Abstract—Programming languages themselves have a limited number of reserved keywords and character based tokens that define the language specification. However, programmers have a rich use of natural language within their code through comments, text literals and naming entities. The programmer defined names that can be found in source code are a rich source of information to build a high level understanding of the project. The goal of this paper is to apply topic modeling to names used in over 13.6 million repositories and perceive the inferred topics. One of the problems in such a study is the occurrence of duplicate repositories not officially marked as forks (obscure forks). We show how to address it using the same identifiers which are extracted for topic modeling.

We open with a discussion on naming in source code, we then elaborate on our approach to remove exact duplicate and fuzzy duplicate repositories using Locality Sensitive Hashing on the bag-of-words model and then discuss our work on topic modeling; and finally present the results from our data analysis together with open-access to the source code, tools and datasets.

Index Terms—programming, open source, source code, software repositories, git, GitHub, topic modeling, ARTM, locality sensitive hashing, MinHash, open dataset, data.world.

I. INTRODUCTION

There are more than 18 million non-empty public repositories on GitHub which are not marked as forks. This makes GitHub the largest version control repository hosting service. It has become difficult to explore such a large number of projects and nearly impossible to classify them. One of the main sources of information that exists about public repositories is their code.

To gain a deeper understanding of software development it is important to understand the trends among open-source projects. Bleeding edge technologies are often used first in open source projects and later employed in proprietary solutions when they become stable enough\(^1\). An exploratory analysis of open-source projects can help to detect such trends and provide valuable insight for industry and academia.

Since GitHub appeared the open-source movement has gained significant momentum. Historically developers would manually register their open-source projects in software digests. As the number of projects dramatically grew, those lists became very hard to update; as a result they became more fragmented and started exclusively specializing in narrow technological ecosystems. The next attempt to classify open source projects was based on manually submitted lists of keywords. While this approach works \(^1\), it requires careful keywords engineering to appear comprehensive, and thus not widely adopted by the end users in practice. GitHub introduced repository tags in January 2017 which is a variant of manual keywords submission.

The present paper describes how to conduct fully automated topic extraction from millions of public repositories. It scales linearly with the overall source code size and has substantial performance reserve to support the future growth. We propose building a bag-of-words model on names occurring in source code and applying proven Natural Language Processing algorithms to it. Particularly, we describe how “Weighted MinHash” algorithm \(^2\) helps to filter fuzzy duplicates and how an Additive Regularized Topic Model (ARTM) \(^3\) can be efficiently trained. The result of the topic modeling is a nearly-complete open source projects classification. It reflects the drastic variety in open source projects and reflect multiple features. The dataset we work with consists of approx. 18 million public repositories retrieved from GitHub in October 2016.

The rest of the paper is organised as follows: Section II reviews prior work on the subject. Section III elaborates on how we turn software repositories into bags-of-words. Section IV describes the approach to efficient filtering of fuzzy repository clones. Section V covers the building of the ARTM model with 256 manually labeled topics. Section VI presents the achieved topic modeling results. Section VII lists the opened datasets we were able to prepare. Finally, section VIII presents a conclusion and suggests improvements to future work.

II. RELATED WORK

A. Academia

There was an open source community study which presented statistics about manually picked topics in 2005 by J. Xu et.al. \(^4\).

Blincoe et.al. \(^5\) studied GitHub ecosystems using reference coupling over the GHTorrent dataset \(^6\) which contained 2.4 million projects. This research employs an alternative topic modeling method on source code of 13.6 million projects.

\(^1\)Notable examples include the Linux OS kernel, the PostgreSQL database engine, the Apache Spark cluster-computing framework and the Docker containers.
Instead of using the GHTorrent dataset we’ve prepared open datasets from almost all public repositories on GitHub to be able to have a more comprehensive overview.

M. Lungi [7] conducted an in-depth study of software ecosystems in 2009, the year when GitHub appeared. The examples in this work used samples of approx. 10 repositories. And the proposed discovery methods did not include Natural Language Processing.

The problem of the correct analysis of forks on GitHub has been discussed by Kalliamvakou et.al. [8] along with other valuable concerns.

Topic modeling of source code has been applied to a variety of problems reviewed in [9]: improvement of software maintenance [10], [11], defects explanation [12], concept analysis [13], [14], software evolution analysis [15], [16], finding similarities and clones [17], clustering source code and discovering the internal structure [18], [19], [20], summarizing [21], [22], [23]. In the aforementioned works, the scope of the research was focused on individual projects.

The usage of topic modeling [24] focused on improving software maintenance and was evaluated on 4 software projects. Concepts were extracted using a corpus of 24 projects in [25]. Giriprasad Sridhara et.al. [26], Yang and Tan [27], Howard et.al. [28] considered comments and/or names to find semantically similar terms; Haiduc and Marcus [29] researched common domain terms appearing in source code. The presented approach in this paper reveals similar and automatically generated files. The first step in our preprocessing is to run linguist over each repository’s master branch. From 11.94 million repositories we end up with 402.6 million source files in which we have high confidence it is source code written by a developer in that project. Identifying the programming language used within each file is important for the next step, the names extraction, as it determines the programming language parser.

### B. Extracting Names

Source code highlighting is a typical task for professional text editors and IDE’s. There have been several open source libraries created to tackle this task. Each works by having a grammar file written per programming language which contains the rules. Pygments [36] is a high quality community-driven package for Python which supports more than 400 programming languages and markups. According to Pygments, all source code tokens are classified across the following categories: comments, escapes, indentations and generic symbols, reserved keywords, literals, operators, punctuation and names.

Linguist and Pygments have different sets of supported languages. Linguist stores it’s list at master/lib/linguist/languages.yml and the similar Pygments list is stored as pygments.lexers.LEXERS. Each has nearly 400 items and the intersection is approximately 200 programming languages (“programming” Linguist’s item type). The languages common to Linguist and Pygments which were chosen are listed in appendix A. In this research we apply Pygments to the 402.6 million source files to extract all tokens which belong to the type Token.Name.

### C. Processing names

The next step is to process the names according to naming conventions. As an example class FooBarBaz adds three words to the bag: foo, bar and baz, or int wdSize should add two: wdsize and size. Fig. 1 is the full listing of the function written in Python 3.4+ which splits identifiers.

In this step each repository is saved as an sqlite database file which contains a table with: programming language, name extracted and frequency of occurrence in that repository. The total number of unique names that were extracted were 17.28 million.
NAME_BREAKUP_RE = re.compile(r"[\^a-zA-Z]+")

def extract_names(token):
    token = token.strip()
    prev_p = ['"]

    def ret(name):
        r = name.lower()
        if len(name) >= 3:
            yield r
        if prev_p[0]:
            yield prev_p[0] + r
        prev_p[0] = ""
        else:
            prev_p[0] = r

    for part in NAME_BREAKUP_RE.split(token):
        if not part:
            continue
        prev = part[0]
        pos = 0
        for i in range(1, len(part)):
            this = part[i]
            if prev.islower() and this.isupper():
                yield from ret(part[pos:i])
                pos = i
            elif prev.isupper() and this.islower():
                if 0 < i - 1 - pos <= 3:
                    yield from ret(part[pos:i - 1])
                    pos = i - 1
                elif i - 1 > pos:
                    yield from ret(part[pos:i])
                    pos = i
            prev = this
        last = part[pos:]
        if last:
            yield from ret(last)

D. Stemming names

It is common to stem names when creating a bag-of-words in NLP. Since we are working with natural language that is predominantly English we have applied the Snowball stemmer [37] from the Natural Language Toolkit (NLTK) [38]. The stemmer was applied to names which were >6 characters long. In further research a diligent step would be to compare results with and without stemming of the names, and also to predetermine the language of the name (when available) and apply stemmers in different languages.

The length of words on which stemming was applied was chosen after the manual observation that shorter identifiers tend to collide with each other when stemmed and longer identifiers need to be normalized. Fig. 2 represents the distribution of identifier lengths in the dataset:

It can be seen that the most common name length is 6. Fig. 3 is the plot of the number of unique words in the dataset depending on the stemming threshold:

We observe the breaking point at length 5. The vocabulary size linearly grows starting with length 6. Having manually inspected several collisions on smaller thresholds, we came to the conclusion that 6 corresponds to the best trade-off between collisions and word normalization.

The Snowball algorithm was chosen based on the comparative study by Jivani [39]. Stems not being real words are acceptable but it is critical to have minimum over-stemming since it increases the number of collisions. The total number of unique names are 16.06 million after stemming.

To be able to efficiently pre-process our data we used Apache Spark [40] running on 64 4-core nodes which allowed us to process repositories in parallel in less than 1 day.

However, before training a topic model one has to exclude near-duplicate repositories. In many cases GitHub users include the source code of existing projects without preserving the commit history. For example, it is common for web sites, blogs and Linux-based firmwares. Those repositories contain very little original changes and may introduce frequency noise into the overall names distribution. This paper suggests the way to filter the described fuzzy duplicates based on the bag of words model built on the names in the source code.
IV. Filtering near-duplicate repositories

There were more than 70 million GitHub repositories in October 2016 by our estimation. Approx. 18 million were not marked as forks. Nearly 800,000 repositories were de facto forks but not marked correspondingly by GitHub. That is, they had the same git commit history with colliding hashes. Such repositories may appear when a user pushes a cloned or imported repository under his or her own account without using the GitHub web interface to initiate a fork.

When we remove such hidden forks from the initial 18 million repositories, there still remain repositories which are highly similar. A duplicate repository is sometimes the result of a git push of an existing project with a small number of changes. For example, there are a large number of repositories with the Linux kernel which are ports to specific devices. In another case, repositories containing web sites were created using a cloned web engine and preserving the development history. Finally, a large number of github.io repositories are the same. Such repositories contain much text content and few identifiers which are typically the same (HTML tags, CSS rules, etc.).

Filtering out such fuzzy forks speeds up the future training of the topic model and reduces the noise. As we obtained a bag-of-words for every repository, the naive approach would be to measure all pairwise similarities and find cliques. But first we need to define what is the similarity between bags-of-words.

A. Weighted MinHash

Suppose that we have two dictionaries - key-value mappings with unique keys and values indicating non-negative "weights" of the corresponding keys. We would like to introduce a similarity measure between them. The Jaccard Similarity between dictionaries \( A = \{i : a_i\}, i \in I \) and \( B = \{j : b_j\}, j \in J \) is defined as

\[
J = \frac{\sum_{k \in K} \min(a_k, b_k)}{\sum_{k \in K} \max(a_k, b_k)} = I \cup J
\]

where \( a_k = 0, k \notin I \) and \( b_k = 0, k \notin J \). If the weights are binary, this formula is equivalent to the common Jaccard Similarity definition.

The same way as MinHash is the algorithm to find similar sets in linear time, Weighted MinHash is the algorithm to find similar dictionaries in linear time. Weighted MinHash was introduced by Ioffe in [2]. We have chosen it in this paper because it is very efficient and allows execution on GPUs instead of large CPU clusters. The proposed algorithm depends on the parameter \( K \) which adjusts the resulting hash length.

1. for \( k \) in range\((K)\):
   1.1. Sample \( r_k, \epsilon_k \sim Gamma(2, 1) \) - Gamma distribution (their PDF is \( P(r) = re^{-r} \)), and \( \beta_k \sim Uniform(0, 1) \).

1.2. Compute

\[
t_k = \left\lfloor \frac{\ln S_k}{r_k} \right\rfloor + \beta_k
\]

\[
y_k = e^{r_k(t_k - \beta_k)}
\]

\[
z_k = y_k \epsilon_k
\]

\[
a_k = \frac{c_k}{z_k}
\]

2. Find \( k^* = \arg \min_k \ a_k \) and return the samples \((k^*, t_{k^*})\). Thus given \( K \) and supposing that the integers are 32-bit we obtain the hash with size \( 8K \) bytes. Samples from \( Gamma(2, 1) \) distribution can be efficiently calculated as \( r = -\ln(u_1 u_2) \) where \( u_1, u_2 \sim Uniform(0, 1) \) - uniform distribution between 0 and 1.

We developed the MinHashCUDA [41] library and Python native extension which is the implementation of Weighted MinHash algorithm for NVIDIA GPUs using CUDA [42]. There were several engineering challenges with that implementation which are unfortunately out of the scope of this paper. We were able to hash all 10 million repositories with hash size equal to 128 in less than 5 minutes using MinHashCUDA and 2 NVIDIA Titan X Pascal GPU cards.

B. Locality Sensitive Hashing

Having calculated all the hashes in the dataset, we can perform Locality Sensitive Hashing. We define several hash tables, each for it’s own sub-hash which depends on the target level of false positives. Same elements will appear in the same bucket; union of the bucket sets across all the hash tables for a specific sample yields all the similar samples. Since our goal is to determine the sets of mutually similar samples, we should consider the set intersection instead.

We used the implementation of Weighted MinHash LSH from Datasetch [43]. It is designed after the corresponding algorithm in Mining of Massive Datasets [44]. LSH takes a single parameter - the target Weighted Jaccard Similarity value (“threshold”). MinHash LSH puts every repository in a number of separate hash tables which depend on the threshold and the hash size. We used the default threshold 0.9 in our experiments which ensures a low level of dissimilarity within a hash table bin.

Algorithm 5 describes the fuzzy duplicates detection pipeline. Step 6 discards less than 0.5% of all the sets and aims at reducing the number of false positives. The bins size distribution after step 5 is depicted on Fig. 4 - it is clearly seen that the majority of the bins has the size 2. Step 6 uses Weighted Jaccard similarity threshold 0.8 instead of 0.9 to be sensitive to evident outliers exclusively.

Table I reveals how different hash sizes influence on the resulting number of fuzzy clones:

Algorithm 5 results in approximately 467,000 sets of fuzzy duplicates with overall 1.7 million unique repositories. Each repository appears in two sets on average. The examples of fuzzy duplicates are listed in appendix B. The detection algorithm works especially well for static web sites which share the same JavaScript libraries.
1: Calculate Weighted MinHash with hash size 128 for all the repositories.
2: Feed each hash to MinHash LSH with threshold 0.9 so that every repository appears in each of the 5 hash tables.
3: Filter out hash table bins with single entries.
4: For every repository, intersect the bins it appears in across all the hash tables. Cache the intersections, that is, if a repository appears in the same existing set, do not create the new one.
5: Filter out sets with a single element. The resulting number of unique repositories corresponds to "Filtered size" in Table I.
6: For every set with 2 items, calculate the precise Weighted Jaccard similarity value and filter out those with less than 0.8 (optional).
7: Return the resulting list of sets. The number of unique repositories corresponds to "Final size" in Table I.

After the exclusion of the fuzzy duplicates, we finish dataset processing and pass over to training of the topic model. The total number of unique names has now reduced by 2.06 million to 14 million unique names. To build a meaningful dataset, names with occurrence of less than $T_f = 20$ were excluded from the final vocabulary. 20 was chosen on the frequency histogram shown on Fig. 6 since it is the drop-off point.

After this exclusion, there are now 2 million unique names, with an average size of a bag-of-words of 285 per repository and Fig. 7 displays the heavy-tailed bag size distribution.

### V. TRAINING THE ARTM TOPIC MODEL

This section revises ARTMs and describes how the training of the topic model was performed. We have chosen ARTM instead of other topic modeling algorithms since it has the most efficient parallel CPU implementation in bigARTM according to our benchmarks.

#### A. Additive Regularized Topic Model

Suppose that we have a topic probabilistic model of the collection of documents $D$ which describes the occurrence of terms $w$ in document $d$ with topics $t$:

$$p(w|d) = \sum_{t \in T} p(w|t)p(t|d).$$

Here $p(w|t)$ is the probability of the term $w$ to belong to the topic $t$, $p(t|d)$ is the probability of the topic $t$ to belong to the document $d$, thus the whole formula is just an expression of the total probability, accepting the hypothesis of conditional independence: $p(w|d, t) = p(w|t)$. Terms belong
to the vocabulary $W$, topics are taken from the set $T$ which is simply the series of indices $[1, 2, \ldots n_t]$. We'd like to solve the problem of recovering $p(w|t)$ and $p(t|d)$ from the given set of documents \( \{d \in D : d = \{w_1 \ldots w_{n_d}\}\} \). We normally assume $\hat{p}(w|t) = \frac{n_{dw}}{n_d}$, $n_{dw}$ being the number of times term $w$ occurred in document $d$, but this implies that all the terms are equally important which is not always true. "Importance" here means some measure which negatively correlates with the overall frequency of the term. Let us denote the recovered probabilities as $\hat{p}(w|t) = \phi_{wt}$ and $\hat{p}(t|d) = \theta_{td}$. Thus our problem is the stochastic matrix decomposition which is not correctly stated:

$$\frac{n_{dw}}{n_d} \approx \Phi \cdot \Theta = (\Phi S)(S^{-1}\Theta) = \Phi' \cdot \Theta'. \quad (7)$$

The stated problem can be solved by applying maximum likelihood estimation:

$$\sum_{d \in D} \sum_{w \in d} n_{dw} \ln \sum_{t} \phi_{wt} \theta_{td} \rightarrow \max_{\Phi, \Theta} \quad (8)$$

upon the conditions

$$\phi_{wt} > 0; \sum_{w \in W} \phi_{wt} = 1; \theta_{td} > 0; \sum_{t \in T} \theta_{td} = 1. \quad (9)$$

The idea of ARTM is to naturally introduce regularization as one or several extra additive members:

$$\sum_{d \in D} \sum_{w \in d} n_{dw} \ln \sum_{t} \phi_{wt} \theta_{td} + R(\Phi, \Theta) \rightarrow \max_{\Phi, \Theta} \quad (10)$$

Since this is a simple summation, one can combine a series of regularizers in the same objective function. For example, it is possible to increase $\Phi$ and $\Theta$ sparsity or to make topics less correlated. Well-known LDA model [45] can be reproduced as ARTM too.

The variables $\Phi$ and $\Theta$ can be effectively calculated using the iterative expectation maximization algorithm [46]. Many ready to be used ARTM regularizers are already implemented in the BigARTM open source project [47].

**B. Training**

Vorontsov shows in [3] that ARTM is trained best if the regularizers are activated sequentially, with a lag relative to each other. For example, first EM iterations are performed without any regularizers at all and the model reaches target perplexity, then $\Phi$ and $\Theta$ sparsity regularizers are activated and the model optimizes for those new members in the objective function while not increasing the perplexity. Finally other advanced regularizers are appended and the model minimizes the corresponding members while leaving the old ones intact.

We apply only $\Phi$ and $\Theta$ sparsity regularizers in this paper. Further research is required to leverage others. We experimented with the training of ARTM on the source code identifiers from III and observed that the final perplexity and sparsity values do not change considerably on the wide range of adjustable meta-parameters. The best training meta-parameters are given in Table II.

We chose 256 topics merely because it is time intensive to label them and 256 was the largest amount we could label. The traditional ways of determining the optimal number of topics using e.g. Elbow curves are not applicable to our data. We cannot consider topics as clusters since a typical repository corresponds to several topics and increasing the number of topics worsens the model’s generalization and requires the dedicated topics decorrelation regularizer. The overall number of iterations equals 18. The convergence plot is shown on Fig. 8.

The achieved quality metric values are as given in Table III.

On the average, a single iteration took 40 minutes to complete on our hardware. We used BigARTM in a Linux environment on 16-core (32 threads) Intel(R) Xeon(R) CPU E5-2620 v4 computer with 256 GB of RAM. BigARTM supports the parallel training and we set the number of workers to 30. The peak memory usage was approximately 32 GB.

It is possible to relax the hardware requirements and speed up the training if the model size is reduced. If we set the frequency threshold $T_f$ to a greater value, we can dramatically reduce the input data size with the risk of losing the ability of the model to generalize.

**TABLE II**

| Parameter                  | Value |
|----------------------------|-------|
| Topics                     | 256   |
| Iterations without regularizers | 10    |
| Iterations with regularizers | 8     |
| $\Phi$ sparsity weight     | 0.5   |
| $\Theta$ sparsity weight   | 0.5   |

**TABLE III**

| Metric     | Value  |
|------------|--------|
| Perplexity | 10168  |
| $\Phi$ sparsity | 0.964  |
| $\Theta$ sparsity | 0.913  |
We further normalize each row of the matrix by $L^2$ to get the repository embeddings:

$$R_t = R_n \times T_n. \quad \text{(11)}$$

We further normalize each row of the matrix by $L^2$ metric:

$$R_t^{\text{normed}} = \text{rowwise} \frac{R_t}{\|R_t\|_2}. \quad \text{(12)}$$

The sum along every column of this matrix indicates the significance of each topic. Fig. 9 shows the distribution of this measure.

C. Converting the repositories to the topics space

Let $R_t$ be the matrix of repositories in the topics space of size $R \times T$, $R_n$ be the sparse matrix representing the dataset of size $R \times N$ and $T_n$ be the matrix representing the trained topic model of size $N \times T$. We perform the matrix multiplication to get the repository embeddings:

$$R_t = R_n \times T_n. \quad \text{(11)}$$

We further normalize each row of the matrix by $L^2$ metric:

$$R_t^{\text{normed}} = \text{rowwise} \frac{R_t}{\|R_t\|_2}. \quad \text{(12)}$$

The sum along every column of this matrix indicates the significance of each topic. Fig. 9 shows the distribution of this measure.

VI. TOPIC MODELING RESULTS

The employed topic model is unable to summarize the topics the same way humans do. It is possible to interpret some topics based on the most significant words, some based on relevant repositories, but many require manual supervision with the careful analysis of most relevant names and repositories. This supervision is labour intensive and the single topic normally takes up to 30 minutes to summarize with proper confidence. 256 topics required several man-days to complete the analysis.

After a careful analysis, we sorted the labelled topics into the following groups:

- **Concepts** (41) - general, broad and abstract. The most interesting group. It includes scientific terms, facts about the world and the society.
- **Human languages** (10) - it appeared that one can determine programmer’s approximate native language looking at his code, thanks to the stem bias.
- **Programming languages** (33) - not so interesting since this is the information we already have after linguistic classification. Programming languages usually have a standard library of classes and functions which is imported/included into most of the programs, and the corresponding names are revealed by our topic modeling. Some topics are more narrow than a programming language.
- **General IT** (72) - the topics which could appear in Concepts if had an expressive list of key words but do not. The repositories are associated by the unique set of names in the code without any special meaning.
- **Technologies** (87) - devoted to some specific, potentially narrow technology or product. Often indicates an ecosystem or community around the technology.
- **Games** (13) - related to video games. Includes specific gaming engines.

The complete topics list is in appendix C. The example topic labelled "Machine Learning, Data Science” is shown in appendix D.

It can be observed that some topics are dual and need to be splitted. That duality is a sign that the number of topics should be bigger. At the same time, some topics appear twice and need to be de-correlated, e.g. using the "decorrelation" ARTM regularizer. Simple reduction or increase of the number of topics however do not solve those problems, we found it out while experimenting with 200 and 320 topics.

VII. RELEASED DATASETS

We generated several datasets which were extracted from our internal 100 TB GitHub repository storage. We incorporated them on data.world [48], the recently emerged “GitHub for data scientists”, each has the description, the origin note and the format definition. They are accessed at data.world/vmarkovtsev. Besides, the datasets are uploaded to Zenodo and have DOI. They are listed in Table VII.

| Name and DOI | Description |
|--------------|-------------|
| source code names 10.5281/zenodo.284554 | names extracted from 13,000,000 repositories (fuzzy clones excluded) considered in section III |
| 452,000,000 commits 10.5281/zenodo.285467 | metadata of all the commits in 16,000,000 repositories (fuzzy clones excluded) |
| keyword frequencies 10.5281/zenodo.285293 | frequencies of programming language keywords (reserved tokens) across 16,000,000 repositories (fuzzy clones excluded) |
| README files 10.5281/zenodo.285419 | README files extracted from 16,000,000 repositories (fuzzy clones excluded) |
| duplicate repositories 10.5281/zenodo.285377 | fuzzy clones which were considered in section IV |

VIII. CONCLUSION AND FUTURE WORK

Topic modeling of GitHub repositories is an important step to understanding software development trends and open source
communities. We built a repository processing pipeline and applied it to more than 18 million public repositories on GitHub. Using developed by us open source tool MinHashCUDA we were able to remove 1.6 million fuzzy duplicate repositories from the dataset. The preprocessed dataset with source code names as well as other datasets are open and the presented results can be reproduced. We trained ARTM on the resulting dataset and manually labelled 256 topics. The data processing and model training are possible to perform using a single GPU card and a moderately sized Apache Spark cluster. The topics covered a broad range of projects but there were repeating and dual ones. The chosen number of topics was enough for general exploration but not enough for the complete description of the dataset.

Future work may involve experimentation with clustering the repositories in the topic space and comparison with clusters based on dependency or social graphs [49].

### APPENDIX A

**PARSED LANGUAGES**

| Language | Language | Language | Language |
|----------|----------|----------|----------|
| abap     | coldfusion | hy       | moocode  |
| abl      | common lisp | i7       | moonscript |
| actionscript | component | idl     | mapd     |
| ada      | pascal    | igor     | nasm     |
| agda     | console   | igorpro  | nemerle  |
| akh      | coq       | inform 7 | nesc     |
| alloy    | csharp    | io       | newlisp  |
| antr     | csound    | j        | nimmrod  |
| apl      | cucumber  | j        | nit      |
| appscript | cuda     | isabelle | nix      |
| arduino  | cython    | jasmin   | nixos    |
| as3      | d         | java     | nsis     |
| aspectj  | dart      | javascript | numpy  |
| aspx-vb  | delphi    | jsp      | obj-c    |
| autohotkey | dosbatch | julia    | obj-c++  |
| autot    | dylan     | kotlin   | obj-j    |
| awk      | ec        | lasso    | objectpascal |
| bash     | eiffel    | lassoscript | octave  |
| batchfile | elisp    | lean     | ocaml    |
| befunge  | exir      | liaskell | ooc      |
| blitzbasic | eln     | lls      | opa      |
| blitzmax | emacs     | limbo    | openedge |
| bmax     | erlang    | lisp     | pan      |
| bpo      | factor    | literate agda | pascal  |
| bplus    | fancy     | literate | pawn     |
| brainfuck | fantom   | haskell  | perl     |
| bro      | fish      | livescript | php    |
| bsdmake  | fortran   | llvm     | pike     |
| c        | forxpro   | logos    | plpgsql  |
| c#       | fsharp    | logtalk  | posh     |
| c++      | gap       | lsl      | povray   |
| ceylon   | gas       | lua      | powershell |
| cfc      | genshi    | make     | progress |
| cfn      | gherkin   | mako     | prolog   |
| chapel   | ghl       | mathematica | puppet |
| chpl     | gnuplot   | mailab   | pyrex    |
| cirru    | go        | mf       | python   |
| clipper  | golo      | minid    | qml      |
| clojure  | gosu      | mmal     | robotframework |
| cmake    | groovy    | modelica |       |
| cobol    | haskell   | modula-2 |       |
| coffeescrit | haxe    | monkey   |       |
| r        | scheme    |        |         |
| racket   | scilab    |        |         |
| ragel    | shell     |        |         |
| rb       | shen      |        |         |
| rebo     | smalli    |        |         |
| red      | smalltalk |        |         |
| redcode  | smarty    |        |         |
| ruby     | sml       |        |         |
| rust     | sourcepawn |        |         |
| sage     | splus     |        |         |
| salt     | squeak    |        |         |
| scala    | stan      |        |         |
| scala    |          |        |         |
| scala    |          |        |         |

### APPENDIX B

**EXAMPLES OF FUZZY DUPLICATE REPOSITORIES**

#### A. Linux kernel
- 1406/linux-0.11
- yi5971/linux-0.11
- love520134/linux-0.11
- wfrewood/source-linux-0.11
- sunrunning/linux-0.11
- Aaron123/linux-0.11
- junjee/linux-0.11
- pengdonglin137/linux-0.11
- yankatosat/linux-0.11

#### B. Tutorials
- dcarbajosa/linuxacademy-chef
- jachinh/linuxacademy-chef
- flarotag/linuxacademy-chef
- qhawk/linuxacademy-chef
- paul-e-allen/linuxacademy-chef

#### C. Web applications 1
- choysama/my-django-first-blog
- mihue/jdago-first
- PubMahesh/my-first-django-app
- nickmalhotra/first-django-blog
- Jmeggesto/MyFirstDjango
- atlwendy/django-first
- susancodes/first-django-app
- quipper7/DjangoFirstProject
- phidang/first-django-blog

#### D. Web applications 2
- iggitie/omrails
- ilrobinson81/omrails
- OCushman/omrails
- hambini/One-Month-Rails
- Ben2pop/omrails
- chrisloruss/omrails
- arjunurs/omrails
- crazystingray/omrails
- scorcoran33/omrails
- Joelf001/Omrails
APPENDIX C

COMPLETE LIST OF LABELLED TOPICS

A. Concepts

1) 2D geometry
2) 3D geometry
3) Arithmetic
4) Audio
5) Bitcoin
6) Card Games
7) Chess; Hadoop #
8) Classical mechanics (physics)
9) Color Manipulation/Generation
10) Commerce, ordering
11) Computational Physics
12) Date and time
13) Design patterns; HTML parsing
14) Email *
15) Email *
16) Enumerators, Mathematical Expressions
17) Finance and trading
18) Food (eg. pizza, cheese, beverage), Calculator
19) Genomics
20) Geolocalization, Maps
21) Graphs
22) Hexademical numbers
23) Human
24) Identifiers
25) Language names; JavaFX #
26) Linear Algebra; Optimization
27) Machine Learning, Data Science
28) My
29) Parsing
30) Particle physics
31) Person Names (American)
32) Personal Information
33) Photography, Flickr
34) Places, transportation, travel
35) Publishing; Flask #
36) Space and solar system
37) Sun and moon
38) Trade
39) Trees, Binary Trees
40) Video; movies
41) Word Term

B. Human languages

42) Chinese
43) Dutch
44) French *
45) French *
46) German
47) Portuguese *
48) Portuguese *
49) Spanish *
50) Spanish *
51) Vietnamese

C. Programming languages

52) Assembler
53) Autoconf
54) Clojure
55) ColdFusion *
56) ColdFusion *
57) Common LISP
58) Emacs LISP
59) Emulated assembly
60) Go
61) HTML
62) Human education system
63) Java AST and bytecode
64) libc
65) Low-level PHP
66) Lua *
67) Lua *
68) Makefiles
69) Mathematics: proofs, sets
70) Matlab
71) Object Pascal
72) Objective-C
73) Perl
74) Python
75) Python, ctypes
76) Ruby
77) Ruby with language extensions
78) SQL
79) String Manipulation in C
80) Verilog/VHDL
81) Work, money, employment, driving, living
82) x86 Assembler *
83) x86 Assembler *
84) XPCOM

D. General IT

85) 3-char identifiers
86) Advertising (Facebook, Ad Engines, Ad Blockers, AdMob)
87) Animation
88) Antisipam; PHP forums
89) Antivirus; database access #
90) Barcodes; browser engines #
91) Charting
92) Chat; messaging
93) Chinese web
94) Code analysis and generation
95) Computer memory and interfaces
96) Console, terminal, COM
97) CPU and kernel
98) Cryptography
99) Date and time picker
100) DB Sharding, MongoDB sharding
101) Design patterns; formal architecture
102) DevOps
103) Drawing *
104) Drawing *
105) Forms (UI)
106) Glyphs; X11 and FreeType #
107) Grids and tables
108) HTTP auth
109) iBeacons
110) Image Manipulation
111) Image processing
112) Intel SIMD, Linear Algebra #
113) IO operations
114) Javascript selectors
115) JPEG and PNG
116) Media Players
117) Metaprogramming
118) Modern JS frontend (Bower, Grunt, Yeoman)
119) Names starting with “m”
120) Networking
121) OAuth; major web services #
122) Observer design pattern
123) Online education; Moodle
124) OpenGL *
125) Parsers and compilers
126) Plotting
127) Pointers
128) POSIX Shell; VCS #
129) Promises and deferred execution; Angular #
130) Proof of concept
131) RDF and SGML parsing
132) Request and Response
133) Requirements and dependencies
134) Sensors; DIY devices
135) Sockets C API
136) Sockets, Networking
137) Sorting and searching
138) SQL database
139) SQL DB, XML in PHP projects
140) SSL
141) Strings
142) Testing with mocks
143) Text editor UI
144) Threads and concurrency
145) Typing suggestions and dropdowns
146) UI
147) Video player
148) VoIP
149) Web Media, Arch Packages #
150) Web posts
151) Web testing; crawling
152) Web UI
153) Wireless
154) Working with buffers
155) XML (SAX, XSL)
156) XMP
157) .NET
158) Android Apps
159) Android UI
160) Apache Libraries for BigData
161) Apache Thrift
162) Arduino, AVR
163) ASP.NET *
164) ASP.NET *
E. Technologies

165) Backbone.js
166) Chardet (Python)
167) Cocos2D
168) Comp. vision; OpenCV
169) Cordova
170) CPython
171) Crumbs; cake(PHP)
172) cURL
173) DirectDraw
174) DirectX
175) Django Web Apps, CMS
176) Drupal
177) Eclipse SWT
178) Emacs configs
179) Emoji and Dojo #
180) Facebook; Parse SDK #
181) ffmpeg
182) FLTK
183) Fonts
184) FPFA, Verilog
185) FreeRTOS (Embedded)
186) Glib
187) Ionic framework, Cordova
188) iOS Networking
189) iOS Objective-C API
190) iOS UI
191) Jasmine tests, JS exercises, exercise #
192) Java GUI
193) Java Native Interface
194) Java web servers
195) Javascript AJAX, Javascript DOM manipulation
196) Joomla
197) jQuery
198) jQuery Grid
199) Lex, Yacc compiler
200) libav / ffmpeg
201) Linear algebra libraries
202) Linux Kernel, Linux Wireless
203) Lodash
204) MFC Desktop Applications

F. Games

244) 3D graphics and Unity
245) Fantasy Creatures

* Repeating topic with different key words, see section VI.
# Dual topic, see section VI.
