Multiclass classification of GitHub repositories

Paul Anton
5anton@informatik.uni-hamburg.de

Erik Fließwasser
5fliessw@informatik.uni-hamburg.de

Thomas Hummel
5hummel@informatik.uni-hamburg.de

Waleed Mustafa
5mustafa@informatik.uni-hamburg.de

I. Introduction

This report contains details about our approach to solving the InformatiCup Challenge of 2017. Roughly, our workflow assumed the following phases: acquiring and processing repository information II, identifying and extracting valuable features III and classification IV. Our efforts resulted in two similarly performing final models that are able to classify repositories into one of the 7 given classes.

II. Data (pre)processing

A. Manually labeled / search results

The first step in acquiring class representative repositories was to use the search endpoint of the GitHub API. A number of search words per class were manually selected to search for repositories. Example words include: ”assignment”, ”course”, ”framework”.

Following the search for repositories, a manual validation was performed. A script to facilitate manual validation was developed. Given as input a list of repositories, the script opens the GitHub page for each repository and gives the annotator the options to label it as one of the categories.

One problem with this way of obtaining data is that of the sampling bias. As the number of search keywords was selected per class, this biased the returned repositories to only ones that contain those keywords. For example, a classifier that only searches for keywords should give a decent performance on our data set and might not generalize well.

We have used two ideas to counter this problem. The rationale behind both methods is that Bag Of Words features could select the search keywords as the most discriminating feature. However, while search keywords might be very powerful features, they will gain more power just because of sampling bias. A solution might be building a classifier that gives less importance to such features.

The first idea was to bootstrap more search keywords, that is, using the retrieved repositories, we built a Support Vector Machine (SVM) classifier using Bag of Words (BOW) on the readme data in a one vs all fashion. Then we used the weights of the SVM as an indicator of the word importance to a class classification. As more keywords are used this way for fetching more repositories, the data will be more general, resulting in lowering the importance of the first set of keywords.

The second idea was to randomly drop features so that the classifier learns to not depend entirely on them. Note that, here we randomly drop features from all the features not only those that are affected by the keywords. The reason of which is the intuitive correlations that might exist between the keywords features and other features in the system.

More information about the tuning of the classifiers can be found in section IV.

The distribution per class of our final data set is as follows:

- **DATA** - 163
- **DEV** - 344
- **DOCS** - 196
- **EDU** - 133
- **HW** - 222
- **WEB** - 242
- **OTHER** - 124

B. API data crunching

The main endpoint that we used to fetch repository information was the Repositories endpoint provided by the GitHub API. With the purpose of being able to later extract information from the already fetched data, we stored the results locally – one repository per file – in the same format they were returned i.e. JSON.

Some numerical features (further mentioned in section III) were extracted in one step by querying the corresponding URL in the stored repository object and performing a count on the response (while accounting for the pagination employed by the API). Other features (such as the ones based on the languages used in the repository) required further processing and subsequent queries to the API.

III. Features

In this section we discuss the features used in our solution.

A. Metadata

Basic data: A GitHub API call for the main repository endpoint delivers a list of basic metadata about it. The returned JSON object contains information like repository name, description, URLs linking to additional information as well as a couple of numerical values, e.g. size, forks, and

watchers of the repository. Retrieving further information about counts of issues, languages, branches and tags was based on the assumption that repositories can be separated according to patterns in the contribution, the level of interest of the community and also the usage of advanced git tools such as tagging.

Language data: While the basic data already covered the count of programming languages fetching data about the names of these languages needs further processing. Calling the repository's language endpoint returns a list of all used programming languages and their corresponding usage measured in lines of code. This quantity of data was normalized to proportions summing up to 1 and each programming language occurring in at least one repository was used as a feature.

B. Activity data

Commit interval data: The commit endpoint of the GitHub API gives information about date and time of every single commit of the repository. In a first step, we use this data to get values for the difference of days between the first and the last commit and the average of commits per day. By using the repository size in KB, the normalization is ensured and the correlation between the duration of a project and its size becomes a valid assumption, as shown by the performance of the BaseClassifier using these features in table II.

Commit data: For more detailed features we considered date time-stamps of all commits and computed proportions of commits of weekdays vs weekend days, working hours vs non-working hours and the average time distance between commits. In order to avoid the technical hiccup of fetching the entire commit history for a repository, we have experimented with two options:

1) limiting the number of commits to be fetched to the first 5000/ 10 000 and observing the classification performance
2) comparing the impact of the commit features on the classification performance with the one of the commit interval features; we obtained the insight that the least expensive to calculate commit interval features have roughly the same performance and thus both feature sets can be used interchangeably (table II).
   Apart from date timestamps, we consider that fetching the entire commit history is a great opportunity to extracting more useful features (commit messages for bag of words, analysis of author vs committer pattern etc.) but these have not been explored thoroughly.

C. Content data

Readme data: Readme files are the common way to describe the purpose and content of a repository on GitHub. Therefore, it contains rich information regarding the content which can be used for the task of classification. In contrast to all previous features, where we retrieved numerical values, Readme data consists of raw text data including digits, URLs, HTML tags and others. Thus, cleaning up the data is the first step before further processing it. We use BeatifulSoup[1] to get rid of HTML tags, to apply tokenizing which removes non-letter characters, to convert everything to lower case and to remove English stop words.

File and folder names: File and folder names are also very likely to carry important information about the repository. During manual labeling, we noticed that in most cases a look at the first level of the folder structure is enough to have a good clue about the repositories label. Folder and file names are treated as text, thus, before using them for classification, they have to undergo cleaning as well. In contrast to Readme data, folder and filenames do not tend to contain HTML tags, URLs and stop words. Instead, the data often consists of words containing "CamelCase" and underscore characters. Therefore, in both cases the words are split up, special characters are removed and everything is converted to lower case. Since numbers might be meaningful in cases where folders or files have the same name and are only distinguished by a given number those are replaced by a placeholder instead of removing them.

D. Keyword Spotting

During manually labeling of data, specific words, different than search keywords, have been found to repeat in the description (Readmes) and folder-/filenames for specific classes of repositories. The presence of specific keywords might allow to describe repositories and to separate them from each other. This again can be affected by our sampling bias but with the use of random selection of features can mitigate this problem.

1) Keywords: Keywords were separately defined for readmes and folder/file names, as descriptions and naming of folders/files are very different from each other. In the following, examples of used keywords are shown.

a) Description/Readmes:

- **DATA:** data, dataset, sample, set, database
- **DEV:** library, package, framework, module, app
- **DOCS:** documentation, collection, manuals, docs
- **EDU:** course, coursera, slide, lecture, week
- **HW:** homework, solution, deadline, problem
- **WEB:** web, website, homepage, javascript, template

b) Folder and file names:

- **DATA:** dataset, csv, pdf
- **DEV:** scripts, pom.xml, install, test, bin
- **DOCS:** doc
- **EDU:** course, slide, lecture, assignment
- **HW:** homework, hw0, task, lesson, week_
- **WEB:** website, css, img, images

Overall, 81 keywords were used, 51 for readmes and 30 for folder/file names. As the category of **OTHER** is not clearly

[1] https://www.crummy.com/software/BeautifulSoup/bs4/doc/

defined on its own and is rather defined as "being not one of the six main classes", it can not be represented by keywords directly. It can be, however, represented indirectly by having no specific patterns of keywords (e.g. having none of these), opposed to the other classes.

2) Feature extraction: The content of the readmes can be represented as a long string. In order to prepare the readmes for feature extraction, they were cleaned up by removing unnecessary stopwords, which reduces their representational size. The folder and file names of each repository consist of a string of file and folder names separated by spaces. For each repository, a feature vector is created by spotting occurrences of the predefined keywords in the readmes and the folder/file names. A feature vector for readmes is a binary vector of the length of the defined keyword list of readmes (length of 51), and a binary vector of length 30 for the folder/file names. The feature vector thereby has a value of 1 in the position of an occurring keyword, and zero otherwise. Afterward, the feature vectors for the readme and the folder/file names are combined to form a big feature vector of size 81. This feature vector is used as an input for a random forest classifier.

E. Bag of Words

Features extracted from readme files as well as from tree structure are text data. In order to deal with it and to extract information, the *Bag of Words* approach is chosen. The random forests handling them are further referred to as ReadmeClassifier and TreeClassifier. The difference between this and the keywords spotting features is that here we use all of the words and not only specific ones. Also, tree features account for the entire depth of a project, while the keyword spotting was only performed on the root folder.

IV. Classification

This section outlines some of the insights we gathered when performing classification on each category of features, as well as solutions to encountered obstacles.

For classification, there are many models and algorithms which make the choice of an algorithm a hard task. Among the famous methods are Neural Networks (Deep Learning), Support Vector Machines (SVM), Random Forests (RF), Logistic Regression (LR) and meta algorithms like boosting and bagging. While Neural Networks have lately received much attention in Machine Learning community, they usually require very large datasets. Since the collected data was not large enough, we resorted to the statistically more stable methods of SVM, RF, and LR. Random Forests are well known for their high performance and statistical stability due to the fact that they designed to reduce the variance of the model (they are a form of bagging). Support Vector Machines are also well known for their generalization capabilities. For those reasons, SVM and RF received more attention than other models in our experiments.

Different classification models were built using the different sources of data and were validated using cross validation.

Table I
BaseClassifier - RandomForest configuration

Parameters to optimize	Value range	Final value
n_estimators	500 - 10 000	5000
max_depth	5, 10, 20, 30, 50, 100	30
min_samples_split	1, 2, 5, 10	2
min_samples_leaf	1, 2, 5, 10	1

We noticed that different models give different errors on different classes, that is, every model is best for a specific class. That inspired an ensemble of trained models and/or features.

In the next subsections we describe our different models and their ensemble.

A. Random Forest

As noted in the data collection section, we needed to select random features to build different classifiers so that we mitigate the problem of sampling bias. In Random Forest, we get this behavior out of the box as different models are built using random subset of the features. Additionally, in most of our experiments, Random Forest always yielded the best results.

1) BaseClassifier: In our implementation, a *BaseClassifier* is represented by a random forest trained on one feature set at a time (language features, commit features, basic repo features, etc.). The hyper-parameters of the random forest were optimized by using grid search with cross-validation. Both the classifier and the tuning method are using the existing implementations provided by the scikit-learn package[2]. The values of the parameters are depicted in table I.

B. Ensemble

As noted above, different data sources performed differently on different classes. This calls out for an ensemble of them. We have tried different ways, among them, we report the two most prominent. First, since the best performing model of on all data was RF, we have all the features concatenated. In a first iteration, and RF is built which we call *SolidClassifier*.

Second, we merged the outputs of different classifiers, namely, the predicted probability of classes assignments. Details follow in the next subsections.

1) SolidClassifier: Since most of our best models for all the features are RF, it made sense to concatenate all the features and train one big RF. One initial attempt was to perform a basic merge of all the repo, commit and language features which already displayed an increase in performance. Consequently, adding the keyword spotting feature vectors (representing readmes and root folder and filenames) resulted in a yet better performance, as shown in table II. This so-called SolidClassifier performed better than all BaseClassi-

[2] http://scikit-learn.org/stable/

fiers in all the categories and became our first final solution. A more graphical representation of this is shown in figure 1.

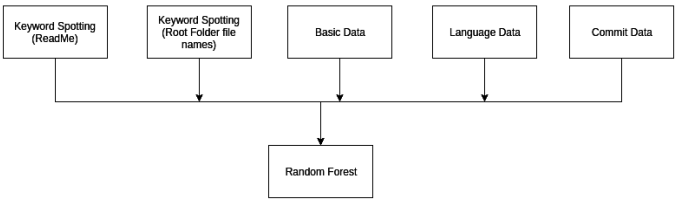

Figure 1. Concatenate all of the features and build a Random Forest out of all of them

2) ProbClassifier: In the second method of our ensemble experiment, we treated ReadmeClassifier and TreeClassifier as entities that sum the high-dimensional input features into a meaningful – with respect to classification task – lower dimensional feature vector. This lower dimensional feature vector is the predicted probability of the class assignment of aforementioned classifiers. Intuitively, these class probabilities represent the knowledge that they inferred from the training data to predict the class assignment. Different normalization and classification models were tried on different outputs of ReadmeClassifier and TrainClassifier. As table III shows, this ProbClassifier is entitled to become a competitor for our first solution (SolidClassifier). The resulted architecture is depicted in figure 2.

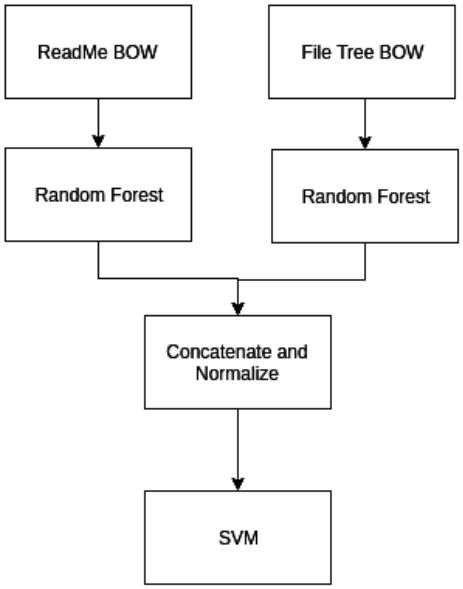

Figure 2. Ensemble of classifiers by classifying the probability output of base classifiers

V. Evaluation

The evaluation of our approach was done in two different ways. On one hand, the classification was tested based on the data which was collected for building up the model. Before training of the classifiers, this data was randomly split into training set (80 %) and test set (20 %). In order to ensure the stability of the classification, splitting and training were repeated with different random number generators. For each resulting test set, the F1 score is computed. The reported score , tables II and III is the mean of all computed F1 scores.

Furthermore, the challenge requires testing the prediction model on a specifically given test set. Since the target labels for this test set are not given the labeling had to be done by hand before using it to get a measurement of the quality of our classifier. It is important to note that those repositories were not part of our training procedure and not even used to validate or choose models or hyper parameters or to get insights from.

Table II
F1 scores for all four Base Classifiers, the Base Merge classifier, and the Solid Classifier on each single label and their average values. The highest scores of all are bold whereas the highest scores of the Basic Classifiers (without Base Merge and Solid) are marked by a star.

Classifier	DATA	DEV	DOCS	EDU	HW	WEB	OTH.	Avg
Basic	0.47*	0.66*	0.46*	0.35*	0.49*	0.53	0.13	0.44
Language	0.36	0.58	0.28	0.23	0.35	0.66*	0.12	0.37
Commit	0.11	0.36	0.37	0.08	0.49*	0.28	0.14*	0.26
Commit Int.	0.19	0.32	0.27	0.11	0.48	0.21	0.13	0.25
Base Merge	0.48	0.70	0.58	0.26	0.57	0.68	0.14	0.49
Solid	**0.64**	**0.74**	**0.73**	**0.54**	**0.73**	**0.77**	**0.19**	**0.62**

Table III
F1 scores for Readme Classifier, Tree Classifier, and Probabilities Classifier on each single label and their average values. The highest scores of all are marked bold

Classifier	DATA	DEV	DOCS	EDU	HW	WEB	OTH.	Avg
Readme	0.50	0.73	0.67	**0.62**	0.68	0.74	0.00	0.56
Tree	0.65	0.75	0.72	0.27	0.61	0.81	0.30	0.59
Prob.	**0.72**	**0.76**	**0.78**	0.45	**0.69**	**0.83**	**0.30**	**0.65**

VI. Discussion and Future Work

Probabilities classifier III seems to outperform Solid classifier II except for EDU and HW. This suggests that a combination of them might yield a better overall performance. Also, it is apparent that the performance on others is very low. This is the case because most repositories of the **OTHER** class are empty and therefore it is impossible to extract and calculate most of the features. This could be easily mitigated by handling the case of empty repositories by a rule-based method.

A semi-supervised method like recent upcoming Ladder Networks (special form of neural networks) might also be very useful, as we now acquired a decent amount of labeled data towards the end of the project. This will help working on larger unlabeled datasets.

Table IV
VALIDATION

Repo	Intuitive clf	SolidClassifier	ProbClassifier
ga-chicago/wdi5-homework	HW	1	1
Aggregates/MI_HW2	HW	1	1
datasciencelabs/2016	EDU	0	0
githubteacher/intro-november-2015	EDU	0	0
atom/atom	DEV	1	1
jmcglone/jmcglone.github.io	WEB	1	1
hpi-swt2-exercise/java-tdd-challenge	EDU	0	0
alphagov/performanceplatform-documentation	DOCS	1	1
harvesthq/how-to-walkabout	DATA	0	0
vhf/free-programming-books	EDU	0	0
d3/d3	DEV	1	1
carlosmn/CoMa-II	HW	1	1
git/git-scm.com	DEV	1	1
PowerDNS/pdns	DEV	1	1
cmrberry/cs6300-git-practice	HW	0	1
Sefaria/Sefaria-Project	DEV	1	1
mongodb/docs	DOCS	1	1
sindresorhus/eslint-config-xo	DEV	1	1
e-books/backbone.en.douceur	EDU	0	0
erikflowers/weather-icons	DEV	1	1
tensorflow/tensorflow	DEV	1	1
cs231n/cs231n.github.io	EDU	1	1
m2mtech/smashtag-2015	EDU	0	0
openaddresses/openaddresses	DATA	0	0
benbalter/congressional-districts	DATA	1	1
Chicago/food-inspections-evaluation	DATA	0	0
OpenInstitute/OpenDuka	DEV	1	1
torvalds/linux	DEV	0	0
bhuga/bhuga.net	EDU	0	0
macloo/just_enough_code	EDU	1	0
hughperkins/howto-jenkins-ssl	DEV	0	0

Table V
AVERAGE RECALL AND PRECISION MEASURES PER CLASS FOR THE TWO FINAL MODELS

Classifier	Performance	DATA	DEV	DOCS	EDU	HW	WEB
SolidClassifier	Recall	0.25	0.81	1	0.22	0.75	1
	Precision	0.33	0.75	0.5	1	0.37	1
ProbClassifier	Recall	0.25	0.81	1	0.11	1	1
	Precision	1	0.6	0.6	0.5	0.5	0.5

Figure 3.

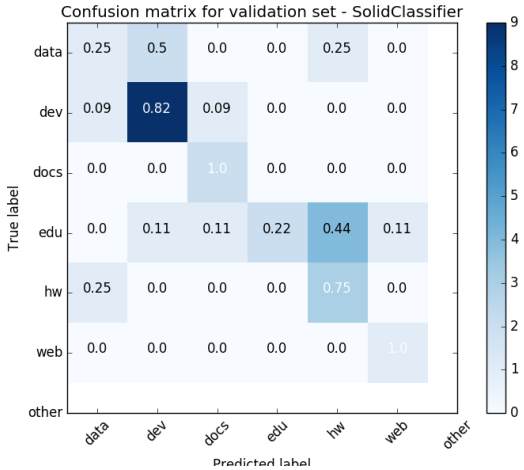

Figure 4.