Documentation: *GitHub Classifier* for the *InformatiCup 2017*

Andreas Grafberger, Michael Leimstädtner, Stefan Grafberger, Martin Keßler

February 1, 2017

Contents

1 First Insights

1.1 Confusion with class definitions

One of the biggest 'challenges' was the unclear class definitions. We spent a lot of time discussing how to classify different samples, there were many samples where different classes would have been a reasonable fit. We tried to extend the class definitions on our own while trying to avoid conflicts with the sample repos and the short definitions in the challenge descriptions. We decided we wanted to classify the samples in a logical, consistent and somewhat intuitive way, even though some things are debatable, e.g. WEB was defined as static personal websites and blogs. But when is a website non-static? As soon as there's some JavaScript or PHP involved? And is *personal websites only* a reasonable definition? Or shouldn't a small, static website of some sports team also be classified as a WEB sample? Because of that we expect the results of the different teams to be very diverse and hard to compare. What we want to emphasize in the beginning: we didn't take shortcuts and also deliberately chose to not make our lifes easy, e.g. it would have been possible to just put every website that's not a huge web app into WEB instead of trying to differentiate like the given class definitions suggest. Doing that would have led to a huge increase in the precision of our classifiers in the WEB class and other classes made us make similar decisions.

1.2 Intelligent Sample Collection

Probably the most important sources of information about a GitHub repository is the readme and the short description text. As we knew we had to use them, we knew to achieve really good results we would need an extreme amount of samples, as text is a very hard to use feature for a classifier. Using word counts as example we knew we would get an input vector with a dimension of several thousand. So for a classifier to actually learn the importance of each value of the input vector, we would probably need at least ten thousands samples, and that's not even counting in the majority class problem - repos belonging into the *DEV* category outnumber the repos of other categories by a large factor. As the repositories are really difficult to classify cleanly and consistently and because classification is really time consuming in this particular scenario, we set ourselves the goal to try to utilize several different techniques to achieve a good result with a much smaller amount of samples. The main method we used for that was **Active Learning**. We also tried different methods, like manually selecting repositories, (for that, it was really important to choose them as diverse as possible) of different subclasses of the different classes, as example the tutorial-repository subclass of the *EDU* class, and use them to get a jumpstart on the different categories. Later, when our classifier got better, we also tried to only classify samples where our best classifiers were confident that it was neither a *DEV* nor *OTHER* repo, as these classes are much less interesting then the other ones. By utilizing methods like these we are pretty happy with the performance we reached with only around 2000 samples.

1.3 We built ourselves a somewhat reusable tool to help us with machine learning tasks like the given one

Experimenting with different features, classifiers and parameters can take a huge amount of time. So we designed and implemented a tool that allowed us to use different classifiers like a black box, to save and load them and to test and analyse and compare their performance. To make experimenting with different features possible, we didn't only save the features we ended up using in the end in our database, but saved all the information we thought could be useful in our database, so each classifier could independently choose and process the features for itself.

1.4 Classification Strategy

Being inspired by machine learning competitions like *Kaggle* or the *Netflix Prize* we learned that almost every winning solution for most challenges there consists of combinations of various classifiers, also known as *Ensemble Learning*. So we knew that we would somehow end up incorporating this technique right from the start. Eventually it ended up increasing the quality of our classifications drastically thus confirming

our previous assumption. In addition to that we wanted to try several state of the art algorithms and neural net architectures like *Word2Vec* or *LSTMs*.

1.5 Our Strategy to get our train data

As we approached this problem from a machine learning perspective we knew from the start that our models require a large and diverse collection of manually classified examples. We immediately decided we needed a tool to help us building up a large collection of training samples. So, in order to speed up this process and make it as pleasant as possible, we first set up a website whose aim was to display all necessary information about a repository and to make it easy to classify the samples, the resulting data got saved to a database server. GitHub only grants a limited number of API-calls available in a short amount of time so early on we stored all hand-classified repositories with extracted features in a database to access them without restrictions. In this process we filtered out all information (features) we needed and also possibly lacked to do so confidently. Pretty soon we were surprised by how different our perspectives were in regards to which class to assign to many repositories. For some thoughts on this at this time see *Discussions/List of corner cases.md*. Following this we decided to abolish all samples we classified through weeks and start from the beginning. The danger this diversity proposed to our classification-results was unacceptable. So we worked out precise definitions of each class and listed vital edge cases. The focus hereby laid on understandable definitions, finally sacrificing better evaluation scores of our models since many important distinctions are very hard 'to get' for these models. After we learned from our mistakes we started to only classify in groups, later on handing this task over to only two team members and finally to only one person as it became too time-consuming. Finally this person is responsible for more than 3/4 of all samples we use now. This way we maximized constistency and efficiency without compiling a list of every single corner case to prevent ambiguity. Altough keeping this list updated would be nice to have, especially for outsiders, it has proven to be impossible to do so without sacrificing an immense amount of time (and therefore collecting only half the samples we have now). This problem arose with *Classification Ambiguities.md* and *List of corner cases.md* in the *Discussions* folder. Mistakes made by this one person couldn't be prevented but eliminating the possibility of confusing differences in proposed classifications was our priority. Using this method we finally classified over 2000 repositories by hand.

1.6 First Data Set Impressions

After building up our first set of training-data we were confronted with a serious problem: We encountered a so called Majority Class Problem. As GitHub is mainly used for software development projects the class **Development (DEV)** appeared way more frequently as any other class, taking up around 80% of all public repositories on GitHub. This resulted in our first classifiers assigning *DEV* to almost every repository as it was right doing so most of the time. Our first approach to this problem was to split up the prediction process into a first step where only the distinction between *DEV* and *NOT Dev* has to been made. If it was classified as *NOT Dev* we'd show the repository to a classifier which only knew how to classify such (see classification-skizze.png for more). We dropped this technique later on as we started to weight the importance of each sample during training with respect to the frequency of it's class.

Furthermore we incorporated the use of *Active Learning* in our training process (click here for more information). By this we were able to only present such repositories to us for manual classification which our classifiers were particularly unsure about. We used this approach to improve our classifiers more effectively by not hand-classifying redundant repositories. This was achieved by maintaining a large pool (35,000) of unlabeled repositories in our database. Two modes were implemented:

- *Stream based* One random repository from the pool is selected and shown to the classifiers. If a classifier is unsure about the class, the user (like an Oracle) is being questioned.
- *Pool based* In turns each classifier is shown a subset of these repositories and picks the one it's most uncertain about for further questioning.

To measure the uncertainty about a sample, various formulas are provided. Our experience with this method was very positive although it turned out to not solve the issue of training altogether. The repositories presented to the user were almost exclusively edge-cases. On the one hand that was beneficial

but in order for most of our classifiers to work correctly we needed a large amount of samples with clear and easy to interpret features to confirm assumptions about correlation between features. We partly tackled this problem by adjusting the parameters which determine when a classification is assumed to be confident/unsure.

1.7 Class Descriptions

As we didn't consider the class descriptions to be precise we added further explanations and also partially changed the pre-existing ones, later we decided we have too many new border cases all the time so a single team member took over all the classifying at some point to save the extreme amount of time we spent discussing samples and trying to keep our classifications consistent. (To examine our old extended class descriptions and explanation of edge cases see **Classification Ambiguities.md**)

2 Software Architecture

2.1 Motivation

As we already tried several different classification methods and spent a lot of time iterating and trying to understand how to use each other's code we decided we needed a tool that allowed us to use different classifiers as black box and to analyse their performance independent of the inner workings.

2.2 Goals

> The probably highest priority was to make it as easy as possible to try and compare different solutions.

We tried a great variety of machine learning models (as seen later in this document) and so we needed fast interchangeability and evaluation of models. Additionally we wanted to make the manual classification of new repositories and training of our models so efficient that we can build up a great training-corpus quickly. One step to achieve this was to integrate Active Learning in the process of selecting new training-samples.

For evaluation we predominantly used:

Confusion matrix

▼ Class. \ Reference ►	DEV	HW	EDU	DOCS	WEB	DATA	OTHER	Total	Precision
DEV	9	4	1	1	0	1	0	16	0.56
HW	0	2	0	0	0	0	0	2	1
EDU	1	1	2	0	0	1	0	5	0.4
DOCS	0	0	1	2	0	0	0	3	0.67
WEB	0	0	1	0	3	0	0	4	0.75
DATA	0	0	1	0	0	0	0	1	0
OTHER	0	0	0	0	0	0	0	0	0
Total	10	7	6	3	3	2	0	31	0
Recall	0.9	0.29	0.33	0.67	1	0	0	0	0.58

Figure 1: A sample confusion matrix generated for a classifier

2.3 Overview

The basic components of our Application are divided into a PHP-server with a dedicated database connection and a local Python-Bottle-server. The PHP-server manages the storage and access to all saved repositories and a large part of generating and pre-processing them. To be independent of operating systems and further complications we decided to present all graphic elements in the client browser using HTML. As most machine-learning algorithms are too complex to implement them by ourselves without spending an extraordinary time effort, we relied on pre-existing libraries. The easiest access to such is available with Python so we chose it as our main backend language.

Specific requirements and information about the installation can be found in the section: Installation Manual.

2.4 Python application

Design of framework: **Active Learning Framework Planning Phase.md**

Measure table

Measure	Result
Precision M	67.73 %
Recall M	53.93 %
Fscore M	64.43 %
Average Accuracy	79.6 %
Error Rate	62.24 %
Precision µ	58.97 %
Recall µ	58.97 %
Fscore µ	58.97 %

Figure 2: Different measures used to evaluate our classifiers

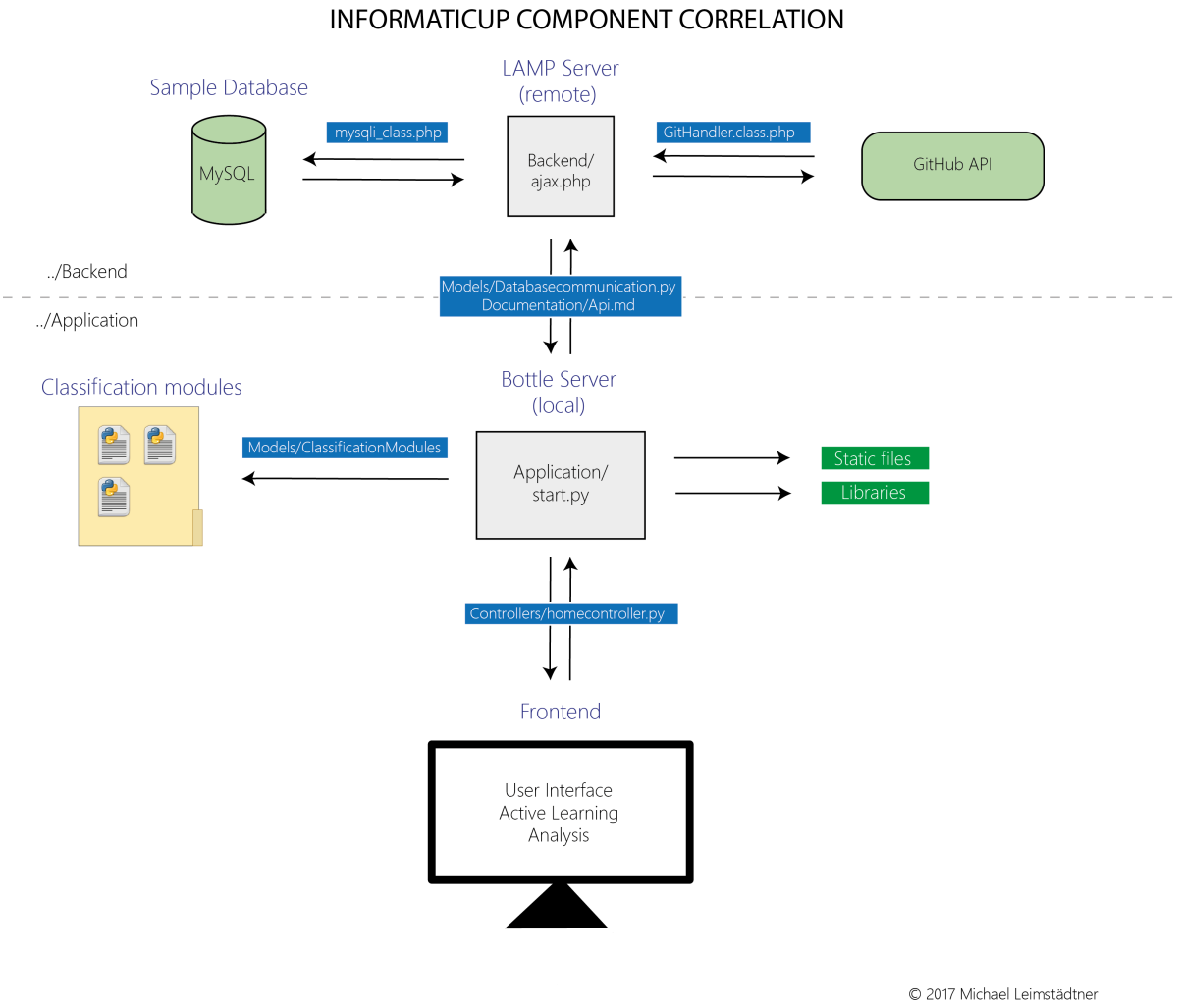

Figure 3: A basic overview on the component dependency

The main functionality is split up in 6 core components: First being *DatabaseCommunication.py* which handles all access to our database. This way we can - for example - download all our training data by just calling a single function. The returned samples/repositories provide all features available, it's now up to the classifier to filter out the features it needs. This happens in *FeatureProcessing.py*. The extraction and pre-processing of a sample's features happens here. Formatting and processing for I/O and presentation is done in *JSONCommunication.py*. Here we turn classifier results into confusion matrices and create XML-files for saved classifiers. The management of all classifiers currently presented is the goal of *ClassifierCollection.py*. At the start of the application we load all used classifiers in this collection and so if we want to train, test, save and generally use them, we just need to call the methods of this specific class. The interchangeability of these classifiers is guaranteed by creating the abstract class *ClassificationModules.py*. You will find more information about them later under *Prediction Model*. To evaluate our classifiers we use *classifierMeasures.py*. Given a confusion matrix it generates the desired evaluation scores (Fscore, Precision, ...).

2.5 User Interface

An explanation of our User-Interface with all functions can be found in the section **Frontend Manual**.

In order to keep the Frontend highly dynamic even with time-intensive user requests such as *train a classifier* we decided to use the JavaScript library VueJS in order to bind certain 'states' to the HTML DOM tree. This results in a quite large *index.html* file which represents every possible GUI states that are being styled by *overview.css*. Reactivity and observation is being brought to it by the file *frontend.js*, which also has a connection to the Python Controller *HomeController.py*. To sum up this nested relationship of different files: JavaScript tries to satisfy user wishes by updating the internal view state via Python services.

2.6 Webserver

The *Webserver* is basically a PHP server running the file contents of the folder **Backend**. Its main file *ajax.php* provides a bunch of services such as access to all collected data samples with additional filters and count-functionalities. In order to store and fetch those samples, it is connected to a MySQL database via *mysqli_class.php*. *Ajax.php* is also the only file that has access to the GitHub API via the controller *GitHandler.class.php* - this is why the server can be used to *mine* random repository samples and store extracted data to the database in order not to being limited to GitHub's API in production.

More detailed information about its services and more can be found in the section *Backend API*.

3 Installation

3.1 Requirements

- Python 2.7 64bit
- pip (module manager)
- Firefox 50
- Screen size >= (1280px x 600px)
- RAM > 4GB

3.2 Notes

This guide can be followed to get the frontend running on your local system. If you don't want to install one of the listed dependencies or want to emulate a preconfigured client image, please refer to the section *VirtualBox*. Even though running the client in an emulation should only be considered in exceptional cases.

Please make sure that the working / installation **directory** of Python and the g++ compiler do **not contain special characters** such as German umlauts. Keep in mind that installations with pip require **super user rights**.

3.3 Modules

Either downloaded and installed with the command pip install path/to/file.whl (on Windows) or installed with pip install xxx (on UNIX or OS x).

1. numpy + mkl
2. scipy
3. cherrypy on Windows or **paste** on UNIX or OS x for compatibility reasons

Installed with the command ***pip install xxx***

1. bottle
2. keras
3. sklearn
4. nltk
5. gensim
6. pattern
7. theano
8. demjson

3.4 Additional Dependencies

- Microsoft Visual C++ Compiler for Python 2.7
- A 64bit g++ compiler
- Keras has to be configured to use Theano (instead of Tensorflow). In UNIX systems, this can be changed at **/.keras/keras.json**. The same file can be found at **%userprofile%/.keras/keras.json** on Windows.

3.5 When everything is said and done

Now that every library and dependency is installed, you can open the **command shell** as a superuser in the folder *Application*. Type *python start.py* and wait until the GUI is being opened in a new Tab in Firefox (this will take a couple of minutes). Meanwhile, you can learn how to interact with the GUI in the section *Frontend Manual*. Note that you can read the docfiles in the User Interface as well.

If you want to test a file of repository links without a GUI (e.g. the Appendix B file) switch into the folder *Application* and place your input file. It must consist of a line separated list of GitHub repository URLs. Open the command shell and type *python predict.py ¡yourfile¿* and wait until the script has finished. The results will be saved at *classification_result.txt* in the same directory.

Note: If there are still packages missing that do not show up in the list, please install them via pip.

3.6 VirtualBox

Instead of installing everything from above, you can also use the VM image. A virtualisation software such as Oracle VirtualBox is needed in this case as well as a unzipping tool such as 7zip. Please be sure to reserve enough RAM for the emulation.

Download VM

Root login credentials are

Username: informaticup

Password: informaticup

Note that our GitHub repository is now public and visible for everyone.

Once Ubuntu has booted, change to the folder */GithubClassifier/* and replace it with the folder *Github-Classifier* of our submission. Open the folder and run *python Application/start.py* inside the terminal as root.

3.7 And what about the Backend?

We had many thoughts about running the backend locally as well or not. This would imply running two localhosts (Apache with PHP and Bottle with Python) synchronously as well as a local MySQL database - all that leads to many irrelevant and *OS-restricted* dependencies.

Therefore we decided to rely on the remote server even in the final submission, while providing you both the source code running on the machine (located in the submission folder */Backend/*) and full access to the server and its database.

Accessible via PuTTY on SSH port 7822

IP: 67.209.116.156

Username: <provided separately>

Password: <provided separately>

Backend location: /var/www/html

Database access: http://67.209.116.156/phpmyadmin/

Database username: <provided separately>

Database password: <provided separately>

If you still insist running the Backend locally, follow this guideline:

- Run **A**pache, **M**ySQL and **P**HP on your local machine. Note that the port of this localhost must be set different to 8080
- Installing PHPMyAdmin as well will make it easier to import database files

- Make sure Apache's DocumentRoot is set to the /Backend folder of our submission
- Create a new MySQL user and import the database files located in /Database/SQL dumps/u51019db1.sql
- Now switch to */Backend/mysqli_class.php* and update username, password, server name and database name
- Go to http://localhost:¡port¿ to see if everything is running
- The last step is to replace every occurrence of *67.209.116.156* to *localhost:<port>*. This should be limited to the files:

  /Application/Models/DatabaseCommunication.py

  /Application/Views/scripts/crawler_local.js
- Restart the application and enjoy :-)

4 Frontend Manual

The User Interface is designed to serve both for training and testing purposes. To keep them separated there are four different modes that can be selected in the header, each with its own behaviour.

1. Stream Based Active Learning
2. Pool Based Active Learning
3. Test All Classifiers
4. Handle User Input

4.1 Stream Based Active Learning

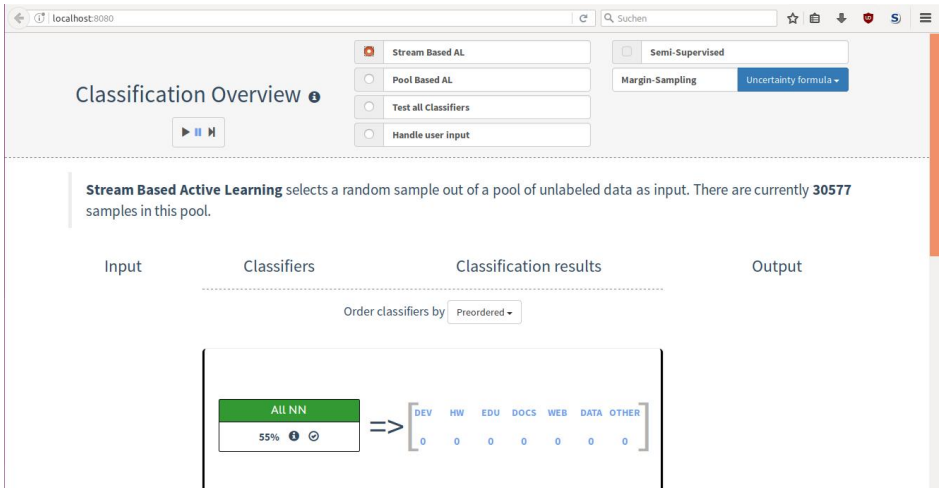

Figure 4: Stream Based Active Learning configuration

Selecting this or one of the other **AL** modes will display three buttons on the left side of the header. They are used to control the stream of input repositories. If you are done classifying repositories, either press the *pause* button or simply switch mode.

Stream Based Active Learning means that a random sample out of our pool of unlabeled data is being selected and classified by every classification module in the list. If one or more modules are uncertain about their output, you will be asked to classify the sample on your own. Otherwise the sample will be skipped. Manually classified samples will then be transferred to the labeled pool of training samples that are used to increase the classifier precision. Skipped samples could be transferred to the semi-supervised training pool, though we decided to disable this feature because of the current precision level.

In order to calculate their uncertainty, classifiers are using the selected uncertainty formula.

4.2 Pool Based Active Learning

This variation works similar to Stream Based AI: it picks in turn a single classifier (marked in blue) which determines the sample with the highest uncertainty out of a random partition of the unlabeled sample pool. Because of its nature of choosing the most uncertain sample, user classification will always be asked for the displayed sample.

4.3 Test All Classifiers

The testing mode has been designed to manage and analyse classification modules. This includes operations and visualisations regarding their train state, save files and measurable outcome.

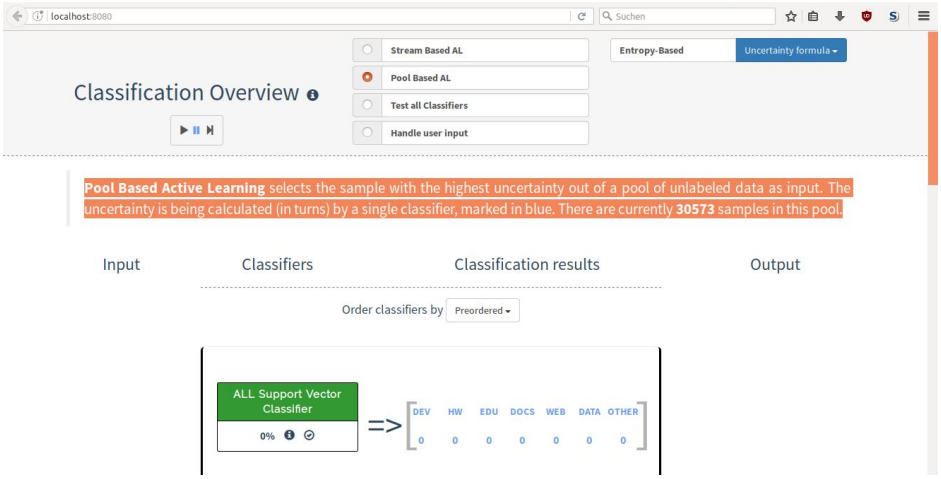

Figure 5: Pool Based Active Learning configuration

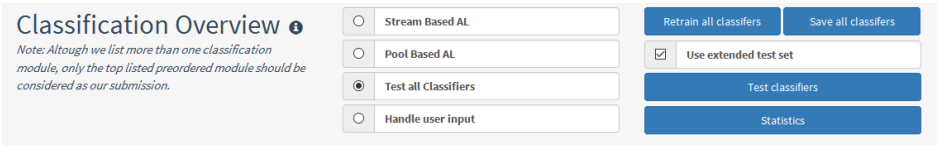

Figure 6: The heading section of the test mode

Retrain all classifiers will reset every (or every untrained) classifier and trains them on the current train set. Note that this will take a lot of time for modules that use high-dimensional features. *Save all classifiers* saves a copy of every module state to the disk.

If *Use extended test set* is disabled, test results are based on the Appendix B, which has been pre-classified by our own classification criteria. When enabled, a more representative test set of about 300 samples is being used to calculate test results.

The button *Test classifiers* will bring the test results up to date if the checkbox above has changed. Note that you don't need to test classification modules manually in any other case.

In order to get a rough idea about the train/test set sample class distribution and per-table or rather per-attribute statistics, just hit the button *Statistics*.

4.4 Handle User Input

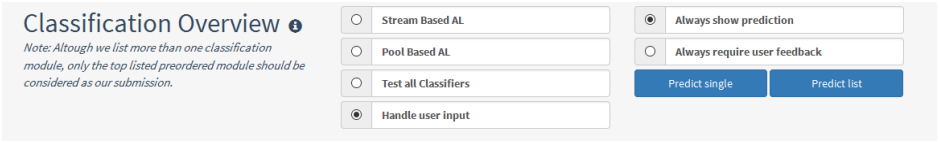

Figure 7: Possible options for the mode ''Handle User Input"

At times it might be interesting to test the classification modules on one or more non-random self-chosen repositories. If you are interested in the classification only, select the radio button *Always show prediction*. If you want to train the classifiers on your manual classification, chose *Always require user feedback*.

Clicking on *Predict single* will ask you to insert the link of a repository. It will then fetch information about that given repository from the server and calculate classifier results that are being shown to the user. Even entire lists of repository URLs can be processed by clicking the button *Predict list*.

4.5 Main Sections

The page is divided into a header, footer and main section. Depending on the actions performed in the control header, the main section (which always consists of an input, the classification modules and an output) changes its scope and contents.

4.5.1 Input

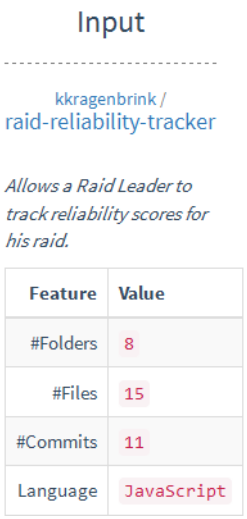

Figure 8: The input layout of a random repository

If any **A**ctive **L**earning or user input mode has been selected, the input will show a brief overview about the currently processed sample. Note that the attributes displayed to not comply with the features used by our classifiers.

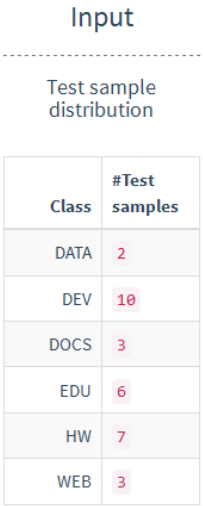

Figure 9: If the test mode is selected, the input section will display a sample distribution of the test test

In the *test mode*, a class distribution of the selected test set will be displayed. The test set can be changed inside the header section.

4.5.2 Classifiers and Classification Results

This section is basically an ordered list of classification modules (defined in */Application/Modules/ClassificationModules*). The green little box consists of the module name, the value of the selected measure, and two actions:

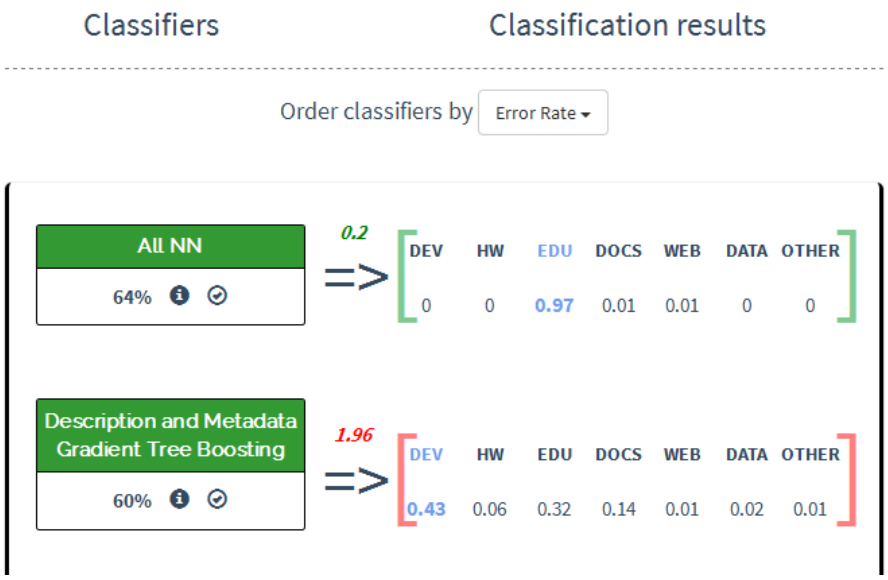

Figure 10: The central section shows several information about each classifiers results and performance

- *Show details* will open a wrapper containing deep insights in that classifier
- *Disable classifier* prevents this classifier to be asked for an output

Next to the green box, an arrow => points to a matrix of float values. Those values represent either the precision per class (if you are inside the *test* mode) or the output probability the corresponding module would label the input sample as part of the given class. Green or red values over the arrow => are pointing out whether the classifier is sure about its prediction or not.

The *detailed page* comes along with a few additional actions and analysis information:

- The **Performance Graph** is visualising the precision per class in a typical radar chart - different versions are overlapping to spot precision changes.
- New **versions** of that classifier can be created by hitting the button *Save current image*. To load an old image, chose the proper version and click *Load version*. The old version will be retested instantly on the selected test set.
- A **confusion matrix** lists precision and recall in absolute and normalized values
- The **measure table** lists the outcome values of this classifier for every predefined measure.

4.5.3 Output

Once again, the output changes whether or not the *test mode* is being selected. If it's not, the output gives you a brief overview about how the top-ranked classifier labels the selected input sample along with a list of how other modules would have guessed. Depending on your settings, an orange button labeled with **'?'** shows up if one or more classifiers are unsure about their output. Clicking on that button opens a new tab or window with more detailed repository information and the possibility to assign your personal classification label to that repository.

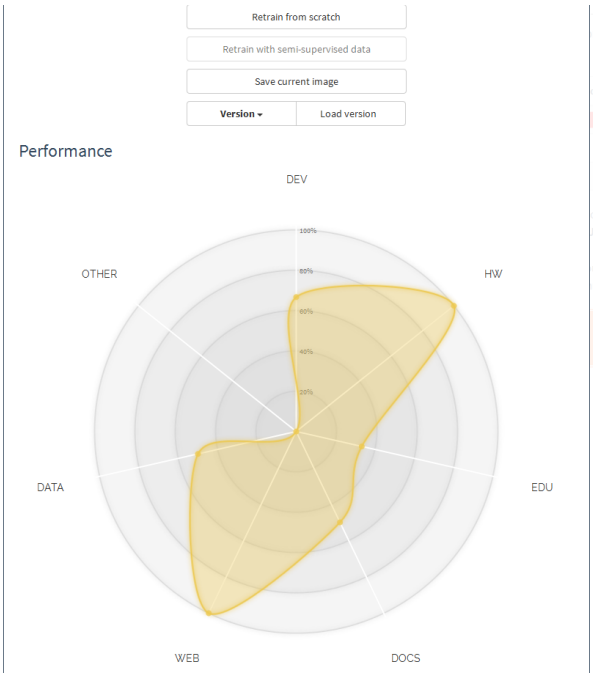

Figure 11: More information about a specific classifier can be obtained in the detailed page

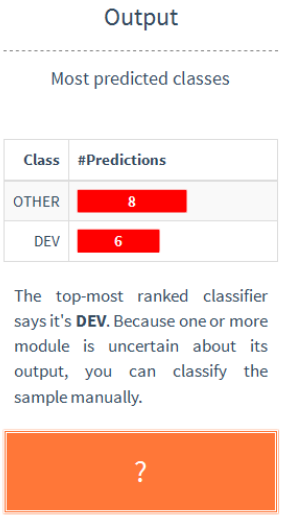

Figure 12: The output section will show the predicted class and will ask the user for manual adjustment if there is any uncertainty

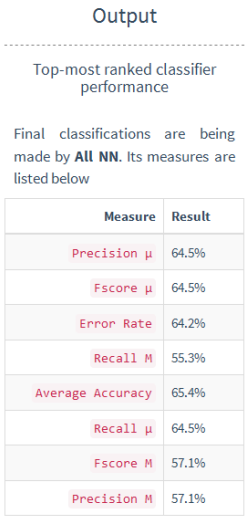

Figure 13: If the test mode is selected, it will show several measure results instead

The *test* output on the hand is non-interactive, its purpose is to present a summary about the top most listed classifier. Note that even if this may change depending on the measure selected in the central section, only the classifier selected in *Preordered* should be considered as our submission classifier.

4.6 The Footer

You can find a link to the competition website at the very bottom of the GUI along with the names of every team member. Clicking on one of us will display a wrapper containing some information about him.

5 Backend API

This file describes the services provided by /Backend/ajax.php

5.1 Base url

Queries with the need of database information are always sent via **GET** to the backend host, which is by default **http://67.209.116.156/ajax.php** (a small but stable database server). Services can be accessed with the attribute **key**, results can be refined using the additional attribute **filter**, **limit**. **selector** and **table**. The server response consists of a JSON object containing a Boolean **success** which determines the execution state and **data**, the query result.

The backend host can be switched if every occurrence of the IP written above is being swapped, but database integrity must be guaranteed.

Sample-Query: http://67.209.116.156/ajax.php?key=api:all&table=train&limit=5

5.2 Base Services

- ?key=api:all

  Returns all rows of the table **table**. (Attributes: **table**, **limit** (opt.), **filter** (opt.), **selector** (opt.))

- ?key=api:train

  Returns all rows of the table train. (Attributes: **limit**(opt.), **filter** (opt.), **selector** (opt.))

Similar calls are possible for **api:train**, **api:test**, **api:unlabeled**, **api:to_classify**, **api:semi_supervised**, **api:standard_train_samples** and **api:standard_test_samples**.

5.3 Additional Services

Attributes and its possibly states are being described in the following section.

- ?key=api:single

  Returns a random sample of the given **table**. (Attributes: **table**, **selector** (opt.))

- ?key=api:equal

  Returns an equal amount of samples based on the class count of the given table. (Attributes: **table**)

- ?key=api:class

  Returns all samples of the given class **name**. (Attributes: **table**, **name**, **selector** (opt.)).

  Application: **/?key=api:class&name=**< *classname* >

- ?key=api:count

  Returns the amount of rows affected by **table** and **filter**. (Attributes: **table**, **filter**)

- ?key=api:class-count

  Combination of the two above. Returns a class-based row count. (Attributes: **table**, **filter**)

- ?key=api:tagger-class-count

  Returns the class-based row count, limited to the given **tagger**. Used to see how many samples have been classified by a single person. (Attributes: **table**, **tagger**)

- ?key=api:move

  Moves a repository, taken from the pool **from_table** to the pool **to_table**. If **label** is set, the label of the sample will be changed. The attribute **api-url** must be passed as an identifier. (Attributes: **from_table**, **to_table**, **api_url**, **label** (opt.))

Table 1: List of database tables

Table Name
unlabeled
train
test
to_classify
standard_train_samples
standard_test_samples

5.4 Services dependent on the GitHub API

- ?key=api:generate_sample_url

  Generates and returns the API-url of a **random** GitHub repository.

  Attributes **client_id** and **client_secret** can (but mustn't) be passed in order to be used by the server. If they aren't, hardcoded credentials will be used with their limitation of 5000 API calls per minute.

- ?key=api:generate_sample

  This service depends on the parameter **api_url**. Feature extraction and accordingly data dumping to the database is being made for the given repository. The unlabeled row is being saved and returned to the client. If the parameter **api-url** is empty, a random sample is being generated. The attribute **class** (opt.) leads to an instant classification of the sample, if empty the class $UNLABELED$ is being used.

  Attributes **client_id** and **client_secret** can (but mustn't) be passed in order to be used by the server.

5.5 Attributes

The most essential attribute is **table**, its value defines the database table on which the query is being executed. Possible values are listed in table 1.

To keep queries efficient, the attribute **selector** determines which columns should be returned. If empty, ***** will be used.

Example: **selector=class, api_url**

SQL equivalent: **SELECT class, api_url FROM ...**

The parameter **limit** defines the maximum amount of result data samples, which are being chosen randomly.

The attribute **filter** can be set to (if supported by the service) a **base 64 encoded** array structured like [**attr1=val1,..**]. Operators available are =, <, <=, > and >=. Pairs of attribute specific filters can be separated either with **,**, which results in an **AND** conjunction or with | which forms an **OR** disjunction. (In our case, OR binds stronger (!) than AND).

Example

$btoa(\text{"}class = DEV|HW, stars > 3\text{"})$;

generates a possible **filter** value (in JS). The SQL equivalent is:

$SELECT...WHERE(class = \text{`}DEV'OR class =' HW')AND(stars >' 3')$

Possible filter attributes (Or: the general table structure):

Table 2: List of table attributes that can be filtered

Name	**Type**	**Description**
api_calls	Integer	Number of calls needed to gather the data
api_url	String	The Git-API url for this repo
author	String	Repository author name
avg_commit_length	Integer	Average commit message length
branch_count	Integer	
class	String	The label given by our classifier
commit_count	Integer	
commit_interval_avg	Integer	Average #days between two commits
commit_interval_max	Integer	Maximum #days between two commits
contributors_count	Integer	
description	String	
files	String Array, separator '	Files of the first layer
file_count	Integer	
folders	String Array, separator '	Folders of the first layer
folder_count	Integer	
forks	Integer	
hasDownloads	Boolean	Is the repo downloadable
hasWiki	Boolean	
id	Integer	Internal ID
isFork	Boolean	
open_issues_count	Integer	
language_main	String	The most used language
language_array	String Array, separator '	A list of used languages
name	String	The repository title
readme	String	
stars	Integer	
treeArray	String Array, separator '	A list of folder paths present in this repository
treeDepth	Integer	Maximum folder depth
url	String	
watches	Integer	

6 Features and Prediction Model

6.1 Features

6.2 Implementation Details

All classifiers have access to the features of a repository by using the functions provided in *FeatureProcessing.py*. They can request the features for a repository by calling the specific functions like *getMetadataVector* or *getReadme*.

6.3 Feature Development

To create the optimal feature vector every team member compiled a list of possible features (see Discussion/Feature Vector Ideas). We then discussed every proposal and added further ones. It's notable that the features considered most important by us at first where almost exclusively text based. Readme, description, folder names, file names, author name and more. Unfortunately we had to limit the access of our models to the folder and filenames of only the first layer in a repository's folder-structure. This was due to the previously mention API-Call limit. In these discussion we discovered many features we first neglected: used programming languages (with a possible emphasis on the main language), depth of the folder-structure, commit count, average commit-length, branch count, whether a download of the repository is allowed, folder-count, number of files and more. Later we additionally implemented the average Levenshtein distance between folder and filenames. This was done due to a lack of any feature that can give us valuable information despite the fact that many DOCS or HW folder-/filenames are often similar.

6.4 Text

- **Frequency-based methods:** We count the frequency of specific tokens or words in our documents and therefore encode the text in a sparse vector with each element representing how often one specific word/token occurs (large number = high frequency). Using this approach we had to consider how long this vector may be in order to be as efficient as possible. We only count the frequency of the x most frequent terms (excluding stop-words). While short vectors (and therefore less words we can keep track of) allow more robust classification results for our classifiers, we may lose important information that could make important distinctions (such as HW vs EDU) impossible. Having this problem in mind we also used a trick to reduce the necessary dimensionality a lot by not encoding each word but the word stem ('library' and 'libraries' are bot represented by '*librari*'). The resulting number of necessary words/tokens turned out to differ from the text we encoded. We use smaller numbers (2000) for repository-descriptions and even less (200) for folder-/filenames. The readme turned out to need a lot more (6000). This all was implemented using the *Tfidf-Vectorizer* from the sklearn-package.

- **Word embedding:** An alternative approach is to not represent a document as a vector accounting for all used words but to represent each word as vector which holds information about the context of it. So we end up with a matrix where each row stands for such a word representation. This embedding is learned through algorithms like presented in this paper. A so called *Word2Vec* model pre-trained on Google-News articles which has a vocabulary size of 3 million distinct words is being used here for. Each word-vector is fed into a recurrent neural network (explained later) one after another.

- **Character by character:** But instead of learning such a complex representation for each word we can just directly feed a text character-wise into such a network. This method came in handy when trying to classify depending on features like the repository-name. In many cases the name was too specific and complex to have ever appeared before and so no vector-representation is available with previous methods.

- **Edit distance:** When trying to classify repositories that belong to Homework, Documents or Education it's important to know how similar the names of files or folders are. Such repositories often

contain folders like *Week 1, Week 2, Week 3, ...* So to hand that information directly to our classifiers we measured the average *Levenshtein distance* of all files and folders.

6.5 Summary of used text-features

- **Repository-name**
- **Author-name**
- **Description**
- **Readme**
- **Filenames**
- **Foldernames**
- **Average Levenshtein distance of filenames**
- **Average Levenshtein distance of foldernames**

6.6 Numerical features, Metadata

- **hasDownload**: Boolean value if repository can be downloaded directly.
- **watches**: Count of people who follow latest news in respect to new pull requests and issues that are created.
- **folder_count**: Count of folders is currently limited due to limited API-calls available.
- **treeDepth**: Depth of folder-structure. Also capped at the moment due to limited API-calls.
- **stars**: Count of people who use this notification system.
- **branch_count**: Count of branches currently in use.
- **forks**: Count of created forks.
- **commit_interval_avg**: Average interval in which commits where created.
- **Commit count**: Number of commits since creation of repository.
- **contributors_count**: Count of contributors to this repository.
- **open_issues_count**: Count of open issue.
- **avg_commit_length**: Average text-length of commit messages.
- **hasWiki**: Boolean value if wiki is available.
- **file_count**: Count of files in first layer of folder-structure (Where readme is usually located).
- **commit_interval_max**: Longest interval between commits.
- **isFork**: Boolean value if repository is fork of another.
- **ReadmeLength**: Count of characters in readme.
- **verwendete Sprachen**: Used programming languages (represented as vector with each column representing one language. 0.5 if language is being used, 1.0 if it's the repository's main language).

6.7 Dismissed features

- **Commit messages:** We considered them to not obtain enough valuable information to sacrifice both the increase in input-dimension for the classifiers and necessary API-calls.

6.8 Possible features for the future

- **Count of filenames with min. Lev-distance**: Could be more informative for classes like *HW* or *DOCS* than the average Levenshtein distance and might be implemented in the future.
- **Document vector:** A possible approach similar to word embedding is often referred to as *Doc2Vec*, presented in this paper. While we weren't able to test this approach yet we're excited to see how it will compete against our current methods.

6.9 Prediction Model

6.10 Implementation Details

We wrote a lot about the desired interchangeability and modularity of our models in order to train, evaluate and compare them as efficient as possible. This was achieved through the following design: Classifiers are represented by an abstract class called **ClassificationModule** (*ClassificationModule.py*), Ensemble Classifiers are classes which inherit the class **EnsembleClassifier** (*EnsembleCLassifier.py*) which just adds a small amount of functionality to the ClassificationModule. Through that we got a unified interface for our classifiers and each Model is represented by a class which just has to implement the abstract methods from ClassificationModule like *train* or *predictLabelAndProbability*.

6.11 Used models

We hereby present all models used in the process, to look at our first tests see the **Playground** folder.

- **Neural Networks** are without any doubt the most hyped up machine learning models since many years. Mastering many difficult challenges like finding the right category out of thousands for images at human like performance. So we tried the most popular architectures to make use of this immense capability. The first architecture we tried was a standard **feedforward** network. Depending on the complexity of the feature-space we mostly used 2 or less layers, also trying 3 layers for frequency based feature-spaces. When using more layers we quickly saw an immense loss in *generalization* and the network *overfitted* on the training data (meaning that it only memorized repositories and their right class instead of actually learning how to interpret the given features). As activation function we favored the newer *ReLU* or *Noisy ReLU* over the previously used *tanh* and *sigmoid* functions for various reasons. As our *optimizer* we chose *ADAM* for this network-type. Having these things in mind our networks performed pretty well, always being among the best classifiers. The above mentioned **LSTM** networks came in handy when classifying based on a sequence of inputs (like characters or word-embedding). By feeding their last activation to themselves, these neurons "remember" their previous inputs (like the last characters forming a sentence). A helpful explanation can be found here. For reasons like improving training time *Stochastic gradient descent* was used for training here. For the last layer of all networks consisted of a fully connected layer with an applied *softmax* function to output actual predictions. To implement them we used the popular hrefhttps://keras.io/Keras library running on top of Theano.
- **SVMs (Support Vector Machines)** were an obvious choice, having proven to be a robust method for both regression and classification problems. After only a few tries we quickly decided to use the popular non-linear *RBF-kernel* over the linear version. Using that we had to find the best combination of the C and gamma values. We used them for all kinds of features, each time they've proven to be a serious competitor to our *neural networks*. The used library was Scikit-learn.
- We considered **Naive Bayes** classifiers to be very promising for text classification as they are in many cases. But even after trying out many *event models* the results were rarely satisfying. After

trying different combinations of lengths of the Frequency based features we omitted their further use. Same for our Nearest Neighbors classifiers, even though we also tried non-text features with them. Both are contained in the Scikit-learn library.

- **Ensemble methods** combine the capabilities of various classifiers to make an even more powerful prediction. The methods we used were incredibly powerful like the **Random Forests** or **Gradient Tree Boosting**. **AdaBoost** turned out to depend strongly on the used base-classifiers but after some tries our other methods outperformed them most of the time so we omitted further use. All these were already implemented in the Scikit-learn library. Parameter tuning mostly depended on adjusting the number of base-classifiers. They were not only often times the most robust classifiers but also outperformed *neural nets* or *SVMs* in many cases. After reading articles like this we began creating our final classifier used for the competition. **Stacking** is a general term for combining only the predictions of other learning algorithms also often referred to as **Stacked Generalization**. Among the great variety of methods we selected, implemented and finally compared three different methods: One was to simply let the classifiers vote democratically by simply taking the **average** probability per class. This made our classification way more robust and delivered an incredibly satisfying result. The second method is the probably most famous one: We trained a **linear classifier** to let the classifiers vote on the class but not without assigning a weight to the predictions of each classifier. This way we wanted to enable the classifiers to mainly decide on the classes they were best at. This was implemented using sklearn's **Linear Regression** class. But the third was the one we actually used for our final model. Instead of just letting the meta-classifier learn, which base-classifier delivered the best results for each class we supplied it with additional information about the repository. This information consisted of a collection of hand-selected features: the count of folders, files and commits, the edit distance among the folder and filenames, the length of readme, the average length of each commit and finally the depth of the folder structure. The reasoning behind this all was the following: The base-classifiers were not only different in terms of machine learning models but were also trained on different features. We use on Neural Net (LSTM) which only has knowledge of the repository-name while one Support Vector Machines has knowledge of all features except that name and so on. So the quality of each classifiers prediction varies from repository to repository. We fed all this data, the predictions of our base-classifiers and in addition this subset of meta-features, in a neural network with one hidden layer. This network is now able to not only tell which sub-classifier generally is most reliable when it comes to one class but knows which classifier might be most reliable specifically for the current repository. When for example the readme-length is very small or it's even empty the classifiers depending on the readme will not produce reliable predictions. But now it's able to recognize that and watch out specifically what other classifiers like the repo-name LSTM predict. This method allowed us to produce even more robust predictions, outperforming all previously mentioned ones.

Table 3: Appendix B Predictions

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

7 Validation

For our own model evaluation we used an extended test-set of approx. 40 repositories per class. In addition to the ones in Appendix B we used respositories committed by another participating team. We reclassified many of them to match our own classification guidelines.

> "Apply your classifier on the repositories included in Appendix B . You can find this file on https://github.com/InformatiCup/InformatiCup2017 as well. Create a Boolean matrix where you compare the results where you compare the results of your classifier and your intuitive classification (if your intuitive classification matches the output of your program, the element in the matrix will result to true, otherwise to false). Compute the recall per category- the number of repositories intuitively placed within a category in the set of repositories that got placed in the same category by your classifier. Compute the precision per category- the number of repositories per category where the results determined by your automatic classifier matched your intuitive classification."

Confusion matrix

▼ Class. \ Reference ▶	DEV	HW	EDU	DOCS	WEB	DATA	OTHER	Total	Precision
DEV	9	5	2	1	1	1	0	19	0.47
HW	0	1	0	0	0	0	0	1	1
EDU	1	0	5	0	0	0	0	6	0.83
DOCS	0	0	0	1	0	1	0	2	0.5
WEB	0	0	0	1	2	0	0	3	0.67
DATA	0	0	0	0	0	0	0	0	0
OTHER	0	0	0	0	0	0	0	0	0
Total	10	6	7	3	3	2	0	31	0
Recall	0.9	0.17	0.71	0.33	0.67	0	0	0	0.58

Figure 14

7.1 Example Repositories

Please document three repositories where you assume that your application will yield better results as compared to the results of other teams."

- Homework **HW** As this repository does neither contain a description nor a readme, there are only a handful of features that could lead to a correct classification: Filenames, Foldernames, repository name and somehow maybe the metadata. But most classifiers which are also trained on filenames and foldernames would classify this repository as DEV. The most certain and also correct classifier was trained on the repository name, made possible by using a Recurrent Neural Net with LSTM-layers. As the stacking model has knowledge whether a readme or description exists it apparently trusts the prediction of this classifier and also few classifiers trained on metadata and not those trained on the readme or description. And therefore classifying it correctly.

- Demo **EDU** At a first glance it seems pretty obvious that *WEB* is the right category. Among all these chinese characters 99% of the words are Web-related. "Web", "HTML", "CSS", "Ajax" and more. But even tough some classifiers fall into this trap, the ensemble-model is very confident that the focus lays on a demonstration and knowledge transfer, as it states only 2 times on the site with the word "Demo". Therefore **EDU** is the right choice.

- Data **EDU** *Data* was one of the classes we struggled with the most. It's so rare that we hardly found sufficient samples first. But after incorporating Active Learning and also looking up some samples manually the classification of such repositories is very robust. To get this right our classifiers had to look out for triggers like *Data* or *Datasets* and react very strongly to them. But anyway our ensemble-classifier still figured out that this repository is not **DATA**. The focus lays on knowledge transfer as it gets more and more clear by reading the readme. Even though for example the LSTM-Net is 100% sure the repository is **DATA** the ensemble learner is being supplied enough meta information to recognize it has to rely on other classifiers here.

7.2 Precision vs Yield/ Recall

We consider a higher precision to be more relevant than a high recall per class. When thinking about a user, looking for repositories of a specific class on GitHub, it appears way more desirable if the repositories proposed by us are actually of the right class. Making sure every *DEV*-repository is presented to the user, potentially including wrongly as *DEV* labeled repositories didn't seem like the right approach. As the precision per class goes up during training, the recall will do so automatically as well. Emphasising

precision while not neglecting recall completely we agreed upon Fscore as our metric. With it it's possible to combine both values into one while being able to favour one over another.

8 Conclusion

8.1 Hard Problem

Text Classification isn't easy and the different classes are extremely hard to distinguish. In fact so hard that even as humans we had problems agreeing on the class labels after classifying hundreds and thousands of repositories before. So, if we humans disagree on every second sample, we probably can't expect our classifiers to perform a lot better than that, especially considering the extreme amount of information per repo.

8.2 Features

During our work on this project we had to make serveral sacrifices. When we discussed which features were important to us as humans while classifying we weren't limited by API-Calls or processing power. But these limitations turned out to be a crucial part of our work. When training on over 2000 repositories it's infeasible to measure edit distances for every filename-combination just as it is utopic to compare the content of every file in a repository. So it's been an exciting process finding little tricks and optimizations to get the most out of this situation. Even though we had to omit the use of some features (Word2Vec-embeddings, file-content, ...) and weren't able to try all possible feature-combinations due to limited time we're pretty happy and confident about our results and feel like this experience will enable us to approach similar problems way more efficiently in the future.

8.3 Our classifiers clearly reached some sort of ceiling

No classifier performed much better than 60% precision M, no matter how long we tried to tweak its parameters. But we reached a value around these 60% with almost every other classifiers, so it appears that what we experienced was some kind of limit how much can possibly be achieved with our features and amount/cleanness of our train data. With an Ensemble classifier, just like in competitions like Kaggle, we somewhat managed to gain a few crucial extra percent points.

8.4 Interpretation of our results

71% precision (M) sounds like a pretty good number when we humans disagreed on what felt like every second sample. While working on the given challenge, we learned a lot about machine learning, although unfortunately we couldn't utilize every method we learnt about during the work on the project. However, we developed a framework that we will very likely use again the next time we get to work on a machine learning project. After building our application, the testing and comparison of different methods and parameters was incredibly comfortable.