.

2.1 Search Engine

With the rise of the world wide web, the potential to quickly access information has increasingly been tapped. Due to the progressively growing amount of available data, a simple key word search in real time often does not account for the comprehensive intent behind the search query. We therefore attempt to establish an intelligent search that integrates contextual information to the input search terms.

Similar to [15], our idea is to separate the real time search from the long term available knowledge base. We therefore differentiate between the on-line search procedure and the previously, off-line collected contextual information in the titles, descriptions, tags and readme files of all search items contained in the knowledge base. However, in contrast to the central goal of the referenced search engine, which is the suggestion of case instructions for technological problems such as printer configuration or network installation, we intend to provide the user with pre-trained neural networks, available in the form of public GitHub repositories.

Aside from the search engineering, the ranking of the results involves a large degree of subjective interpretation. In order to maintain flexibility, we consider both relevance and popularity in the composition of our search item list. [8] create a set of different possible rankings and generate query-specific importance scores for pages. Our approach is similar in the sense that we establish a tag-sensitive search. We manage to parallelize our search according to the search terms and can easily alter the weights in the ranking score, dependant on the emphasis of the search engineer.

2.2 Recommender System

Despite their fast-growing popularity in recent years, a review of academic literature does not reveal any attempts to design a recommender system specifically for suggesting relevant neural networks. We intend to close this gap by extending previous, more general approaches to open-source software project recommenders.

Several more general approaches to creating recommender system architectures for open-source software repositories exist in academic literature. Closest to our approach is the work done in [26], where the authors design a heuristic-based recommender system for GitHub repositories which aims at finding similar repositories based on a query repository. In addition to other meta data, the authors exploit the rich contextual information contained in the readme files of

Seminar Knowledge Discovery and Data Science, SS 2019

Institute of Applied Informatics and Formal Description Methods (AIFB)

Karlsruhe Institute of Technology (KIT)

Supervisors: Prof. Dr. York Sure-Vetter, Anna Nguyen

Date of Submission: 31 August 2019

the project repositories. Their pre-processing employs the standard *tf-idf* weighting scheme for converting raw text data into vector space representations and relevance is determined by calculating the cosine similarity as a measure of similarity between two readme files. We draw on their approach to use the available textual information to derive similarities between repositories. We extend their work by utilizing the more elaborate text document embedding approach known as *doc2vec* [12][4].

[21] uses an unsupervised learning approach (Latent Dirichlet Allocation) based on readme files from GitHub repositories to automatically catalogue them into useful categories. Their approach, however, does not harness the multitude of available user-assigned labels and thus requires manual labeling and interpretation of the extracted categories. We therefore extend their approach by combining an unsupervised document embedding method with supervised learning methods, enabling us to train classifiers which can accurately predict a category of interest.

The combination of automatic labeling based on user-created project descriptions with content-based similarity detection for the purpose of recommending relevant neural networks, to the best of our knowledge, represents a gap in existing literature. Moreover, as far as we are aware, there is no existing recommendation approach which utilizes the architectural information of *ANN* implementations.

3. METHODOLOGY

The fundamental difference between the search and recommendation engine approaches lies in the nature of the query object triggering the search and the way in which the results are presented to the user. For this reason, the methodologies behind the design of these two search components differ greatly. While the search engine receives key words as an input and produces a ranked link list of relevant GitHub repositories as its output, the recommendation engine generates recommendation lists based on a query repository, which itself is the result of a prior keyword-based search. Furthermore, the output of the proposed recommender architecture is twofold, as two separate recommendation lists are presented, one based on semantic similarity between project readme texts and the other based on the similarity of the implemented neural network architecture. Another important aspect of our overall approach is the extraction of a suitable data set to base our search and recommendation designs on. Consequently, this chapter presents the conceived data collection process and the separate approaches to the conceptual design of the search and recommendation functionalities.

3.1 Data Collection

As the data source, we use the popular source code hosting and collaboration platform GitHub with a reported number of over 27 million public repositories, which contains two million projects coded in the programming language Python as the primary project language. We are interested in collecting a data set containing implementations of neural network architectures using the open-source neural network library Keras [2]. Keras is a high-level API written in Python which enables developers to design neural networks and save the models' architectures alongside their trained weights in the universal HDF5 file format. We exploit the fact that many publicly available neural network implementations on GitHub supply such a file, which facilitates the reconstruction, analysis and utilization of the models' architectures. In case such a file is not supplied, we employ code search to extract the relevant code passages from the available source code files in order to reconstruct the implemented neural network architecture.

3.1.1 Collection Process

The *GitHub REST API* [6] provides a convenient interface to perform filtered searches on *GitHub's* public-facing database. Our search query consists of all public *GitHub* repositories coded in *Python* containing the word "keras" either in their title, short description or readme. We limit the time frame of repository creation to span only the time after the official *Keras* release date on 03/27/2015 up until we started our data collection run (06/09/2019). The search yielded a total number of 28,269 repositories which matched the specified search criteria. For each of the found repositories, we extracted an array of relevant metadata for further processing and use in the final search and recommendation engine. We preprocessed the extracted repository information in real-time, identifying the readme language, converting the readme file from markup to plain text and extracting the network architecture from *HDF5* and *Python* source files into a nested *JSON* format. In order to achieve real-time processing of the collected information and complete the process in a reasonable amount of time, we leveraged parallel computing and remote server computing infrastructure. The extracted and preprocessed data was subsequently stored in a *NoSQL*, document-oriented cloud database hosted on *MongoDB*.

3.1.2 Data Quality and Statistics

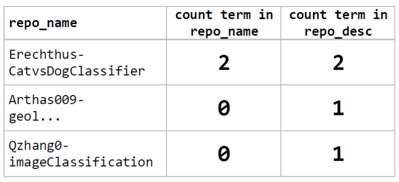
Having established a data base of 28,269 repositories, we determine the features that add a value to our purpose of creating a content-based search and recommendation engine. These features are either prevalent in a text or quantitative format and contain meta information about the owner or the repository itself. Since we are interested in the architecture and application of the neural networks, we regard every text based information as relevant, i.e. repo-name, repo-tags, readme-text and repo-description. From the quantitative figures, we use repo-forks and repo-watch to measure the popularity of the respective repository. This is particularly important for the search engine. We further consider the language, the owner and whether the architecture is made available in HDF5 format.
The data exploration process revealed the collected data to be partially incomplete and imbalanced. In a first step, we restricted our data set to repositories with English language readme files to facilitate the use of natural language processing methods. In principle, however, an extension to languages other than English is possible. With regard to the text-based data, we identify the challenge of dealing with varying numbers of assigned tags readme files of varying length and quality: Only 16.5% possess at least one user-assigned tag and the length of the readme texts varies between zero and approximately 80,000 words. Finally, we found the collected quantitative data to be mostly complete and thus easy to process. For those repositories which do not provide a HDF5 file, we extract information about the neural network architecture directly from the Python source code files in which the model is built. This two-pronged approach allows us to collect a solid data basis of approximately 15,600 repositories with complete architectural information. Our approaches of applying knowledge discovery and natural language processing (NLP) techniques to cope with the remaining challenges posed by the collected data are subjects to the subsequent sections.

3.2 Search

Traditional key word search algorithms can give poor results on search queries in our data base, with its large scale and enormously variable content quality. However, in our more sophisticated approach, we enrich the information contained in the key words provided by the user. We assume that search results can be improved by including the information contained in the name, the description, in user-assigned tags, and readme file of the repository. By introducing a *search matrix*, a *frequency matrix* and an *activity score*, we comprehensively respect the contextual information of the search terms.

3.2.1 Search Engineering

Since we were provided with a first version of *FAIRnets* [14] initially, the principal idea of our search engine has already been defined. The user can search for repositories by one or multiple terms. As soon as the engine receives the query, the words are pre-processed and augmented by semantically similar terms from the Google News Word2Vec data set [7] (more detailed information in paragraph 4.1). For each word of the entire search term, we iteratively count the frequency of occurrence in the name (`repo_name`) and description (`repo_desc`) of available repositories. This information is then saved in a *search matrix* that is set up in runtime for each query individually. Figure 1 shows such a search matrix with 3 sample repositories and a search query consisting of the search terms `'cat'` and `'classification'`:



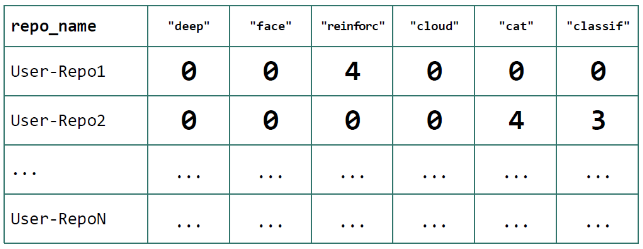
repo_name	count term in repo_name	count term in repo_desc
Erechthus-CatvsDogClassifier	2	2
Arthas009-geol...	0	1
Qzhang0-imageClassification	0	1

Figure 1: Search matrix with frequencies of the search terms in name and description of the repository

3.2.2 Frequency Matrix

As indicated in paragraph 2.1, we draw a clear line between the steps processed at runtime and the information from the data base collected and prepared a-priori and independently from the search terms. In order to expand our contextual information about the repositories, we do not only respect the name and description but base our intelligent information retrieval inter alia as well on the tags and readme of the corresponding repositories. From the data exploration, we learnt that not every repository has been assigned tags, but that some repositories were given up to 27 tags. Regarding tags as straightforward to process with NLP techniques, we want to project the information from the union of all available tags to all available repositories. Therefore, we save the tags in a list, stemmed the single words and removed duplicates. This list acts as the header of our *frequency matrix*, so that we have a column for each tag occurring. As far as the lines of the matrix are concerned, every repository of our test data set represents a single line in the matrix. Note that we identify repositories in the format of `User-Repo`*n*:

Conscious about the structure of the matrix, the understanding of the content is similarly straightforward. As mentioned before, it projects the information from the list of all available tags to every repository. Our approach is to count the occurrence of a tag of interest (column) in the readme file of every repository of the test data set (line). By that, we manage to efficiently explore the contextual information saved in the readme, since we only process a one-dimensional



repo_name	"deep"	"face"	"reinforc"	"cloud"	"cat"	"classif"
User-Repo1	0	0	4	0	0	0
User-Repo2	0	0	0	0	4	3
...	...	...	...	...	...	...
User-RepoN	...	...	...	...	...	...

Figure 2: Frequency matrix with frequencies of the search terms in the readme files of the repositories

list, the header with the tags. The entire procedure regarding the frequency matrix is conducted off-line once in advance and is computationally intensive.

3.2.3 Activity Score

Having integrated information from the name, description, tags and readmes of all repositories so far, we take an additional perspective on the popularity of the suggested repositories and the extent of their information content. We base our approach on [1] and [25]. While the latter is measured by the length of the readme file, we adduce the number of repositories per user, the number of watches and the number of forks as quantitative figures of the respective repositories. After we normalize these figures, we build a weighted sum and introduce it as an *activity score*.

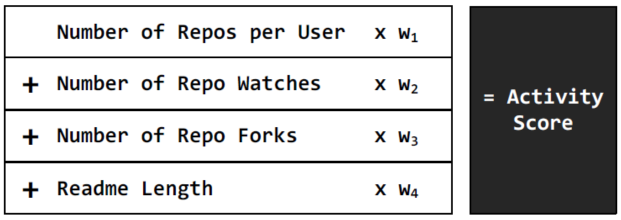

Number of Repos per User x w_1
+ Number of Repo Watches x w_2
+ Number of Repo Forks x w_3
+ Readme Length x w_4
= Activity Score

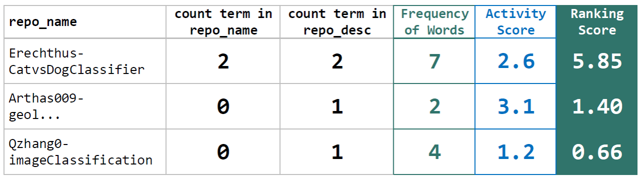
Figure 3: Composition of the *activity score* as weighted sum of popularity measures

Initially, we set the weights as $\mathbf{w} = (0.2, 0.35, 0.35, 0.1)$, but emphasize that this is subject to change depending on the preference of the search engineer. The activity score is calculated off-line and added during runtime after relevant repositories are selected and saved in the search matrix.

3.2.4 Generating a Ranking

Turning back to the search matrix generated for the sample search query, we can further analyze its entire procedure: Every term of the incoming search query and its augmenting neighbor terms are matched with the tags of the *frequency matrix*. Next, the relevant repositories are returned in descending order of term frequency as additional column of the *search matrix*. Subsequently, the activity score of every entry of the search matrix is saved in another attached column of our matrix. Finally and with another weighted summation, we calculate the ranking score and order the results again in descending order. Figure 4 summarizes the composition and ranking of a list of relevant search items.

Designing our search engineering process as described above, results in several extensions to a regular key word search. We manage to iteratively count for every term in the query instead of only the query as concatenated string. Further, we resort both to existent NLP techniques in pre-processing and generate own approaches (i.e. frequency matrix and activity score) in order to exhaust the contextual information



repo_name	count term in repo_name	count term in repo_desc	Frequency of Words	Activity Score	Ranking Score
Erechthus-CatvsDogClassifier	2	2	7	2.6	5.85
Arthas009-geol...	0	1	2	3.1	1.40
Qzhang0-imageClassification	0	1	4	1.2	0.66

Figure 4: Search matrix, complemented by contextual information from the frequency matrix and popularity information from the activity score

of the repositories as comprehensively as possible. The adjustment of the hyperparameters such as the weights in the activity score or ranking score formulae provides both the burden and opportunity to tailor the ranking procedure to individual preferences. The dilemma of emphasizing quantitative versus textual information is hence configurable.

3.3 Recommender System

Our recommendation approach comprises the design of a content-based recommender system considering both semantic and architectural similarity between neural network repositories. In the first step, we extract useful information about the neural network's *intended use* from user-assigned labels and information about the network's *architecture* from provided source files. For those instances in which either information is not available, we use the given textual description contained in the readme file to infer the missing label information about the intended use and the overall network architecture. We then convert the textual information contained in the readme files and the information about the architecture of their neural network implementation into vector space representations using two different embedding approaches. Based on the extracted vector representations, we calculate similarity matrices for both information types between all repositories in our sample data set. Finally, we use the similarity information in combination with content- and architecture-related filters to generate top N recommendation lists.

3.3.1 Attribute Extraction

Label information indicating which broader category a neural network belongs to can serve as a means for filtering recommendations down to the relevant subset of available neural network repositories. In particular, for the purpose of recommending contextually and architecturally relevant projects with respect to a query repository, labels with information regarding a neural network's intended use and its high-level architecture provide a source of knowledge which can be leveraged for more relevant recommendations. We hypothesize that the quality of recommendations resulting from comparisons of a readme file's or network architecture's vector representations can be augmented by using such high-level filters.

GitHub allows its users to label their repositories with relevant topic tags, making them more easily searchable by using category or keyword filters. In our collected data set, however, only 15.9 % of repositories with an English readme file feature user-assigned categories. We analyzed the category assignments among the labeled repositories and manually matched corresponding sub-categories to the more general categories "*Computer Vision*", "*Natural Language Processing*" and "*Numerical Prediction*". 30.2 % of all repositories with assigned topic labels were thus matched to either of the three artificial parent categories. With respect to the whole data set, the fraction of filtered repositories whose labels could be matched to either of the aforementioned categories only accounts for 4.5 % of all repositories, equaling a total number of 1,266.

In order to obtain high-level information about the neural network's architecture, we harness the architectural information previously extracted from available h5 files or python source code. This approach allows for a significantly larger sample size compared to the *intended use* case as 42.2 % of all repositories in our sample provide information with respect to the neural network architecture. The *network type* attribute is assigned the possible values *feed forward*, *recurrent* and *convolutional* based on a an automated analysis of the network's individual layers with respect to their affiliation to the layer classes provided by the *Keras* framework. In case any of the network's layers are identified as a *convolutional* or *recurrent* layer, the respective label is assigned. In case neither of these two classes are present, the network is assigned the *feed forward* type label. Note that while *feed forward* and either *convolutional* or *recurrent* are mutually exclusive class affiliations, a network can be assigned both the *convolutional* and the *recurrent* class labels.

3.3.2 Missing Attribute Inference

As labels pertaining to the intended use and architectural information are only provided for a small subset of the whole data set, a recommendation filter based on the extracted information is not suited for a generating comprehensive recommendations. In order to mitigate this limitation in data availability, supervised machine learning methods trained on the available labeled subset of the data can be employed to fill in the missing information with high-quality predictions. The descriptive readme files provide a yet untapped source of unstructured information which can be used to that end. We thus leverage the fact that most repositories in our data set have a curated English readme file to fill in missing information about the network's *intended use* and *architectural type* by training classifiers with on the artificially created labels and generated vector representations of the text files.

3.3.3 Readme File Embedding

In order to obtain a similarity matrix for all repositories, the textual information contained in the readme files needs to be preprocessed and subsequently transformed into a numerical vector representation. We follow a common text preprocessing pipeline consisting of removing numbers, special characters and stop words as well as lemmatization of words with the goal to reduce inflectional forms.

There are several approaches for obtaining fixed-length vector representations of text documents. The simplest and most established are so-called bag-of-word approaches which rely on simple word counts and various derivatives. One of these derivatives is the *tf-idf* weighting scheme which weights relatively infrequent words higher than commonly used words. Bag-of-words approaches generally suffer from sparsity in its resulting vectors and the inability to capture the semantics of long text documents. More recently, distributed representations in the form of document embeddings have been proposed as a remedy for these shortcomings. In [4] the authors demonstrate how such embeddings have the ability to outperform classical approaches and even more sophisticated approaches such as Latent Dirichlet Allocation (LDA) in the task of determining the semantic similarity (relatedness) of documents.

The concept of document embeddings is closely related to word embeddings and was first proposed as an algorithm termed *Paragraph Vector* [12]. The goal of this approach is to obtain dense fixed-length vector representations of whole documents of variable length. This is achieved by training document representations alongside word representations of words sampled from that document. While word representations are shared across documents, the document representation is only shared in the context of sliding window samples from that same document. After the training of the embedding, the resulting vector representations can be used as inputs for supervised machine learning algorithms or for unsupervised tasks such as clustering or similarity calculations. The main advantages over bag-of-words models lie in the preservation of semantics and the dense representation of documents of varying sizes.

We use the concept of document embeddings to create semantics-preserving vector representations of the repositories' readme texts. Following the best-practice recommendation of the original authors, we employ the vector concatenation version of the distributed memory (PV-DM).

3.3.4 Neural Network Architecture Embedding

On a meta level, artificial neural networks can be described as a sequence of various layers with respect to a propagation and loss function, whereas the type of these components and their sequence mainly define the architectural design. Therefore, the determination of similarity between given architectures needs to capture complex semantic relationships. Evidently, the use of (semi-) supervised approaches seems not to be practical to process the huge database of an ANN search engine. However, the work of [18] introduces the so-called *RDF2Vec* concept which enables unsupervised feature extraction from *RDF* (Resource Description Framework) graphs with the help of language modeling approaches. This approach is structured by three major steps: data conversion, information extraction and vector representation.

First, we convert the collected database in the meta data model *RDF* which has its roots in the Semantic Web development [3]. This results in *subject-predicate-object* triples which describe the relationship between two resources e.g. (*Convolutional Neural Network - is subclass of - Neural Network*). In turn, multiple triples of this kind represent a labeled directed *RDF* graph which can cover the underlying semantic relationship of the resources. More formally, an *RDF* graph represents a graph $G = (V, E)$ consisting of a set of vertices V and a set of directed edges E [19]. The complete ontology used in our work can be found online [14].

In the next step, we use random graph walks to extract the generated semantic information as recommended by [18]. Generally speaking, for a given graph $G = (V, E)$, we generate n random walks $P_v n$ of depth d from each entity in the graph. The randomness of this procedure is given by the iterative random selection of the following edges and nodes until d iterations are reached. In our implementation we perform ten random walks with four succeeding vertices for each start node which results in a string corpus of entity sequences.

Subsequently, we use the resulting text corpus to train a word embedding which enables us to represent each *ANN* architecture as a numerical vector in the embedding vector space. In that vector space, the relative position of all entities to each other describes their semantic relationship. In the context of this work, this represents the desired architectural similarity.

3.3.5 Similarity Calculation

There are several methods for measuring the similarity between entities based on their vector space representations. For the purpose of this research, we apply such a similarity measure to both text document (i.e., readme) and neural network architecture embeddings. We opt for the commonly used *cosine similarity* measure, which quantifies vector similarity by determining the cosine of the angle between between two vectors. This is practically achieved by means of calculating the standardized dot product of the two vectors. The similarity of two entities based on their vector representations can thus be calculated using the following equation, where w_1 and w_2 denote the vectors of the two compared documents:

$$sim(w_1, w_2) = cos(w_1, w_2) = \frac{w_1 \cdot w_2}{\|w_1\| \cdot \|w_2\|} \qquad (1)$$

Calculating the similarities between all pairs of entity representations according to the above formula results in a symmetric similarity matrix as the cosine similarity is itself symmetric. Given a collection of n entities, calculating all possible combinations requires a total of $\frac{n^2-n}{2}$ calculations, as the matrix diagonal is a 1-vector of length n. As a result, the complexity of the task grows in a quadratic manner, requiring a distributed calculation to ensure an acceptable calculation time for large entity collections. As each similarity calculation can be conducted independently from each other, multi-core machines can be leveraged to provide a time-effective calculation.

3.3.6 Generating Recommendations

The calculated similarity matrices for both readme content and architecture similarity are used a basis for generating purely content-based recommendations for the user. As we do not assume any prior knowledge about the user's preferences or precise search goal, we aim to present the most relevant (i.e., similar) neural networks in terms of readme content similarity and architecture similarity based on the currently visited neural network repository. The current state of the search, meaning the currently visited details page of a neural network, thus serves as the query for generating and presenting top N recommendation lists, separately for each search dimension (i.e., semantic and architectural similarity). The extracted and predicted attributes *intended use* and *architecture* serve as optional filters by which to restrict the search space to a subset with presumed higher a-priori relevance. The top N lists are created by sorting the filtered or unfiltered row vectors of the respective similarity matrix pertaining to the query repository in descending order.

4. IMPLEMENTATION

After much conceptional description of underlying ideas in the preceding paragraphs, we focus on the implementation and execution of the pronounced tasks in the sequel, both for the search engine and for the recommender.

4.1 Search Engine

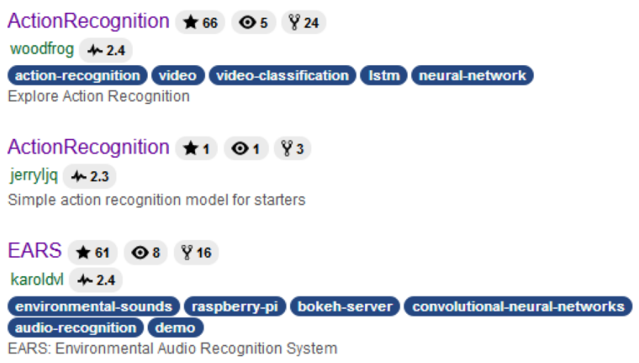
Separating the operations to be fulfilled in runtime from more comprehensive tasks that can be achieved a priori allows us to accelerate the searching process. Nevertheless, natural language processing techniques are applied both in advance and at presence of a search query and are hence described first. Subsequently, we present how a query is processed from handing over the search terms until the results are displayed as list. A review of achievements and challenges complements the paragraph about the implementation of our search engine.

4.1.1 Natural Language Processing

With the framework provided by GitHub to create and maintain a repository, users enter textual information in different extent and quality. To tackle the challenge of information extraction in Python, we use NLP packages such as *nltk* or *gensim*. By their means, we manage to pre-process natural language well. Considering extensive readme files, we start to tokenize them and save the single words in lists. Subsequently, we apply a stemming method, before we remove duplicates automatically and stopwords manually. Besides from readme files, we treat descriptions and tags analogously. Finally, we augment these textual information by semantically similar words form the Google News data set [7]. With the *word2vec* approach, we integrate even more context and save it in the respective lists of words.

4.1.2 Query Processing

The detailed description of our conceptional methodology to set up a search engine in paragraph 3.2 is implemented in several classes with appropriate methods. Processing a query involves the application of NLP techniques as well as interface operations. First, the initial search terms are passed to pre-processing functions as described in the previous paragraph. As augmented and yet precise lists of stemmed words, we dispose of our search term vector. We then generate the search matrix for our individual query and start to count the occurrence of words from the search vector in the names or descriptions of repositories from the data base. At this step, we implemented a function that removes duplicates in the search list. This first selection of potential search suggestions is handed over to the frequency matrix by their identifications, where a method returns the entries of the matrix per occurring tag. We match the entries of the search vector with the tags, so that the actual columns the method has to screen are dramatically reduced. Our method attaches the information from the frequency matrix as an additional column to the individually set up search matrix for our specific query. In the next step, we fetch the activity score for every repository and add it to the search matrix. On the basis of this finalized search matrix, we determine the ranking score per entry in order to prepare the ranking. Following this step, we hand over the repository identifier to our data base and query up-to-date information so that the interface can list the search items accordingly. This involves the link to the more detailed repository profile page, the owner, the description, assigned tags as well as the star, watch and fork figures.



ActionRecognition ★ 66 5 24
woodfrog 2.4
action-recognition video video-classification lstm neural-network
Explore Action Recognition

ActionRecognition ★ 1 1 3
jerryljq 2.3
Simple action recognition model for starters

EARS ★ 61 8 16
karoldvl 2.4
environmental-sounds raspberry-pi bokeh-server convolutional-neural-networks audio-recognition demo
EARS: Environmental Audio Recognition System

Figure 5: Sample ranking list of the first three results to the query `'recognition'`

4.1.3 Achievements and Challenges

On the basis of the initial version of FAIRnets as a starting point, we discovered potential to improvement in intelligent information retrieval, runtime, flexibility and plausibility of the ranking. First, we achieved an enhanced information augmentation by integrating the tags and readmes instead of only the name and description. Additionally, we enriched these words by their context by applying word2vec on the Google News data set. Second, in spite of computing the matrices as efficiently as possible, we still see a challenge to accelerate the search drastically. Third, the weighted sums acting as scores are subject of individual configuration according to the preferences of the constructor of the engine. Fine tuning the hyper parameters implies both an achievement and a challenge. Finally, the detailed presentation of the search results allow the user of the search engine to choose preferred results on an extended information basis.

4.2 Recommender System

The recommender system's practical implementation comprises two distinct areas of machine learning: the training and evaluation of the *supervised* classifiers used to infer missing attributes and the configuration and training of the *unsupervised* embeddings of the models' architectures and readme files. The following sections discuss the technical realization and parametrization of the employed approaches.

4.2.1 Classifier Training

We use the artificially labeled subset of the overall data to train and validate a multi-label classifiers for both *suggested use* and *architecture* prediction based on the available readme files. As an input for the *Intended Use* classifier, a *doc2vec* document embedding with a vector size of 30 is used to attain a vector space representation suited for classification tasks. We also use a simple *tf-idf* transformation for a benchmark model. Our model comparison and parameter tuning process resulted in the selection of a *support vector classifier* with a radial basis function kernel as the base classifier. Since multiple labels are possible, we used a *label powerset* approach to convert the problem into a multi-class problem. The validation using standard 10-fold cross-validation resulted in a weighted F_1 score of 0.915. Our benchmark *tf-idf* model (*label powerset* in conjunction with a *logistic regression* model) achieved a slightly lower weighted F_1 score of 0.895. We then trained the classifier on all available data and applied it to make predictions for all remaining 23,564 repositories with no application information but an English readme file.

We followed a similar process as described above for predicting missing information about the neural network type. The 10-fold cross-validated parameter tuning yielded a surprising result. For the task at hand, the benchmark model using a *tf-idf* transformation (*support vector classifier*) of the text data achieved a considerably higher F_1 score. While the embedding approach using the same configuration consisting of a *support vector classifier* in conjunction with a *label powerset* transformation achieved a 10-fold cross-validated weighted F_1 score of only 0.686, the benchmark model results in a weighted F_1 score of 0.720.

On a more general note, the overall achieved F_1 scores for predicting the neural network *architecture* are considerably lower than the scores achieved for predicting the *suggested use*. We hypothesize that the reason for this result is twofold: data quality and readme content. Since in case of multiple h5-files or model-building source files the ultimately utilized file was chosen at random, the readme file contents do no necessarily exclusively describe the model the architectural information was extracted from. In addition, the contents of the readme files tend to more reliably include descriptions of the network's use case than it does of the network's architecture.

4.2.2 *Embedding Training*

As training of embeddings is too computationally complex to efficiently parameter-tune and their quality hard to assess in an automated fashion, we rely on previous literature for our hyper-parameter selection. Our implementation is based on the *gensim* [17] libary.

For the training of the readme file embedding we used the distributed memory (PV-DM) version of *gensim doc2vec*. We further we chose an embedding size (vector size) of 100, as [4] have shown in their work that the embedding quality does not generally significantly improve with an increasing vector size. The context window size was set to 6 and the minimum word count to 5. We trained the embedding model for a total of 40 epochs. As a sanity check for the quality of the trained embedding, we examined the top 5 similar words for common domain words such as "neuron" and "network" and found the suggested close substitutes sufficiently adequate. We further performed a dimensionality reduction (principal component analysis) on the inferred document embeddings in order to visualize the automatically labeled documents in a 2-dimensional space and found the documents sharing the same label to exhibit discernible clustering behavior with distinct class boundaries (see fig. 6).

In the case of the ANN architecture embedding we used *word2vec* neural language model with the *Skip-Gram* algorithm as [19] received the best results in testing several recommender systems with this approach. We opted for a window size of 10 and a total of 30 training epochs to obtain a 20-dimensional word embedding.

For all other hyper-parameters, we relied on *gensim*'s default model settings.

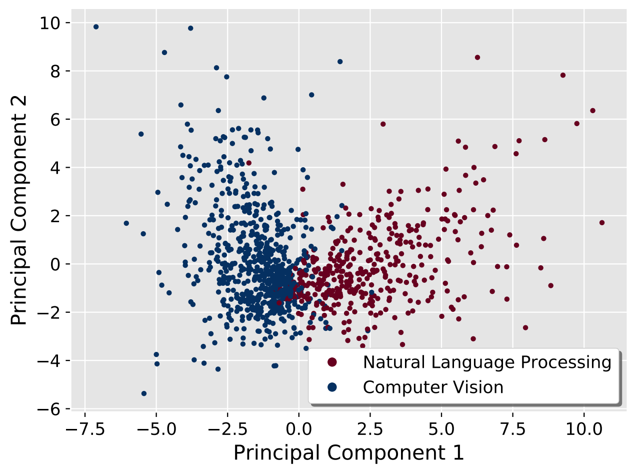


Figure 6: Visualization of the Semantic Distance of the *doc2vec* Embedding

4.3 Technical Search and Recommendation Implementation

4.3.1 *Back End*

The practical implementation of the search engine and recommender system is based on the FAIRnets [14] search engine project for neural networks developed by the Institute of Applied Informatics and Formal Description Methods (AIFB) at the Karlsruhe Institute of Technology. The service is built as a *Django* [5] web application using version 2.2.1 under *Python* 3.6. For loading and accessing the neural network ontology and RDF graph data, we use the XML-serialized *OWL* data format. For all other data retrieval and manipulation purposes, we rely on the Pandas *DataFrame* [13] format.

4.3.2 *Front End*

The user interaction with the search and recommendation engines is facilitated by the *FAIRnets* front end, which features a two-layered interface known from popular search engines. The *search layer* provides an input field for keyword-based user query and displays a paginated list of search results upon submission of the search query. Each individual results listing comprises of the repository and owner name, an extract from the textual description and relevance and popularity measures such as stars, subscription counts, number of forks, and activity score.

Following the hyperlinks of the search results directs the user to a *details layer*, where repository-based recommendations are displayed alongside structured, relevant information pertaining to the retrieved search result (i.e., GitHub repository). In addition to general project information such as the owner and project name, license and the number of stars, subscribers and forks, the details page further showcases information pertaining to characteristics of neural network implementations. This information comprises the intended or suggested (i.e., predicted) use, the employed optimizer and loss function, the general architecture (type of network, total number of layers and neurons) and more detailed architectural information specifying the number of neurons for each layer and the activation function used. The latest version of the supplied readme file is dynamically loaded from GitHub's servers so as to provide a detailed textual description of the software project without having to follow external links. The presentation of repository information thus differs greatly from the GitHub project pages in that it is highly customized to repositories featuring Keras implementations of neural networks. The two separate top N recommendation lists are displayed in a sidebar next to the query repository's details and contain additional information aimed at helping the user determine the recommendation item's relevance without having to follow the hyperlink to the respective details page. The default setting filters the readme-based recommendations based on the known or inferred use case while the architecture-based recommendations are filtered according to the query repository's extracted architecture. However, the user interface allows for deletion and re-activation of those filters by the user. Figure 7 outlines the schematic design of the system's implementation and basic user interaction.

5. EVALUATION

To test the quality and validity of our search and recommendation approaches, we conduct a quantitative evaluation by means of two small-scale user studies. We base the studies on a list of 25 sample search queries representing typical queries of interest to researchers and practitioners. Figure 8 shows the 25 sample queries used for assessing the quality of the generated search results lists. As the recommender system requires a single repository as a query for generating the recommendations, we use the respective top search result of each of the sample query terms as a starting point. Note that the evaluation of recommendation performance is solely based on the query repository and not the original search term in order to allow for a separate evaluation of the functions.

5.1 Test Data Set

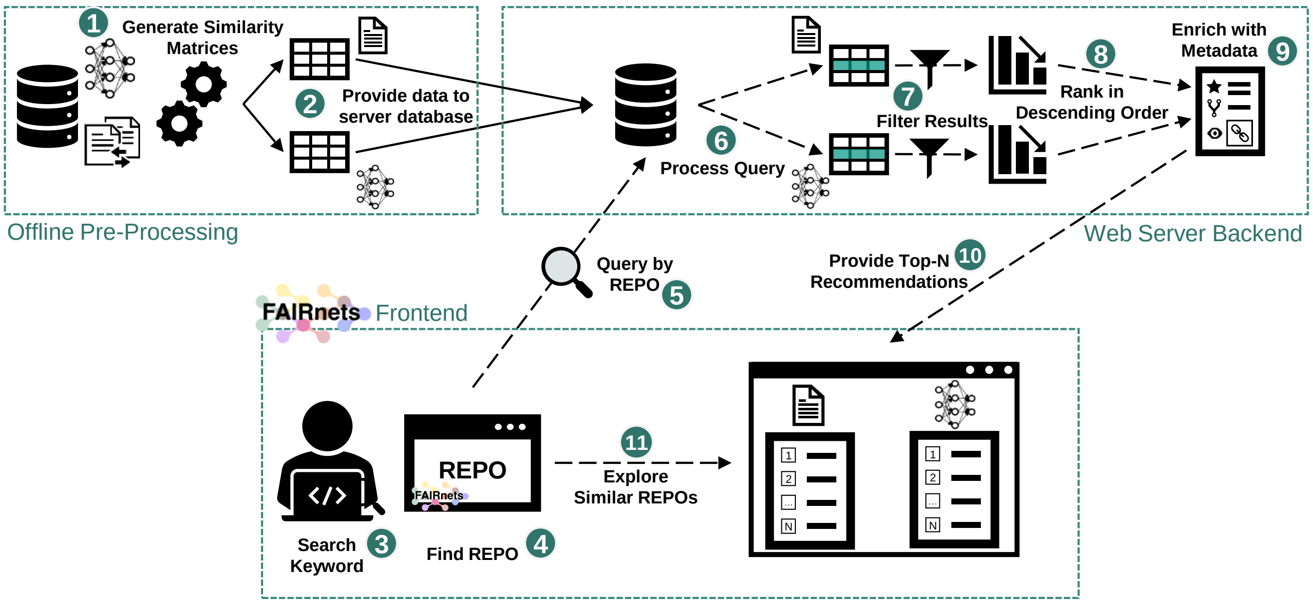


Figure 7: Schematic overview of recommender design

As the calculation of the required similarity matrices and the data pre-processing for the search engine pose significant computational challenges, we restrict our evaluation to a subset of the available data. The availability of the neural network architecture information and a minimum readme file length of 3,000 characters filter the data down to a manageable number of 3,332 repositories that we conduct our evaluation on. The resulting selection also ensures that recommendations based on the extracted architecture are possible and that the provided readme files contain a minimum amount of contextual information.

5.2 Search Engine

To begin with the evaluation of the search itself, we consider 25 sample queries (c.f. Figure 8). For each of the queries, we analyze the ranking of the top-10 results from our test data base. The results of the achieved first indication based on human judgment are discussed subsequently.

5.2.1 *Experimental Setup*

The setup then provides that two expert users independently re-rank the list of 10 results according to professional judgment of the utility of each of the result. To best fulfill this task, the users are allowed to ground their preference on the quality of the readme or the applicability of the respective repository to the original problem. A user can only assign the ranks 1 to 10 to the search items and must not award a rank twice. If the search engine returns less than 10 results, a user assigns the ranks in descending order, starting with rank 1 for the most suitable search item.

5.2.2 *Evaluation Metrics*

In a list of n search results, the Spearman rank correlation coefficient [22] indicates how well the proposed rank r_i coheres with the claimed actual rank $\hat{r}_i$:

$$\mathrm{r_s} = 1 - 6 \cdot \frac{\sum_{i=1}^{n}(\hat{r}_i - r_i)^2}{n^2 \cdot (n-1)} \tag{2}$$

We regard this measure as most suitable for an initial evaluation since it quantifies the correlation of ordinal data, which our ranking belongs to, since we can compare the search results relatively to each other.

5.2.3 *Results*

Recalling that the experimental setup proclaimed two expert users, we received two Spearman rank correlation coefficients for each of the sample search query i. While $r_{S,i}^{(1)}$ represents the correlation of User 1's judgment and the proposed ranking of our search engine regarding search term i, $r_{S,i}^{(2)}$ does so for the ranking of User 2. Note that we received the individual value for each of the search term i by building the average over the two coefficients:

$$r_{S,i} = \frac{1}{2}(r_{S,i}^{(1)} + r_{S,i}^{(2)}) \tag{3}$$

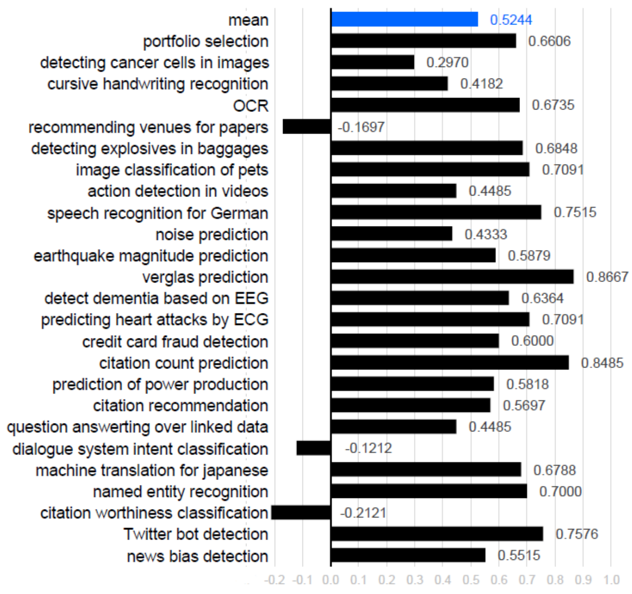


Figure 8: Spearman's rank coefficient for sample search queries

Figure 8 lists the rank correlation coefficients each of the 25 search terms, with a range from 0.2970 to 0.8667 in case of positive values and a mean correlation of 0.5244. However, for *recommending venues for papers* (−0.1697), *dialogue system intent classification* (−0.1212) and *citation worthiness* *classification* (−0.2121), we have negative values for the rank correlation between the suggested ranking and the professionally judged ranking. Search queries such as *verglas prediction* (0.8667), *citation count prediction* (0.8485), *Twitter bot detection* (0.7576) and *speech recognition for German* (0.7515) exhibit a distinct correlation ($r_{S,i} > 0.75$). For the remainder of the sample queries, we also achieve good correlation results that fluctuate around the mean correlation.

5.2.4 Discussion

In order to improve the performance of the search engine even more, it would be advantageous to extend the knowledge base. This would anticipate at least the challenge of the negatively correlated rankings. We attribute this phenomenon to the fact that we use a test data base with insufficient contextual information about the respective queries. We expect the search results to the respective topics to improve due to a broader knowledge base. In case of distinctively positive correlation values, we conclude that the estimation of the users coincides mutually and with the ranking suggested by the search engine. For the remaining search terms that yield good correlation values, a direct coherence between the suggested ranking and the professionally judged ranking is achieved.

Another challenge is to correctly deal with excessively used terms such as *classification* or *prediction.* We did not want to treat them as stopwords but would consider normalizing or degrading their frequency accordingly.

As far as the measure is concerned, a severe shortcoming lies in the fact that is does not include any perspective on the validity of the ranking. This implies that even in the case of only irrelevant search results, ranks 1 to 10 have been assigned to every result. Consequently, the validity might be inherently biased to the extent that the used evaluation method is prone to subjectivity. However, our present evaluation approach gives a first indication based on human judgment and can be improved in the form of long term evaluation studies by experts or analysis of the users' clicking behaviour as demonstrated in [11].

5.3 Recommender System

Since there is no comparable recommendation approach to test our recommender against on our sample data set, we conduct a purely internal evaluation against conceived benchmarks. For the purpose of measuring the performance on the two separate tasks of recommending *semantically* and *architecturally* similar repositories based on a query repository, we treat those tasks separately in our evaluation.

5.3.1 Experimental Setup

We base our evaluation on 25 sample query repositories obtained from the top query results from our previous search engine evaluation. In case of duplicates, we replaced them with the nearest search result that had not been selected yet. We further excluded such repositories that contain larger collections of different neural network implementations and replaced them in the same manner, since in those cases it is not clear how to evaluate the relatedness to the recommended networks. Two of the authors were randomly assigned to one of the testing scenarios. We thus obtained one set of ratings for each of the two tasks by two separate judges. Both judges have considerable python coding experience and are proficient in the python framework *Keras.* In the rating process, the two judges were given the names of all the query repositories which they iteratively looked up using the *FAIRnets* search engine interface. Upon opening the details page of the query repository, they were shown a top 7 recommendation list, as proposed by [9], with some high-level information on the corresponding repositories. For each of the 7 recommendations, they followed the hyperlink to the respective details page and studied the relevant information in order to reach a conclusion about the degree of relatedness to the query repository. Following a common scale frequently used in literature, we used a 5-item Likert scale ranging from 1 to 5 with 1 corresponding to "highly irrelevant" and 5 corresponding to "highly relevant" (see table 1). We deem a recommendation to be successful if the corresponding user-rating is four or higher.

Finally, this basic evaluation procedure was conducted for five different recommendation setups. For the readme-based recommendations, we compared a benchmark system with recommendations based on a simple *tf-idf* approach with those based on the proposed *doc2vec* and a *doc2vec* setup with an additional *intended/suggested use* filter to display only recommendations with the same known or inferred applications as the query repository. In the case of the neural network architecture embedding, we compared two different approaches: one with an unfiltered recommendation list and one with a set filter for listing only results with the same network type as the query network.

5.3.2 Evaluation Metrics

Following the approach used in other studies, we use four different evaluation metrics to measure the effectiveness of our approach. These metrics are:

Precision at k (P@k).

It describes the percentage of relevant documents among the *top-k* for a given query and its result list [20].

$$\text{P@k} = \frac{|\{\text{relevant documents}\} \cap \{\text{k retrieved documents}\}|}{|\{\text{k retrieved documents}\}|} \tag{4}$$

The mean of the *P@k* values over all queries for a given k is expressed as *mean precision at k (MP@k).*

Average Precision (AP).

The *AP* is defined by the mean of the *P@k* values for all *k* which describe the position of a relevant document in the result list. This can be described formally, with *rel(k)* as an indicator function which equals 1 if the item at rank is relevant and zero if not [24].

$$\text{AP} = \frac{\sum_{k=1}^{n}(P@k \times \text{rel}(k))}{|\{\text{relevant documents}\} \cap \{\text{k retrieved documents}\}|} \tag{5}$$

Again, the mean of the *AP* values over all queries is stated as *mean average precision (MAP).*

Mean Reciprocal Rank (MRR).

For a given list of queries Q, the *mean reciprocal rank (MRR)* is defined as the mean over all reciprocal ranks i of Q which indicates the position of the first correct answer [16].

$$\text{MRR} = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{\text{rank}_i} \tag{6}$$

Discounted Cumulative Gain (DCG@k).

The *DCG@k* considers all given ratings for a *top-k* result list and discounts them with a logarithmic factor of their position. This implies that highly relevant documents which appear lower in the search results get penalized [10].

$$\mathrm{DCG@k} = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)} \tag{7}$$

Since the *DCG@k* value is not normalized, the evaluation of it is limited. This can be done by calculating an *ideal DCG* for the perfect ranked result list. To do so, the rating for all items in data set is needed which is not the case for the used one. However, it makes sense to compare the absolute values of different evaluation setups.

5.3.3 Results

Readme-Based Recommendations.
The evaluation of the readme-based recommendations reveals a clear advantage of both the filtered and the unfiltered embedding performance over the *tf-idf* benchmark in terms of *P@k* (see Figure 9), *MAP*, *MRR* and *DCG@7* (see Table 2). In addition, the filtered version of the *doc2vec* embedding approach consistently outperforms the unfiltered version in all reported measures. In terms of the *MAP*, the unfiltered *doc2vec* has an edge over *tf-idf* of 18.7% (0.62 and 0.52, respectively). With the filter set, the advantage increases to 29.1%, or a score of 0.67 in absolute terms. The results for the *MRR* are analogous, with the unfiltered embedding score of 0.64 being a 16.0% increase and the filtered score (0.70) a 27.4% increase over the *tf-idf* benchmark (0.55). The same ranking in terms of relative performance also holds for the *DCG@7* measure.

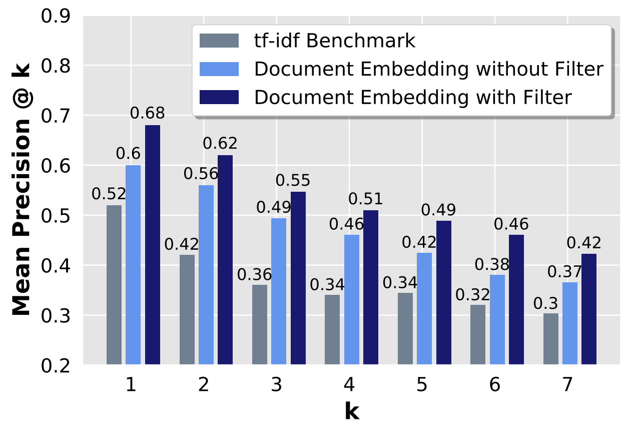


Figure 9: Precision@k results for content-based recommendations

ANN Architecture-Based Recommendations.
Figure 10 and table 3 show a clear trend in all used metrics: The use of an additional filter for the network type dramatically improved the measured performance. As Figure 10 shows, the *P@k* do not decrease in a linear manner. Without the additional filter, the overall mean is around 0.4, which follows a weak decreasing trend for all $k > 2$. For the filter setup, it lies around 0.24 with no clear trend. The *MRR* of 0.60 with filter is distinctly higher as without (0.42). The same goes for the MAP values of 0.53 (filter) and 0.39 (no filter). The absolute *DCG@7* exhibits an increase of around 44% in comparison to the no filter setup.

5.3.4 Discussion

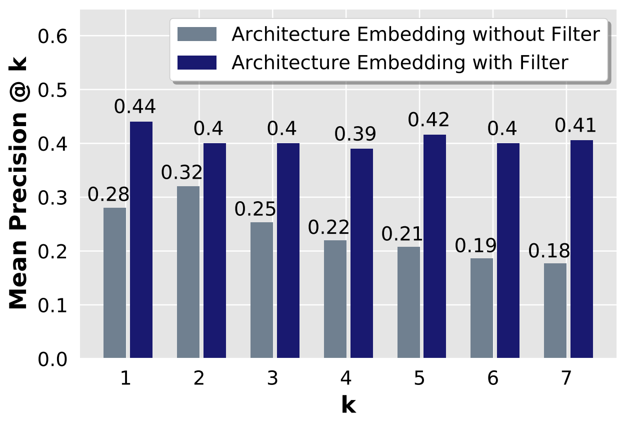


Figure 10: Precision@k results for architecture-based recommendations

Readme-Based Recommendations.
The results support the notion that recommending similar neural network repositories based on their readme file embeddings constitutes a valid and promising approach to recommending relevant repositories. The apparent performance advantage in terms of the reported measures of the readme file embedding approach over the *tf-idf* benchmark suggests that a document embedding approach is more suited for detecting text-based content similarities between repositories. We hypothesize that this finding can be attributed to the ability of document embeddings to capture the semantics of a document more holistically than a word-count-based approach which ignores the order of words and sentences in a document. The addition of the filter based on the extracted and inferred use cases of the neural networks boosts recommendation performance even further. This result, however, is not surprising considering the fact that filtering by the use case generally results in the filtering out of completely unrelated repositories which would have otherwise received the ratings "highly irrelevant" (1) or "irrelevant" (2). In practice, the combination of the embedding-based similarities and the automatically applied filter results in generally highly relevant recommendations for the vast majority of tested queries. For those query instances where results were poor, the lack of relevant recommendations can be traced back to the limited scope of the evaluation data set which in many cases only contains few samples related to the query topic. Conducting the evaluation on a larger data set may alleviate the effect of topic distribution and result in a more representative picture of recommendation quality.

ANN Architecture-Based Recommendations.
The evaluation of the recommendations based on the *ANN architecture embedding* show that with the selected parameters it is currently not possible to reliably identify similar *ANN* implementations. This can be derived from Figure 10 which shows that in neither of the two experimental setups it is possible to realize a precision higher than 0.5. However, this does not indicate that the proposed approach is not promising. Quite to the contrary, the measures *MRR*, *MAP* and *DCG@k*, which take the position of the results into account, show both solid values for each experimental setup. In particular, the scores of the filter setup (*MRR*: 0.60, *MAP*: 0.53, 44% increase of *DCG@7*) indicate that there is a potential to rank the ANN architectures in a consistent order. Since our ranking criteria is the similarity to the query architecture, a *MRR* of 0.60 indicates that the first relevant/highly similar implementation was shown slightly

Table 1: Rating Scale Used as proposed by [26]

Score	Relevance	Description
1	Highly Irrelevant	The participant finds that there is absolutely nothing in common between the retrieved and query repositories.
2	Irrelevant	The participant finds that the two repositories only have little in common.
3	Neutral	The participant finds that the two repositories are marginally relevant.
4	Relevant	The participant finds that the two repositories are similar on a number of aspects.
5	Highly Relevant	The participant finds that the retrieved and query repositories are similar in most aspects, and even some parts may be identical.

Table 2: Mean reciprocal rank, mean average precision and discounted cumulative gain at 7 results for content-based recommendations

Approach	MRR	MAP	DCG@7
tf-idf Benchmark	0.55	0.52	10.42
Embedding without Filter	0.64	0.62	12.46
Embedding with Filter	0.70	0.67	12.91

Table 3: Mean reciprocal rank, mean average precision and discounted cumulative gain at 7 results for architecture-based recommendations

Approach	MRR	MAP	DCG@7
Embedding without Filter	0.42	0.39	10.36
Embedding with Filter	0.60	0.53	14.96

more often at the first position on average. The *MAP* of 0.53 (*MAP** of 0.58, when queries without any relevant result are ignored) also indicates that the order of the relevant items tends to be promising. In other words, it can be assumed that an architecture embedding approach has the potential to differentiate between similar and less similar ANN designs.

Indeed, the effect is still limited and as discussed, the precision is still relatively low, but there are several limitations in our setup which could have accounted for this effect: First, as mentioned in section 5.3.1, we did not tune any hyperparameters during the embedding training due to limited computing power and a lack of labeled data. Moreover, the training corpus, the number of epochs and the used dimension of the resulting embedding are relatively small. The underlying ontology which defines the representation of the architectural network structure was further adopted without any changes. Opting for a representation which takes the ordering of layers into account might have contributed to a more capable embedding. All of the aforementioned limitations could explain the overall limited ability of the vector representation to capture the underlying semantics in a distinct manner. The increase in the *DCG@7* from the no filter to the filter setup, however, demonstrates the high potential for a combined approach with a filtered result list. In consequence, a comprehensive hyperparameter optimization on a larger data set and further refinements of the underlying ontology with advanced filtering methods may boost model performance.

6. CONCLUSION AND OUTLOOK

For a final conclusion of our project, we significantly expanded the capabilities of the *FAIRnets* search engine for neural networks by integrating an intelligent search mechanism and adding a promising recommender system functionality. The evaluation of our approach demonstrates notable improvements with respect to finding and recommending items of interest. Our novel search mechanism retrieves search results not only based on the title and description of the respective GitHub repository, but integrates more comprehensive contextual data such as the assigned topic tags among others. Our conceived activity score and derived ranking allow space for individual adjustment depending on the user's preference for hard, quantitative measures and more qualitative textual information. In terms of the added recommender functionality, we employ two separate embedding approaches for the tasks of identifying and recommending similar repositories based on readme content and neural network architecture. By that means, we achieve superior results compared to a simple benchmark approach. Our research also shows that filtering the recommendation list by relevant project characteristics further enhances the quality of recommendations.

Our exploratory research efforts offer interesting opportunities for further research. As we used only a small subset of the available data for evaluation and a small number of expert judges, it would be worthwhile to test our approaches on a more comprehensive data set to boost both internal and external validity. Adding user characteristics and preferences to our purely content-based recommender system might increase the performance by delivering even more relevant recommendations to the user. Thus, using a hybrid recommendation approach as proposed in [23] represents a valuable addition to our methodology. As an alternative to the currently used input format of keywords, we consider a more advanced search as a possible advancement so that the user can describe his search problem in more detail.

7. REFERENCES

[1] H. Borges, A. Hora, and M. T. Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344. IEEE, 2016.

[2] F. Chollet et al. Keras, 2015.

[3] W. Consortium. Resource description framework (rdf): Concepts and abstract syntax, 2004.

[4] A. M. Dai, C. Olah, and Q. Le V. Document embedding with paragraph vectors, 2015.

[5] Django Project. Django, 2019.

[6] I. GitHub. Github rest api v3, 2019.

[7] Google. Google news data set, 2013.

[8] T. H. Haveliwala. Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, pages 517–526. ACM, 2002.

[9] D. Hawking, N. Craswell, P. Bailey, and K. Griffihs. Measuring search engine quality. *Information Retrieval*, 4(1):33–59, 2001.

[10] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[11] T. Joachims. Optimizing search engines using clickthrough data, 2002.

[12] Q. Le V and T. Mikolov. Distributed representations of sentences and documents, 2014.

[13] W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 51 – 56, 2010.

[14] A. Nguyen, T. Weller, and Y. Sure-Vetter. Fairnets, 2019.

[15] T. D. Nguyen. Intelligent search engine for associated on-line documentation having questionless case-based knowledge base, Aug. 22 1995. US Patent 5,444,823.

[16] D. R. Radev, H. Qi, H. Wu, and W. Fan. Evaluating web-based question answering systems. In *LREC*, 2002.

[17] R. Řehůřek and P. Sojka. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, pages 45–50, Valletta, Malta, May 2010. ELRA. http://is.muni.cz/publication/884893/en.

[18] P. Ristoski and H. Paulheim. Rdf2vec: Rdf graph embeddings for data mining. In *International Semantic Web Conference*, pages 498–514. Springer, 2016.

[19] J. Rosati, P. Ristoski, T. Di Noia, R. d. Leone, and H. Paulheim. Rdf graph embeddings for content-based recommender systems. In *CEUR workshop proceedings*, volume 1673, pages 23–30. RWTH, 2016.

[20] M. Sanderson. Christopher d. manning, prabhakar raghavan, hinrich schütze, introduction to information retrieval, cambridge university press. 2008. isbn-13 978-0-521-86571-5, xxi+ 482 pages. *Natural Language Engineering*, 16(1):100–103, 2010.

[21] A. Sharma, F. Thung, P. S. Kochhar, A. Sulistya, and D. Lo. Cataloging github repositories. In E. Mendes, S. Counsell, and K. Petersen, editors, *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering - EASE'17*, pages 314–319, New York, New York, USA, 2017. ACM Press.

[22] C. Spearman. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[23] X. Sun, W. Xu, X. Xia, X. Chen, and B. Li. Personalized project recommendation on github. *Science China Information Sciences*, 61(5):355, 2018.

[24] A. Turpin and F. Scholer. User performance versus precision measures for simple search tasks. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 11–18. ACM, 2006.

[25] L. Zhang, Y. Zou, B. Xie, and Z. Zhu. Recommending relevant projects via user behaviour: an exploratory study on github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, pages 25–30. ACM, 2014.

[26] Y. Zhang, Lo D, P. S. Kochhar, X. Xia, Q. Li, and J. Sun. Detecting similar repositories on github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 13–23, 2017.