



School of Engineering

InIT Institute of Applied
Information Technology

Project Work Computer Science

A Framework to Optimize Machine Learning Algorithms for Text Classifica- tion through an Intuitive User Interface

Authors

Linus Metzler
Nadina Siddiqui

Main supervisor

Mark Cieliebak

Sub supervisor

Don Tuggener

Date

20.12.2017

Erklärung betreffend das selbständige Verfassen einer Projektarbeit an der School of Engineering

Mit der Abgabe dieser Projektarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat. (Bei Gruppenarbeiten gelten die Leistungen der übrigen Gruppenmitglieder nicht als fremde Hilfe.)

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Projektarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Bei Verfehlungen aller Art treten die Paragraphen 39 und 40 (Unredlichkeit und Verfahren bei Unredlichkeit) der ZHAW Prüfungsordnung sowie die Bestimmungen der Disziplinarmaßnahmen der Hochschulordnung in Kraft.

Ort, Datum:

Unterschriften:

Winterthur, 18.12.2012



Bidelighi

Das Original dieses Formulars ist bei der ZHAW-Version aller abgegebenen Projektarbeiten zu Beginn der Dokumentation nach dem Titelblatt mit Original-Unterschriften und -Datum (keine Kopie) einzufügen.

Abstract

We developed a tool which allows a non-expert user to perform text classification with a selection of machine and deep learning algorithms and compare these algorithms in a straightforward way.

This application consists of a graphical user interface, the implementation of the underlying algorithms, and a server which connects the other two components. The primary focus was on connecting the different tools involved and then presenting the calculations of the algorithms in a visually pleasing fashion. Consequently, we did not optimize the algorithms and instead used readily-available libraries for the underlying implementation. Some of these libraries, however, required special attention as to make their output usable for further, automatic processing.

While a lot of machine and deep learning algorithms have been developed in recent years, they often require an experienced programmer to be able to use these algorithms. Additionally, the setup and configuration of these algorithms is often based on intuition and comparing different algorithms on the same input is rather involved. In the first part we will discuss prior work in this area.

Next, we have a look at our application and its features and capabilities. We will explain how the different components interact with each other and what challenges we faced. This chapter will be augmented with visual examples.

Since we aimed to make text classification simpler to use, especially when one is not sure which algorithm is suited best for the task at hand, we decided to conduct a user test which involved two people who were unfamiliar with the project. The results of this test will be presented in the following section.

As the application should be useful for practical use, we dedicate a chapter to possible use cases and look at both the immediate use case and envision what it could achieve in the field of text classification in general.

Lastly, we take a step back and discuss the results we have achieved in this project. As this project was designed to lay the foundation for future work, we will explore possible extensions and features that could be implemented in future projects.

Zusammenfassung

Wir haben ein Programm entwickelt, welches es einem Benutzer, der über kein Expertenwissen verfügt, erlaubt, Textklassifikation mittels einer Auswahl von Maschinen- und Deep-Learning Algorithmen auf einfache Art und Weise durchzuführen.

Das Programm besteht aus einer graphischen Benutzeroberfläche, der Implementierung der zugrundeliegenden Algorithmen und einem Server, der die anderen beiden Komponenten verbindet. Der Hauptfokus lag auf dem Verbinden der verschiedenen verwendeten Werkzeuge and dem Darstellen der Berechnungen in einer optisch ansprechenden Weise. Folglich haben wir, statt die Algorithmen zu optimieren, bereits verfügbare Bibliotheken für die eigentliche Implementierung verwendet. Bei einigen dieser Bibliotheken waren Anpassungen notwendig, um deren Ausgabe für die automatische Weiterverarbeitung verwenden zu können.

In den letzten Jahren wurden viele Algorithmen entwickelt, welche für Maschinen und Deep-Learning eingesetzt werden, jedoch benötigen diese oft eine/n erfahrene/n Programmierer/in, um sie verwenden zu können. Ausserdem ist das Aufsetzen und Konfigurieren der Algorithmen oftmals auf Intuition basiert und das Vergleichen verschiedener Algorithmen auf den gleichen Daten ist verhältnismässig aufwändig. Im ersten Teil werden wir vorangegangene Arbeit in diesem Bereich erläutern.

Anschliessend werden wir einen Blick auf unsere Applikationen und deren Funktionen und Merkmale werfen. Dabei erklären wir, wie die verschiedenen Komponenten miteinander interagieren und welche Herausforderungen uns dabei begegneten. Dieses Kapitel wird durch visuelle Beispiele ergänzt.

Eines unserer Ziele ist, die Verwendung von Textklassifikation zu vereinfachen, insbesondere bei einer dahingehenden vorhandenen Unsicherheit, welcher Algorithmus am besten für die jeweilige Aufgabe geeignet ist. Daher haben wir einen Test mit zwei Personen, welche mit dem Projekt nicht vertraut waren, durchgeführt. Die Ergebnisse dieses Tests werden im anschliessenden Abschnitt präsentiert.

Da die Applikation auch einen praktischen Nutzen haben sollte, haben wir ein Kapitel möglichen Anwendungsfällen gewidmet. Dabei betrachten wir sowohl den unmittelbaren Anwendungsfall wie auch mögliche Auswirkungen, welche diese Applikation auf die Forschung im Bereich der Textklassifikation im Allgemeinen haben könnte.

Zum Schluss betrachten wir unsere Arbeit aus der Vogelperspektive und diskutieren die in diesem Projekt erreichten Resultate. Da dieses Projekt auf eine Fortsetzung hin ausgerichtet ist, werden wir mögliche Erweiterungen untersuchen und Funktionalitäten betrachten, welche in weiterführenden Projekten implementiert werden können.

Preface

Every day new tools are published, yet only a small fraction is actively maintained and used in production. There are many factors contributing to whether it is used in practice. One of these factors, we believe, is how easy it is to use a tool and whether its interface appeals to a developer.

In machine and deep learning, most tools focus on achieving good results, yet few have a pleasant interface and even fewer a graphical user interface. By building a framework to connect these algorithms to a graphical user interface, we hope to make these tools more accessible.

We would like to thank our coaches, Mark Cieliebak and Don Tuggener, for taking their time every week to give us feedback on our work and being patient with us. We are looking forward to our bachelor thesis with you. Furthermore, we would like to thank Jan Deriu and Fernando Benites for performing the user test. Special thanks to Monika Metzler for proofreading this report.

Contents

Abstract	1
Zusammenfassung	2
Preface.....	3
1 Introduction.....	7
2 Related Work.....	8
2.1 Automatic Machine Learning	8
2.1.1 Auto-Sklearn.....	8
2.1.2 Prodigy.....	8
2.1.3 AutoML.....	8
2.2 Text Classification Tools.....	8
2.2.1 GATE	8
2.2.2 IBM SPSS Modeler Text Analytics	9
3 Application.....	10
3.1 Classification	10
3.1.1 Changes done in Python files.....	10
3.1.2 Data input format	11
3.2 Backend.....	12
3.2.1 SQLite Database.....	12
3.2.2 Node.js Server	13
3.2.3 Python virtualenv Support	14
3.2.4 Challenges.....	14
3.3 GUI.....	14
3.3.1 Tech Stack.....	15
3.3.2 Structure.....	15
3.3.3 Challenges.....	16
3.4 Walkthrough	17
3.4.1 Start.....	17
3.4.2 Monitor	21
3.4.3 Evaluate	27
4 Implemented Algorithms.....	32
4.1 Overview.....	32
4.1.1 Bag Tree.....	32
4.1.2 Deep-MLSA.....	32
4.1.3 Multinomial Naive Bayes.....	32
4.1.4 Random Forest	32
4.1.5 Support Vector Machine	32

4.2	Performance.....	33
5	Use Cases.....	35
5.1	One-click Solution for Researchers.....	35
5.2	Understanding Algorithm Suitability and Solving Text Classification.....	37
6	User Test.....	38
6.1	Instructions.....	38
6.2	Setup.....	38
6.3	Results.....	39
6.3.1	Starting a Job.....	39
6.3.2	Monitoring a Job.....	39
6.3.3	Evaluating a Job.....	40
6.3.4	Additional Comments.....	40
6.4	Conclusion.....	40
7	Conclusion.....	41
7.1	Achieved Results.....	41
7.2	Features Planned for the Bachelor's Thesis.....	41
8	References.....	43
8.1	Bibliography.....	43
8.2	Figures.....	49
8.3	Tables.....	50
9	Appendix.....	51
9.1	Official description of the task.....	51
9.2	Installation Instructions.....	52
9.2.1	Using Docker.....	52
9.2.2	Manual Setup.....	53
9.3	Port Mappings.....	53
9.4	Meeting Notes.....	55
9.4.1	2017-09-19.....	55
9.4.2	2017-09-27.....	56
9.4.3	2017-10-04.....	56
9.4.4	2017-10-10.....	57
9.4.5	2017-10-24.....	58
9.4.6	2017-11-01.....	58
9.4.7	2017-11-08.....	59
9.4.8	2017-11-15.....	59
9.4.9	2017-11-22.....	60
9.4.10	2017-12-06.....	61

9.4.11	2017-12-13	62
9.5	GUI Sketches.....	64
9.6	Timetable.....	65
9.7	License Information	66

1 Introduction

Our task was to develop a tool which allows a user to perform text classification using machine and deep learning algorithms while not requiring the user to write code, and evaluating the results in an effort to give the user an idea which algorithms are suitable for this particular data set and which are not. We were also asked to use readily-available tools to implement said algorithms and adapting code to our needs. Optimizing the algorithms, or the preprocessing and feature extraction step, however, is outside of the scope of this work and is scheduled for a follow-up bachelor's thesis.

It has proven to be useful to devise a "Definition of Done" [1] to have a clear understanding what is part of this project and what is not and to define the scope of the project. Our Definition of Done¹ reads as follows:

- Our application offers a selection of five different algorithms, some of which have multiple configurations, i.e. parameter settings.
- Training and test input data sets can be uploaded.
- We only guarantee execution with the files we provide i.e. CSV files with a fixed column order.
- All parts of the application run on the same machine.
- The application runs the algorithms in parallel.
- While executing the algorithms, the application displays the current score for each one of these processes provided the underlying algorithm exposes this datum.
- If available, the GUI displays a graph of the F1 score over time per algorithm.
- Once an algorithm is done, the final score is displayed, and the underlying model can be used for further evaluations, string and file input, through an interface
- A user can terminate individual processes or a whole algorithm.
- It is possible to switch between different jobs, no matter their execution state provided the host has the necessary computation power.
- Optimizing the algorithms, parameters, preprocessing, and feature extraction is not part of this project.
- Since this work will, as previously mentioned, be continued in the form of a bachelor thesis by the same team, writing a prioritized feature list (see chapter 7.2) for the follow-up project was also part of the assignment.

Please see chapter 9.1 for the official task description.

The nickname of this project, "The Good, The Bad, and The Ugly", the title of a famous Western movie [2], is a play on words with the three commonly used labels in machine learning, "positive", "negative", and "neutral".

¹ Please note: while the DoD is a core instrument in Scrum teams, we did not expressly use Scrum techniques. The attentive reader will note, however, we made use of other Scrum techniques such as sprints, too.

2 Related Work

Classic machine learning requires a lot of human work. Annotations need to be done by hand and the models need to be selected and finetuned by an expert in machine learning. This project aims to reduce some of the workload by offering multiple algorithms to simultaneously classify data. This section will focus on other automatic machine learning (Auto-ML) tools and some text classification tools.

2.1 Automatic Machine Learning

In recent years plenty of projects have yielded promising results in the field of Auto-ML. There are different parts of the machine learning process that can be automated. The following subsections are a selection of projects that all focus on automating something else.

2.1.1 Auto-Sklearn

The Sklearn [3] library provides several classification algorithms that can be configured with multiple parameters. Auto-Sklearn is designed to take the selection of the estimator and the parameters out of your hands. The hardware resources it uses can be limited as well as the time to train one model. [4]

2.1.2 Prodigy

The goal behind Prodigy is to make annotation the data as simple and fast as possible. It even provides some trained models for text classification, e.g. sentiment, topic and intent. Otherwise, the user can design one themselves. The data is classified by the model and displayed with the predicted label. The user can then choose to confirm or deny the prediction. With each user input, the model gets optimized in real time. [5]

2.1.3 AutoML

Google's AutoML has automated the building and testing of neural nets. A controller neural net designs a child neural net and tests it. After analyzing the performance of the child, the controller builds another child taking the evaluation of the previous children into account. The AutoML runs for thousands of cycles and the resulting net is on par with state-of-the-art neural nets designed by machine learning experts. [6]

2.2 Text Classification Tools

The following subsections show two very different text classification tools, the first one is an open source framework to develop your own models, while the second is a paid service for data mining that already includes the needed models.

2.2.1 GATE

General Architecture for Text Engineering, short GATE, has been around since 1995. It offers multiple tools for text analysis.

- GATE Developer: An IDE that provides interactive GUIs to build and analyze algorithms.
- GATE Teamware: A web-based platform for collaborative curation and annotation of data.
- GATE Mimir: A framework for implementing indexing and search functionality across different data types.

Using GATEs APIs search engines or text classifiers can be built and embedded into existing applications. [7]

2.2.2 IBM SPSS Modeler Text Analytics

The IBM SPSS Modeler Text Analytics is designed to process and classify unstructured data like reports, e-mails and meeting notes. It extracts concept from the text and sorts these concepts into categories. The tool provides different models for the data mining. How well a model performs is subject to the data it analyses. [8]

3 Application

The application consists of three main code repositories: *classification* (chapter 3.1), *backend* (3.2), and *gui* (chapter 3.3). In this chapter we look at each of these repositories from a technical point of view which is complemented by a walkthrough of user interface (chapter 3.4).

3.1 Classification

At the heart of the project lies the classification repository. Here the different models get trained and evaluated. At the moment, the repository contains five open-source algorithms written in python. Four of them were originally implemented by Poyu Li in Sklearn [3] to classify IMDb² review ratings based on the review contents [9], the other one, the deep-mlsa [10], a CNN built by SpinningBytes for the sentiment analysis of tweets.

3.1.1 Changes done in Python files

To integrate them into the framework provided by the backend and GUI repository some of the code had to be changed. Because it should still be possible to update the tools the changes were kept as minimal as possible. The necessary alterations can be summarized in four groups:

- Standardize the call of the python scripts
- Output redirection to JSON
- Saving the best model
- Standardize the output

In total only 5 files were changed and 1 created to accommodate all requirements. For an overview of all changes, consult Table 1.

Tool	File	Alteration
deep-mlsa	runner.py	<ul style="list-style-type: none"> - Standardize the call of the python scripts - Output redirection to JSON - Standardize the output
keras	utils/generic_utils.py	Output redirection to JSON
	utils/fit_utils.py	Saving the best model
Sklearn implementation	sentiment.py	<ul style="list-style-type: none"> - Standardize the call of the python scripts - Output redirection to JSON - Standardize the output - Saving best model
	evaluate.py	Created
Sklearn	model_selection/_validation.py	<ul style="list-style-type: none"> - Output redirection to JSON - Saving the best model

Table 1: Changed files

² IMDb refers to the “Internet Movie Database”, a platform which provides information about movies and allows users to leave a review on a movie.

3.1.1.1 Standardized call of the script

Six switch characters were defined and implemented in both scripts main files. For the complete list see Table 2. The changes were done in the main files of the tools: *runner.py* for the CNN and *sentiment.py* and *evaluate.py* for the Sklearn. All parameters that contain paths are saved in the environment string of the operating system. This is done so that no additional arguments need to be passed along to other methods or even other libraries. Once saved, the path can be accessed from everywhere by calling `os.environ['path_variable']`.

Short	Verbose	Example	Required	Function
-c	<code>--config=</code>	config.json	Always	Path to the configurations for the algorithm.
-o	<code>--output=</code>	output.json	Always	Defines where the output is saved. The name of the file is the UUID of the job.
-r	<code>--train=</code>	train.tsv	Train	Contains the data to trains the model.
-e	<code>--test=</code>	test.tsv	Always	Contains the data to test the model.
-m	<code>--model=</code>	model.sav	Test	Path to the model that is to be evaluated.
-s	<code>--single=</code>	"I am happy"	Test (optional)	If the option is present the tool returns a single label for the given query.

Table 2: Command line paramters

3.1.1.2 Output redirection to JSON

The redirection of the final output to JSON is done in the main files as above. Redirecting the intermediate results posed more of a challenge because the command line printout is done in library files. The deep-mlsa uses the Keras [11] library, here, JSON output was added to the method `update()` in `utils/generic_utils.py`. For the Sklearn the problem was solved similarly by editing the `cross_val_score()` method in the `model_selection/_validation.py` file of the Sklearn library.

3.1.1.3 Saving the best model

It is possible to evaluate all algorithms with epochs once the first intermediate score is displayed. For this, the best model needs to be saved. Fortunately, the CNN already had this function implemented. All that had to be done was redirecting the output so that the models could be allocated to a job. The best model is saved in the folder `models/supervised_phase`, using the UUID taken from the output parameter as the name. For the Sklearn a solution had to be implemented. Because the tool does quite a bit of preprocessing, not only the model needed to be saved but also three additional files. In `models/a` folder with the UUID of the job is created to store the four files needed for the evaluation. The preprocessing files are saved in `sentiment.py` and loaded in `evaluate.py` using the pickle library. The model itself is pickled in the file `_validation.py`. Conveniently, those are the same files that had previously been altered.

3.1.2 Data input format

For the training job, two data files are required, one containing the training and one the testing data. It is recommended to use 20% of the data for testing and keeping the distribution of the labels the same for both files.

At the current state of the framework, a fixed data input format is required. Each data entry is in a single line containing four values. The ID, followed by an abbreviation of the data language, and the classification label, and finally the classified text. The different values are separated by tabs and the data is saved as a `.tsv` file. Moving forward with the development, the input format needs to be addressed. Not only is it very inconvenient having to format the data exactly, but also all line breaks and tabs need to be removed from the texts.

3.2 Backend

The backend repository is the glue between the GUI (chapter 3.3) and classification (chapter 3.1) repositories (resp.). Whereas the GUI allows a user to interact with jobs (i.e. start a new job, monitor the status, run evaluations etc.) and the classification tools are responsible for the processing of the aforementioned jobs, the backend on one hand serves as an API for the GUI to control jobs and on the other hand manages the classification processes³.

The backend consists of an SQLite database and a server written in JavaScript on top of Node.js. As a result, the server is very light-weight (less than 1,000 lines of code) and easy to set up thanks to the portability of SQLite and the Yarn⁴ package manager's robustness for installing JavaScript dependencies.

3.2.1 SQLite Database

SQLite was chosen as the database for several reasons:

1. The whole database resides in a single file [12] which makes portability, being desirable for local development in a team, very easy.
2. SQLite fully implements the SQL standard [13] and only a small subset of that standard was required for this application.
3. SQLite itself is thoroughly tested [14] and, notwithstanding its simplicity, is very performant and reliable [15].

3.2.1.1.1 Schema

The schema of the database is reflected in several parts of the application and for that reason, we would like to explore it further. The full schema can be found in the appendix. There are three main models – jobs, processes, and algorithms.

An *algorithm* refers to a directory in the classification repository, consists of a name and description, and defines the interpreter (e.g. Python). Such an example is the “Support Vector Machine” algorithm. Furthermore, an algorithm, for it to be usable, has one or more *algorithm configs* which specify the command line arguments passed to an executable or script for various different calls (train, evaluate string, evaluate file). Each *config* has a name and optionally a description.

A *process* is an instance of an *algorithm config* and as such corresponds to an OS-level process. A *process* stores structured (JSON) and unstructured (plain text) output of a running algorithm as well as extracted and computed data (e.g. maximum F1 score, current F1 score etc.). Additionally, meta information such as PID⁵, hostname, status, and timestamps are stored.

A *job* is started by a user in the GUI. It consists of one or more *processes*, which were created based on the *algorithm configs* the user selected, and store links to the training and test files the user provided. Based on the training file, a baseline is computed and stored as well as an optional user-specified job name. This data is augmented by meta information such as job status and timestamps. Since this model is the only user-addressable model, it also has a time-based universally unique identifier (UUID v1) [16] for simplified URL-generation.

³ Processes, in this context refers to processes in the sense of OS-level processes.

⁴ Yarn is a package manager for JavaScript and an alternative to npm. It is developed by Facebook and its main goals are performance, reliability, security.

⁵ The process ID given to a running algorithm by the OS.

3.2.1.1.2 Access

The database is exclusively accessed and controlled by the Node.js server. Upon starting the server, it checks whether the database exists and if not, it creates the database together with its tables and runs an init file which populates the database with values. This approach, in our experience, has proven to make development simpler. The necessary scripts can be found in the *db/* subdirectory.

3.2.2 Node.js Server

As mentioned in the previous section, the server is responsible for managing and interfacing with the SQLite database. Additionally, it exposes an API on port 3000 to the GUI using an Express [17] server, and it interacts with processes.

The JavaScript code makes heavy use of well-tested and well-known libraries such as Express, Moment.js [18], Lodash [19] and is built on top of the Node.js, a JavaScript runtime [20]. As there are almost two dozen direct dependencies, we chose to rely on Yarn [21] to manage the state of packages and ensuring reliable set-up across machines and operating systems. To ensure consistent coding style and use of best practices, we used ESLint [22].

JavaScript makes heavy use of asynchronous code and both the web server library and the SQLite library we decided to use, make heavy use of callbacks. To prevent landing in the infamous Callback Hell [23], we used the new *async/await* keywords (where possible and appropriate). These newly supported [24] keywords make asynchronous code look almost identical to synchronous code and allow for easier reasoning of the code [25].

In addition to the database, the backend also stores data on disk, relative to its root directory. These files are structured and unstructured output files produced by the classification algorithms and are removed when the database is recreated as they are named after the job UUIDs and are no longer of use as soon as the database has been regenerated. The backend also stores the training and test files uploaded by users on the filesystem.

The server itself is described in *src/server.js* and defines a variety of GET and POST routes. Depending on configuration, logging of incoming and outgoing HTTP requests is performed, which can be useful during development. This feature is automatically disabled in production by setting the environment variable *NODE_ENV* accordingly. As a user may upload files, the server also supports multi-part form data. Since the server and the GUI are not served on the same port, Cross-Origin Resource Sharing HTTP headers [26] are set automatically. For simplicity, these headers are set liberally.

Just like *src/server.js* is responsible for the web server, *src/db.js* handles database communication including initialization and queries. SQL queries are exposed as functions to other JS modules.

Since the *job* and *process* model are more involved than the *algorithm* model, there is an additional JS file for each of the two models in *src/models*. The *job.js* is responsible for creating a job and starting the corresponding processes, which involves setting up pipes for *stdout* and *stderr*, and resolving the arguments working directories for the algorithms. Some of these tasks involve path names which are generated and resolved by *src/helpers.js*. The necessary functionality to manage an individual process, such as monitoring its JSON output, killing and tailing⁶ the process as well as the exit handler, is provided by *src/models/process.js*.

⁶ In the sense of the UNIX command *tail*.

One unintended consequence of this setup is, whenever the server is restarted (which, while developing, happens every time a source file is saved due to a watcher process⁷), any running algorithms are killed, too. This is caused by a `fs.spawn()` [27] call in `src/models/job.js` which spawns a new child process which in turn is killed automatically when its parent process dies. We will discuss this further in chapter 3.2.4.

3.2.3 Python virtualenv Support

All of the currently supported machine and deep learning tools are written in Python and each have different, sometimes conflicting, requirements for packages/libraries upon which they are built on. To overcome this problem, we make use of a well-known package in the Python world, `virtualenv`, which “is a tool to create isolated Python environments” [28]. The algorithm in `src/helpers.js` is capable of detecting a local `virtualenv` (in a subfolder called `venv`) and using the Python interpreter in that virtual environment in a cross-platform fashion (Windows and Linux).

3.2.4 Challenges

As previously mentioned our application was developed under Windows and while Node.js is to a large degree platform-independent (or even platform-agnostic [29]), there are instances where Windows and POSIX-compatible systems do not behave in the same way. One such example is the aforementioned `child_process.spawn()` which we make use of. While coding that particular feature, we switched back and forth between `spawn()` and the related `exec` (and `execFile`) to satisfy our requirements (having control over the process and capturing `stdout` and `stderr`) while also being able to pass arguments to a process without having to write custom logic for each new algorithm. In the end we chose `spawn()`, which seemed to behave fine under Windows 10 and Linux Ubuntu 16.04 LTS, even though argument-passing is slightly cumbersome as every pair of keyword and value has to be an array. We solved this problem by storing the argument string as a JSON array in the database and then using string replacement to substitute the pre-defined variables with the actual values in the argument list.

Another challenge we have briefly touched on previously is the issue of long-running processes and ensuring stability of the server as once the server’s process terminates, all the algorithms, which are child processes, terminate, too. While it is possible to change this behavior [30], it makes the “process killing” feature of the GUI harder to implement and if one is not careful during implementation, a process cannot be killed unless a user has system-level access to the process. As most machine learning algorithms are fairly resource-hungry, we decided against using the detached options at this stage in our work.

3.3 GUI

The (most) visible part of our work is arguably the graphical user interface (GUI). It is the primary source of interaction for any end-user of our application and allows a user to start a job, monitor any job on the server, and, provided a given job has made sufficient progress, run evaluations against the computed models.

The GUI is web-based and runs in any modern browser. By using a web-based approach, we, also by nature of the platform, decouple the presentation from the underlying data and process management, which is handled by the backend. Since machine and deep learning algorithms tend to use a lot server resources, one might add cluster support to the backend. In that case, it will be beneficial not to have a tight coupling between front- and backend.

⁷ This process, in our case `nodemon`, watches for any changes on the filesystem and automatically restarts the server. This is very helpful during development and can easily be turned off for production.

3.3.1 Tech Stack

The GUI heavily relies on Twitter Bootstrap v4 [31] and the Vue.js framework [32]. In an effort to focus on the what rather than the how while also ensuring rapid development and prototyping and browser compatibility, we chose the well-known and feature-rich Bootstrap framework which comes with a lot of built-in components. The decision to use Vue.js was on one hand due its appealing data-driven approach and on the other hand out of curiosity and desire to get familiar with the framework. Fortunately – and due to the popularity of both Bootstrap and Vue, we were able to use a package by the name of “Bootstrap Vue” [33] which bridges the two frameworks and greatly facilitated development.

At its core, Vue uses a virtual document object model (DOM) where a user can write components (custom HTML elements) and data, logic and event handlers are defined by attributes. This combination results in code which is both easy to write and read and files that are self-contained.

As mentioned before, Vue is data-driven, which means the DOM – and thus what is displayed to the user – depends on the state of the data and any modifications a user performs, updates the data store which then updates the view. In our application, the initial state is loaded via AJAX/XHR [34], i.e. asynchronous HTTP requests initiated by the JavaScript code running in the browser. These HTTP requests, which in the first phase use the GET HTTP verb, are then handled by the backend’s Node.js server.

The frontend retrieves all data form the backend, including algorithms a user may start. This allows for a user to use the same GUI to (theoretically) connect to different backends/servers while the GUI is always served from the same server.⁸

To both facilitate bug fixes and implementing new features as well as maintainability and extensibility by other developers, the GUI adheres to most sections of the official Vue.js style guide [35].

Just like for the backend code, we also used ESLint [22] for the GUI’s code to ensure consistent use of syntactic features, spacing, and preventing bad practices.

3.3.2 Structure

Due to unfamiliarity with the Vue.js ecosystem at the beginning of this project, we chose to base the structure of the source code on a freely available template [36], which combines Bootstrap, Vue, and a few other tools such as Webpack [37] and Babel [38].

Webpack bundles and resolves all the assets’ dependencies optimized for both development (e.g. support for source maps [39]) and production (e.g. minification of assets [40]) which allows for cleaner code (by e.g. using JS modules) and does not bother the developer with setting up an assets pipeline.

Babel on the other hand, as the name might suggest, ensures language compatibility, in this case for JavaScript. JS, or rather the underlying ECMAScript specification, is regularly revised [41] – especially in the past few years – and extended, yet browser support is lagging behind – due to vendors and, obviously, outdated browser versions. Babel allows a developer to use the latest ECMAScript version which it then compiles for use with older ECMAScript versions to ensure browser compatibility. A portion of features introduced by newer versions of ECMAScript is syntactic sugar which helps code readability and is therefore desirable to use.

⁸ This feature is not implemented at the time and would require carefully setting HTTP CORS headers.

As Vue is centered around the concept of components, and therefore, all of our work can be found in *src/components*, save for the two files located in the *src* directory and *src/router/index.js*. The two files in *src* serve as a skeleton for the application and initialize Vue.

As the name suggests, *src/router/index.js*, defines all the applications routes and defines which components and packages are loaded into the application.

Our core components, the ones responsible for starting, execution, and evaluating jobs, can be found in the three subfolder *Jobs*, *Monitor*, and *Evaluate* (resp.).

The *Jobs* subfolder only contains one component, the *Start* component, which displays the form to start a new job.

Monitor and *Evaluate* both have a component called *Choose* which allows the user to choose a job on the server to monitor or evaluate (resp.) in case the user has not used a link in the application which already contains the job ID. Once the user has decided which job to use, the *Job* component is used to display the corresponding view.

In the case of monitoring a job, there is an additional subfolder, *Processes*, which contains two more components – *Overview*, and *Detail*. These components are responsible for displaying a simplified or detailed (resp.) view of a process in a job. By refactoring the *Monitor/Job* component into these sub-components, the code not only gained readability and made the component more minimal, it also helped simplifying some of the logic around calculating the “best” process etc. and thus makes reasoning about the code easier.

We will not further discuss the other component files (e.g. the ones located in the *Snippets* subfolder) which are responsible for the navigation, footer, and static sites (home, about).

3.3.3 Challenges

Apart from diving head-first into the ecosystem of a new framework, Vue.js, we faced some other – and mostly unrelated – challenges, most of which are natural when developing a GUI, yet still deserve to be mentioned.

It is often tempting to implement a UI in a proof-of-concept fashion, meaning you are satisfied when it works, yet not a lot of time is spent reflecting on the UI, whether it is intuitive and serves its intended purpose *well* and – in short – the usability (UX). We were fortunate to have weekly meetings with our coaches where we could get feedback on new features and had to explain our thinking behind some decisions. When you explain a feature to someone else, you often realize if it is intuitive or whether it requires a complicated explanation. An example where a lot of feedback went into, are the progress bars in the job overview. Additionally, as time passes, and you *use* a feature you built a few weeks prior, you start noticing the little things that make it easier to use – or not. One such example are buttons where you can un-/check all checkboxes.

Another problem is building the GUI based on how the backend operates and you, as a developer, interact with it, i.e. you focus on the developer instead of the user. What makes sense for a developer may not always be the most obvious way to perform an action for a user – even if that user has a computer science background. While not always easy to do, it is worth the time and effort to try to put yourself in the shoes of the user and imagine how they would expect this action to work.

Another challenge was striking a balance between performing work in the backend and in the GUI, e.g. whether you calculate the maximum F1 score of all processes in a job on the server or the client. Part of this problem was solved, up to a certain degree as byproduct, by refactoring the

Monitor/Job component; first it was a big monolithic component which performed a lot of calculations, some with process-scope, some with job-scope – and some with all-processes-in-the-job scope. By refactoring the component and having sub-components per process, it quickly became a pattern to perform the last category of calculations on the server while keeping some of the calculations (which fall into the category of “computed attribute”, e.g. the maximum of two scores) on the client. Additionally, properties should be calculated where it is sensible to perform them and only on-demand.

And lastly, dealing with and designing for empty state is often a challenge and this application is no different. Since all of the GUI’s content is loaded asynchronously via the network, a number of failures (connection to the server, server is down/has crashed etc.) can occur and at the very least, a loading icon has to be displayed and information regarding a failure has to be shown as well. Furthermore, we display confusion matrices and “F1 score per epoch”-graphs – if we can. Some algorithms either do not have epochs or it is not possible to extract the data necessary for the graph (the same applies to confusion matrices). This has to be taken into account and the user should know a certain piece of information is missing and not have to assume a bug occurred.

3.4 Walkthrough

While the user interface consists of no more than three main screens, they convey a lot of information and are the main interaction point with the user. In this section we would like to present the main screens in the order a user typically engages with them (see chapter 6) and point out the various features each screen (or component, see chapter 3.3) offers.

The application has a main navigation bar which allows the user to start, monitor, and evaluate a job:



Figure 1: The application’s navigation bar

As the application consists of three main screens, this chapter consists of three sections.

3.4.1 Start

When a user wants to start a new job, they would click on “Start a job” (see Figure 1) and be greeted with the following screen:

Start a New Job

Show extra information

Here you can start a new job. A job consists of:

1. Data - training and test data
2. A set of algorithms which should be run over the data sets and evaluated
3. (optional) A name for your job

Data sets

Your training data

Your test data

Algorithms All None

Please choose the algorithms you'd like to run on your data.

deep-mlsa

Code for WWW 2017 conference paper "Leveraging large amounts of weakly supervised data for multi-language sentiment classification" train

Naive Bayes

In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. run

Random Forest

Random forests or random decision forests[1][2] are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set. run

Support Vector Machine

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). default
 standard
 signed
 unsigned

Bag Tree

Individual decision trees tend to overfit. Bootstrap-aggregated (bagged) decision trees combine the results of many decision trees, which reduces the effects of overfitting and improves generalization. run

Name

You may give your job a name. This allows you to better identify your job, but is optional. If you don't provide a name, one will be generated for you.

Optionally, you can give your job a name

Start job on server

Figure 2: The "Start a New Job" screen

18

This screen consists, as described below the heading (see Figure 2) of three parts:

1. Data sets (see chapter 3.4.1.1)
2. Algorithms (see chapter 3.4.1.2)
3. Name (see chapter 3.4.1.3)

These three parts are distinguishable by boxes (or cards, as they are known in the Bootstrap world [42]) to help visually guide the user. This pattern of boxes is used throughout the application and, in case they contain input fields, a thin border around the card indicates whether all required inputs have been filled in or not.

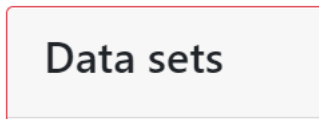


Figure 3: Red indicates not all required inputs have been filled

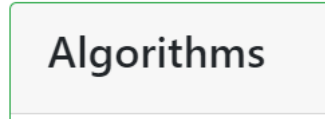


Figure 4: Green indicates all required inputs have been filled

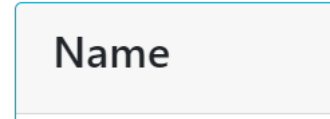


Figure 5: Blue indicates there are non-required inputs that are empty

Additionally, the submit button is only active once all required fields have been filled and also changes its color accordingly:



Figure 6: The submit button has a yellow background and is disabled since not all required fields have been filled.

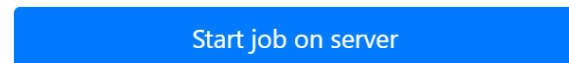


Figure 7: Once all required fields have been filled, the button is active and has a blue (primary color) background.

3.4.1.1 Data sets

The order has been chosen carefully to ensure the flow feels natural. The user would first choose the data sets (which are formatted according to chapter 3.1.2) by either using the “Choose File” button (see Figure 8) or by dragging-and-dropping the file into the corresponding file input. This is the only time the user (partially) leaves the context of the browser and interact with another program (the file browser of the local machine). By asking the user to perform this step first, they will not be further interrupted and can stay within the application. To start a job, both train and test data have to be provided, which is also indicated by the red border in Figure 8.

Figure 8: Choosing the data sets when starting a job

3.4.1.2 Algorithms

By default, all algorithms are active since one of the goals of this application is to make comparison of different algorithms on the same input more straightforward. A user can easily toggle between “all” and “none” for the algorithm selection using the corresponding button⁹ on the top right of the card (see Figure 9). It should be noted, however, while the “none” button may be useful when one only wants to run a small number of algorithms, at least one algorithm is required

⁹ The buttons are two-way coupled i.e. clicking on e.g. the “All” button, checks all algorithm configurations and when all algorithm configurations are checked, the “All” button is active (as seen in Figure 9). The “None” button works analogously.

to start a job. Each algorithm is accompanied by a small description text and has one or more checkboxes (these correspond to the algorithm configuration described in chapter 3.2.1.1.1) to toggle de-/activate running the same algorithm with different configurations.

Algorithms All None

Please choose the algorithms you'd like to run on your data.

deep-mlsa

Code for WWW 2017 conference paper "Leveraging large amounts of weakly supervised data for multi-language sentiment classification" train

Naive Bayes

In machine learning, naive Bayes classifiers are a family of simple probabilistic classifiers based on applying Bayes' theorem with strong (naive) independence assumptions between the features. run

Random Forest

Random forests or random decision forests[1][2] are an ensemble learning method for classification, regression and other tasks, that operate by constructing a multitude of decision trees at training time and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees. Random decision forests correct for decision trees' habit of overfitting to their training set. run

Support Vector Machine

In machine learning, support vector machines (SVMs, also support vector networks) are supervised learning models with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier (although methods such as Platt scaling exist to use SVM in a probabilistic classification setting). default
 standard
 signed
 unsigned

Bag Tree

Individual decision trees tend to overfit. Bootstrap-aggregated (bagged) decision trees combine the results of many decision trees, which reduces the effects of overfitting and improves generalization. run

Figure 9: Choosing the algorithms when starting a job.

3.4.1.3 Name

Once the files have been selected and possible adjustments to the algorithm selections have been made, the user may give the job a name (see Figure 10). This step, however, is optional and if no name is provided the job will be named after its starting time, formatted depending on the server's locale [43]. Yet naming the job is suggested, especially in a multi-user environment in order for different people to be able to identify their job(s).

Name

You may give your job a name. This allows you to better identify your job, but is optional. If you don't provide a name, one will be generated for you.

Optionally, you can give your job a name

Analyzing tweets using deep-mlsa

Figure 10: The input field where a job can be named before starting it.

Once the button is blue (see Figure 7), the user can click on “Start job on server” and is automatically redirected to the “Monitor” screen for the job.

3.4.2 Monitor

The “Monitor” screen (and to an extent also the “Evaluate” screen, see chapter 3.4.3) make heavy use of colors to indicate the status of a job or process. As such, this is a sensible time to introduce the colors and the meanings associated with each color¹⁰ in the context of the application¹¹ (see Table 3). These colors are used consistently and frequently to (subconsciously) be able to better relate the color with the text. Usages include:

- The job status in the title right of the job name (see chapter 3.4.2.1).
- In the overview (“F1 Scores”): a circle icon left of the name of the process as well as the bar chart of the F1 scores (see chapter 3.4.2.2).
- In the process card: the process’ status, the F1 score, and the card’s border (see chapter 3.4.2.3).






Usage	Meaning
 <p><i>Figure 11: The grey created label</i></p>	The process has been created in the server’s database but has not started yet. In the current version of the application, this label is never visible since the processes are started immediately and transition to “running” (see Figure 13).
 <p><i>Figure 12: The green completed label</i></p>	The job or process has completed. In the case of a process, this indicates a non-zero exit code ¹² .
 <p><i>Figure 13: The blue running label</i></p>	This label is used when a process or job is running.
 <p><i>Figure 14: The cyan killed label</i></p>	Should the user have decided to terminate a process through the GUI, the process will have this label to distinguish it from “running” (Figure 13) and “errored” (Figure 15). ¹³
 <p><i>Figure 15: The red errored label</i></p>	

Table 3: An overview of the differently colored labels used and their meanings.

¹⁰ These colors correspond to the “contextual variations” [71] or “variants” [72] in the lingo of the underlying Bootstrap framework and Vue.js package (respectively).

¹¹ In practical usage, we expect the user to learn the colors and what they mean by using the application as there is a 1:1 mapping of color and text.

¹² This is based on the convention “[...] for a child process to return (exit with) zero to the parent signifying success.” [73]

¹³ The keen-eyed user might notice when killing a process, it occasionally has the “errored” label (Figure 15) for a brief time. The reason is as follows: when killing a process, the process returns with a non-zero exit status (i.e. it errored). After a brief timeout, the application then overwrites the status (which has automatically been set to “errored” by the corresponding handler) to “killed” as it is not able to distinguish, based on the exit handler, between an algorithm that has been terminated by the user and one that has errored.

The “Monitor” screen consists, strictly speaking, of two screens – one to choose which job to monitor (see Figure 16) and the actual monitoring screen (see Figure 17). In practice, the former’s on-screen time is close to zero, as the user is automatically taken to the latter after starting a job. Yet, the former screen allows to easily load the monitoring screen for any job.

Monitor

Enter the job ID you'd like to monitor on the server:

Figure 16: The screen where one can choose which job to load and monitor

Assuming the user has just started a job (as described in chapter 3.4.1), the monitoring screen is automatically loaded (see Figure 17). It consists of three parts, where the last part is repeated by as many algorithm configurations as have been chosen for this job.

Tweets3

completed Tools ▾

Monitor status	Training file	Test file	Run time
200 OK	tweets_train.tsv 2.12 MB	tweets_test.tsv 534 kB	This job ran for 2 hours (02:07:36)

F1 Scores current highscore: 49.12% (max: 70.39%)

<input type="radio"/> Naive Bayes: run	00:00:19		58.91% (67.93%)
<input type="radio"/> Support Vector Machine: default	00:13:37		45.22% (61.04%)
<input type="radio"/> Support Vector Machine: unsigned	00:13:07		42.75% (69.55%)
<input type="radio"/> Support Vector Machine: standard	02:07:36		49.12% (70.39%)
<input type="radio"/> Bag Tree: run	00:08:44		57.74%
<input type="radio"/> Random Forest: run	00:01:16		58.18%
<input type="radio"/> Support Vector Machine: signed	02:05:23		34.32% (69.48%)

Processes

completed PID: 71972 F1 score: **58.91% (67.93%)**

Naive Bayes

run

Confusion Matrix

Predicted/Actual	negative	neutral	positive	__all__
negative	219	228	108	555
neutral	220	949	339	1508
positive	107	482	953	1542
__all__	546	1659	1400	3605

Ended 4 hours ago on DESKTOP-7S0F3KC | Execution took a few seconds

F1 Score over Epoch Graph

Figure 17: The “Monitor” screen for a job which ran with all algorithms on the tweets input (excerpt, for space reasons)

3.4.2.1 Header

The first part consists of the job name and status, information about the runtime of the job as well as data about the training and test files (file size and name), as well as a “Tools” menu and the monitor status (see Figure 18). As the monitor refreshes automatically every 1,000 ms, there is an HTTP request every second ¹⁴. The status of the last HTTP request is displayed and colored according to its response status. During normal operation the output is “200 OK” [44], yet it might occasionally be a timeout (in case the server is under high load¹⁵).

¹⁴ Actually, there are $n + 1$ HTTP requests where n is the number of processes and the 1 request is for the job itself.

¹⁵ Such as when starting all algorithms on a dual-core laptop.

The “Tools” menu (see Figure 19) provides easy access to switch to the “Evaluate” screen for the job currently being monitored and, if applicable (i.e. if at least one process is still running), shortcuts to kill all processes and kill all processes of a given algorithm¹⁶.

An interesting detail about the runtime is the fact of it being presented in two different ways; on one hand it is presented as a *hh:mm:ss* string as one would expect, and on the other hand it is also displayed as a “humanized” [45] string i.e. an approximation of the timespan as an effort to make the application feel more natural.

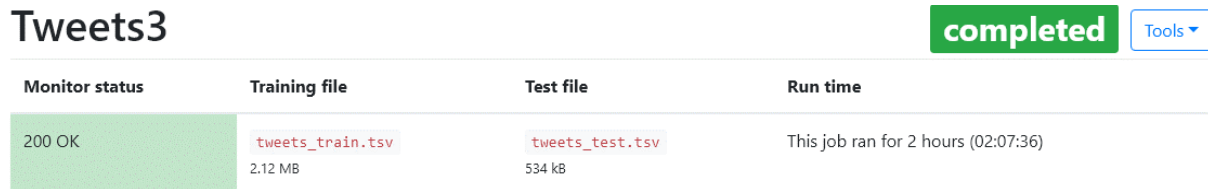


Figure 18: The top part of the “Monitor” screen

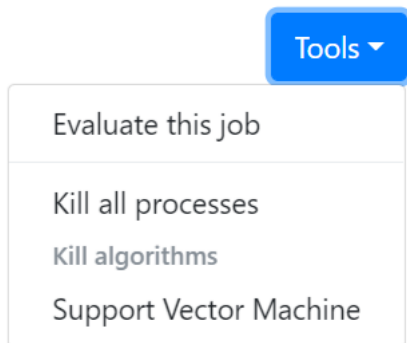


Figure 19: The “Tools” menu for a job

3.4.2.2 Overview

Following the header of the “Monitor” screen, is the overview (named after the corresponding Vue.js component, see chapter 3.3.2) which is again a card and is designed to give the user a quick overview of the processes in the job by providing insight into the process status, runtime, and current/max F1 score as well as, if available, a mini chart of the F1 score per epoch (a larger version of this chart is shown in the detail view of a process, see Figure 23) – and, as an added bonus, the currently highest scoring process is awarded a trophy (see Figure 20). The title of this section (“F1 Scores”) also contains the highest score of any process at the moment (i.e. the maximum of the final scores for completed processes and of the current scores of the running algorithms) as well as the highest score achieved in any epoch. The bar chart on the right packs a lot of information:

- The color is indicative of the process’ status (see Table 3).
- If there is only a single number, this means either the process’ final F1 score is equal to the highest F1 score achieved in any epoch or the process did not provide any details about epochs and the score is the final F1 score.
- If there are two numbers, the first value represents the current (if the process is still running) / final score of this process whereas the value in brackets refers to the highest score this process achieved in any epoch. A popover when hovering over the bar explains this as well.

¹⁶ This is useful when running multiple configurations of the same algorithm in one job.

- The length of the bar corresponds to the current score if the process is still running and to the max score once the process has completed.
- In case the process is running, and the current score is less than the maximum score, there is a yellow bar between the current and the max F1 score to indicate this gap (see Figure 21). This is also explained by a popover [46] appearing when hovering over the yellow area.

For each job, a baseline is calculated which corresponds to the score a random algorithm would achieve (see also chapter 4.2). This baseline is shown by a thin line above the bar, which is either black (see Figure 21) or red (see Figure 22). In the case of a red line, the maximum score this process has achieved in any epoch is below the score a random algorithm would achieve. Once the maximum score is above the baseline, the line turns black.

F1 Scores current highscore: 49.12% (max: 70.39%)

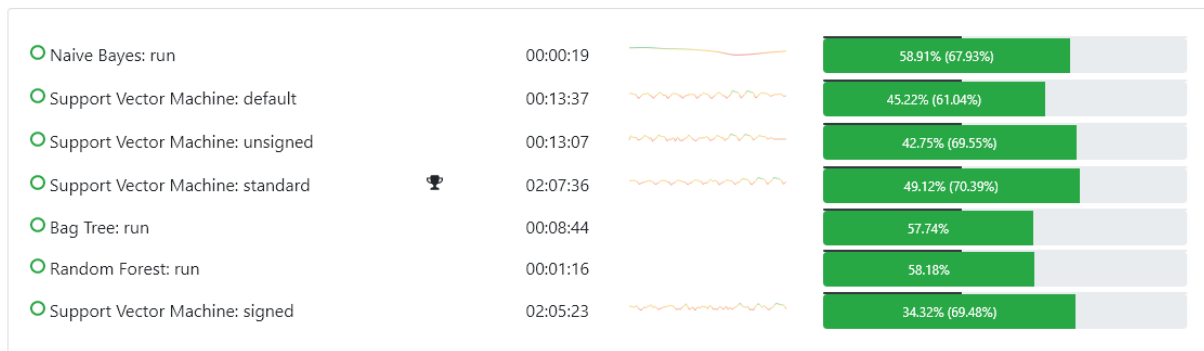


Figure 20: The overview card in the “Monitor” screen



Figure 21: A bar chart where the current score is less than the maximum score

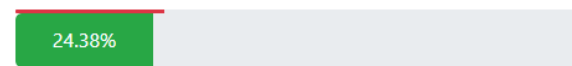


Figure 22: A bar chart where the maximum score is below the baseline

3.4.2.3 Detail

While the overview is designed and intended to gain a quick insight into how the algorithms are performing, which, depending on the use case, is all a user needs, the detail screen (see Figure 23) offers more information presented in a fashion with more whitespace which is “an important element of design” and “increase[s] content legibility” [47]. Each detail screen, following the “Processes” heading, is a card with a border colored according to the process’ status (see Table 3).

On the right side of the header slot [48] is a label with the process’ status and the process identifier¹⁷ (PID) [49] and on the left side the F1 score(s)¹⁸, again on a background colored with the process’ status color.

The footer slot [48] offers information on when the job was started or has ended (depending on whether the process is running) and, if it has completed, how long the execution took. As machine and deep learning algorithms tend to run for a long time, one is usually more interested in how long the process took approximately and not in the exact millisecond figure. For this reason, the durations are displayed in a “humanized” [45] form.

¹⁷ This corresponds to the ID given to the process by the operating system of the server and not to the ID of the process in the database.

¹⁸ For the explanation, please see the description of the overview.

The main content of the card starts with the title consisting of the name of the algorithm followed by the subtitle which corresponds to the name of the algorithm configuration. Following the title, is the confusion matrix on the left and the score-over-epoch graph on the right – assuming a process provides this information¹⁹. The confusion matrix is an important piece of information to see how well the predicted and the actual classes align (see also [50]). The graph shows the development of the F1 score per epoch, providing insight into whether the computed model is improving. This graph is based on the same data as the mini chart in the overview (see Figure 20).

While the score/epoch graph can be interesting to look at – and since it is updated automatically after every epoch, it gives a sense of a “live view” (see also chapter 6) – it can also be helpful when deciding whether to terminate a process before it has completed its computation. When the process is running, there is an additional element in the header slot: a red button on the right to kill the process (see Figure 24). Killing a process has the benefit of freeing up system resources which can be used for other processes (such as pending computations in the job being monitored).

For debugging purposes (both during development and usage of the application) as well as to extract information which is not presented by the GUI, there is the option to display the raw output (*stdout* and *stderr*, see [51] for more information) by clicking “Toggle raw output” (compare Figure 17 and Figure 23). This output may contain artefacts which decrease legibility such as when the underlying algorithm paints a progress bar in the console output. No methods are in place to treat this raw output and it is presented “as is”.

¹⁹ Missing information is either due to the algorithm not providing the information or the process not having completed yet.

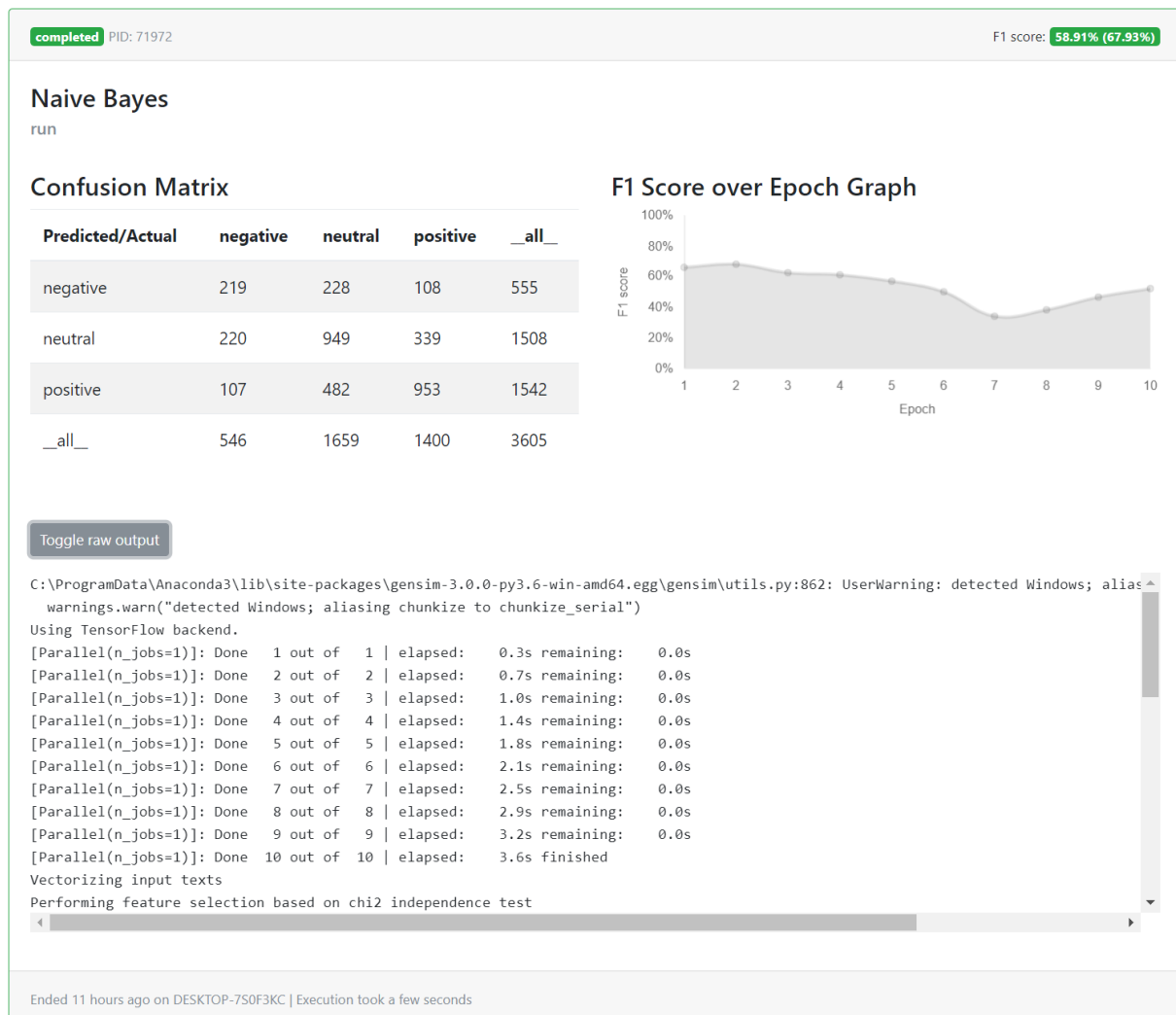


Figure 23: The detail view of a process in the overview (in this case: Naïve Bayes for Tweets) with expanded raw output

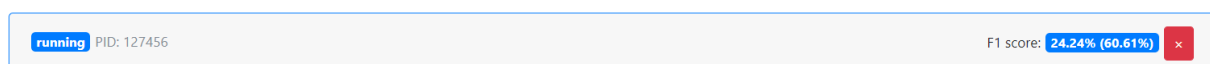


Figure 24: A process which is still running and thus can be terminated

3.4.3 Evaluate

Each algorithm computes a model (see chapter 3.1.1.3) which is saved after the algorithm has completed and, in case the process has more than one epoch, a model is also saved after each epoch, provided its F1 score is the highest score this process has achieved thus far (i.e. only the best²⁰ model is saved). The purpose of the “Evaluate” screen is to be able to interact with the computed models and run evaluations against these models (see Figure 25). The user may at any point in time, after starting the job, switch to the “Evaluate” screen, no matter whether zero, one, more, or all processes have completed.

This screen has a layout and structure similar to the “Start a job” (see chapter 3.4.1) and “Monitor” (see chapter 3.4.2) screens. There is a header accompanied by a short explanation on how to use this screen as well as a “Tools” menu. Following the header (see chapter 3.4.3.1) there are initially two cards, one to choose which models should be used for an evaluation (see chapter 3.4.3.2), and the other one to provide the evaluation input (string or file; see chapter 3.4.3.3). Additionally, for

²⁰“Best” is according to the F1 score where higher is better.

each evaluation that was run²¹, there is an additional card with the result of the evaluation (see chapter 3.4.3.4).

Evaluate Tweets3

[Tools ▾](#)

Here you can evaluate models generated by a job. An evaluation consists of:

1. A set of models you'd like to use for the evaluation
2. Input data

Models

Against which models would you like to run your input?

All models
No models
Best model only

<input type="checkbox"/> Naive Bayes: run	F1 score: 67.93%
<input type="checkbox"/> Support Vector Machine: default	F1 score: 61.04%
<input type="checkbox"/> Support Vector Machine: unsigned	F1 score: 69.55%
<input type="checkbox"/> Support Vector Machine: standard	F1 score: 70.39%
<input type="checkbox"/> Bag Tree: run	F1 score: 57.74%
<input type="checkbox"/> Random Forest: run	F1 score: 58.18%
<input type="checkbox"/> Support Vector Machine: signed	F1 score: 69.48%

Input

What would you like to feed as input to the computed models? You can provide a single string or a file.

Enter a string:

Upload a file:

Some required fields are empty.

Figure 25: The "Evaluate" screen for a job with no models selected and all inputs empty

²¹ Evaluation runs are not saved in the database and as such the results are lost once the browser windows is reloaded.

3.4.3.1 Header

The title consists of the job name and the word “evaluate” and the “Tools” menu offers a shortcut to jump back to the “Monitor” screen (see Figure 26).

Evaluate Tweets3

Here you can evaluate models generated by a job. An evaluation consists of:

1. A set of models you'd like to use for the evaluation
2. Input data

Tools ▾

Monitor this job

Figure 26: The header section of the “Evaluate” screen

3.4.3.2 Models

The next section is the “Models” card where the user can choose which models to use for the evaluation. In the case of algorithms that have epochs and save models after each epoch, the computed models of processes which are still running can be selected, too (see Figure 27). The card's border color is again indicative of whether the required inputs have been filled, i.e. if at least one model has been selected (compare Figure 25 and Figure 27). To increase usability, there are (similar to the “Start” screen; see chapter 3.4.1) buttons to facilitate using either all or no models as well as a separate button to only select the best model²². Apart from the name of the process which generated the model, the F1 score is also presented, with a background matching the color of the process' status (see Table 3).

Models

Against which models would you like to run your input?

All models
No models
Best model only

<input checked="" type="checkbox"/>	Support Vector Machine: default	F1 score: 56.96%
<input checked="" type="checkbox"/>	Naive Bayes: run	F1 score: 67.93%
<input checked="" type="checkbox"/>	Random Forest: run	F1 score: 58.55%

Figure 27: Choosing models where one process is still running

3.4.3.3 Input

After the user has selected the models to be used for the evaluation(s), the next step is to provide the input for the evaluation. The order of the cards was chosen based on the assumption a user is more likely to run several evaluations on the same group of models than to run the same input on different groups of models.

Input

What would you like to feed as input to the computed models? You can provide a single string or a file.

Enter a string:

Upload a file:

Figure 28: The two different input fields for the “Evaluate” screen

²² See footnote 9 on details about the coupling between checkboxes and buttons.

There are two types of input: string input and file input (see Figure 28) as there is the option to either provide a string (see Figure 29) as an input or upload a file (see Figure 30). As soon as either one of the inputs is not empty, the other field is hidden and reappears once the field with input has been emptied, as indicated by an explanatory text below the input field.

Enter a string:

To use a file instead, please clear this input field.

Figure 29: A string was provided as input

The file has to be in a specified format (see chapter 3.1.2) and it will be uploaded once the form is submitted.

Upload a file:

To use a string instead, please clear this input field.

Figure 30: A file was uploaded to be evaluated

As soon as both cards, “Models” and “Input”, have a green border, i.e. all the required fields have been filled, the submit button is active and changes from yellow (see Figure 6) to blue²³ (see Figure 31).

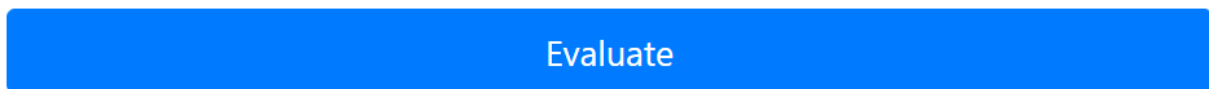


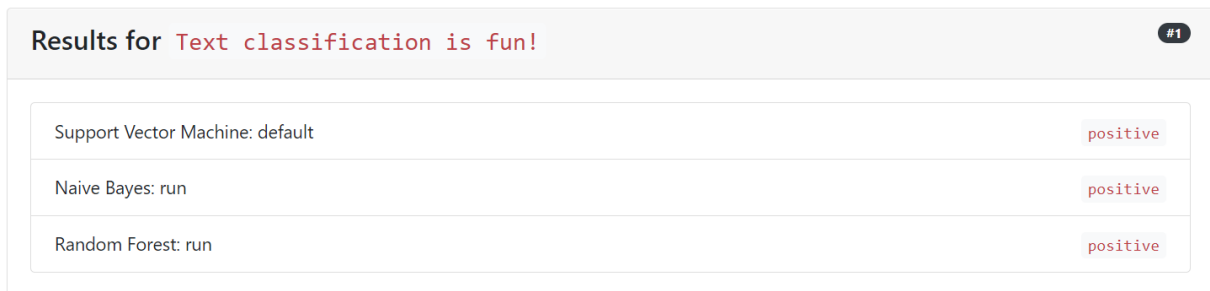
Figure 31: A blue (or green) button indicates the evaluation is ready to be started

²³ Once at least one evaluation has been run, the button remains green and the toggles between disabled and enabled to indicate whether a new evaluation can be started based on the input fields.

3.4.3.4 Results

The “Results” card is shown for every (see also footnote 21) evaluation and next to the title, which indicates the type of input used, is a strictly monotonically increasing counter to be able to differentiate between different evaluation runs. The runs are sorted in descending order and, as the time to perform an evaluation is different for each algorithm, the results are displayed as soon as they become available.

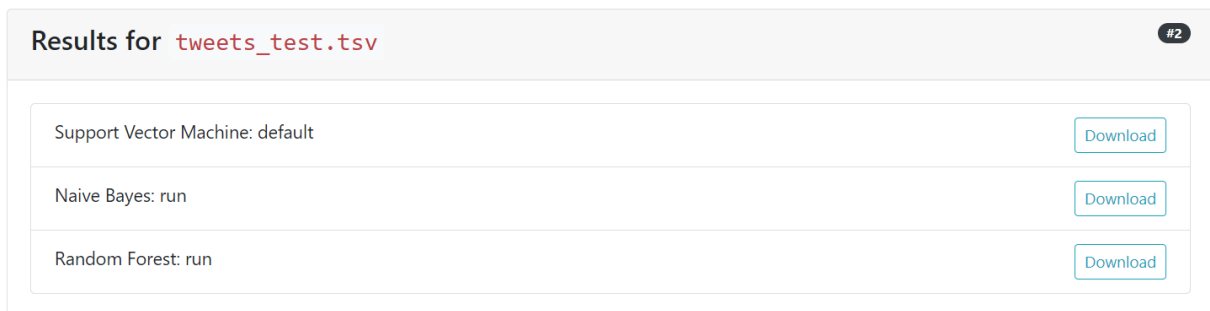
In case a string input was fed into the evaluation, the card displays (see Figure 32) the computed label as soon as the computation of the evaluation has finished.



Results for Text classification is fun! #1	
Support Vector Machine: default	positive
Naive Bayes: run	positive
Random Forest: run	positive

Figure 32: The results of an evaluation with string input

If a file was provided as input, the cards display a button to download the generated output file (see Figure 33) which, compared to the uploaded file, contains an additional column with the predicted label (see chapter 3.1.2).



Results for tweets_test.tsv #2	
Support Vector Machine: default	Download
Naive Bayes: run	Download
Random Forest: run	Download

Figure 33: The “Download” buttons are shown when a file served as input for the evaluation

4 Implemented Algorithms

This section offers an overview of the algorithms currently implemented in the framework and their performance. The algorithms are Bag Tree (BT), Multinomial Naive Bayes (MNB), Random Forest (RF) Support Vector Machine (SVM), all originally implemented in Sklearn for movie review analysis [9] and the deep-mlsa CNN [10].

4.1 Overview

4.1.1 Bag Tree

The bag tree implementation creates 100 decision trees. Bootstrap aggregation, or bagging, is used to reduce the variance. Each tree is built with a part of the data set that has a low variance. Usually a tree contains about $2/3$ of the data, while the rest can be used to test the tree's accuracy. [52]

4.1.2 Deep-MLSA

The deep-mlsa has three training phases, the unsupervised, the distant supervised and the supervised phase. In the first phase raw tweets are used to build the word-embeddings. The second uses tweets with happy and sad smiles to train the CNN. In the last phase the model is trained with the hand-labeled training data. [10]

4.1.3 Multinomial Naive Bayes

The basic Naïve Bayes model evaluates texts based on the presence or absence of a word while the Multinomial Naïve Bayes takes the number of occurrences of the word into consideration, too. [53]

4.1.4 Random Forest

The Random Forest Model is based on the Bag Tree. Unlike the Bag Tree, the Random Forest only uses a small sample of data for each split. Thus the created trees have less correlation because usually the same split values dominate a data set. With smaller samples it is possible for weaker split values to win the competition. [52]

4.1.5 Support Vector Machine

A Support Vector Machine plots a function that separates the data in two-dimensional space. If the plot performs very good on the training data, there is a high chance that the algorithm could be overfitting. [54]

4.2 Performance

The performance is evaluated using two data sets. One is a sentiment analysis of tweets and the other one an entry level rating for job advertisements. See Table 4 for more information on the data files. The deep-mlsa algorithm is not included because the available word-embeddings are for English and the job advertisement data is German. For detailed information on how the CNN performs on tweets, please refer to “Leveraging Large Amounts of Weakly Supervised Data for Multi-Language Sentiment Classification” [10]. For the SVM four different scalings are implemented during preprocessing: default²⁴, standard²⁵, signed²⁶ and unsigned²⁷.

	Tweets	Job advertisements
language	English	German
Size training data	14218	337
Size test data	3605	85
Classes (Distribution)	Positive (43%) Negative (15%) Neutral (42%)	1 (29%) 2 (18%) 3 (20%) 4 (33%)
Random accuracy	38%	27%

Table 4: Data Overview

The Table 5 shows the results of three runs with each data set.

	BT²⁸	MNB	RF²⁸	SVM De- fault	SVM Stand- ard	SVM Signed	SVM Un- signed
Duration Tweets²⁹	10 min	30 s	1.5 min	17 min	2.5 h	2.5 h	15 min
Duration Job Ads²⁹	20 s	15 s	17 s	40 s	3 min	5 min	40 s
Best Training F1 Score Tweets²⁹		67.93%		69.54%	70.39%	68.99%	70.04%
Test F1 Score Tweets²⁹	58.21%	58.91%	58.33%	61.74%	61.16%	61.48%	61.19%
Best Training F1 Score Job Ads²⁹		72.73%		64.35%	84.85%	72.73%	69.7%
Test F1 Score Job Ads²⁹	43.67%	39.25%	42.11%	35.04%	34.86%	42.46%	39.9%

Table 5: Performance of algorithms

The results over the different runs are quite stable, with a difference of maximum 5%. But especially with the job advertisements, there are large gaps between the cross validation during training and the performance on the test data. The worst example is the SVM with standard scaling where the best training score is 84.85% and the test score is 34.86%. A possible cause for this could be overfitting the algorithm to the training data. Optimizing the algorithms was not part of this project though it is a point that needs to be analyzed in the next phase.

Notable is also that the fast algorithms like the Bag Tree or the Random Forest perform better on the job ads data, but for the tweets waiting for the slower algorithms pays off. As seen in Figure

²⁴ No scaling

²⁵ Scaling data to center around 0

²⁶ Scaling data between -1 and 1

²⁷ Scaling data between 0 and 1

²⁸ Algorithm does not provide scores during training

²⁹ Average of three runs

34 the graph of the F1 scores during training for the SVM's changes direction a lot. To get the best model it is therefore prudent to let the training run its course.

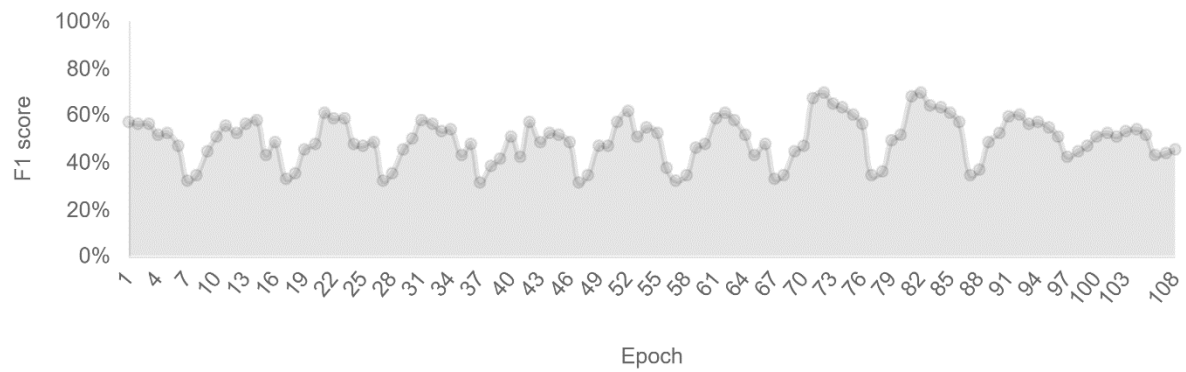


Figure 34: F1 Score/Epoch for SVM with default scaling

5 Use Cases

After having described the application (see chapter 3), we would like to present a few use cases for the application. While some of them may require further work, such as adding additional algorithms (see chapter 3.1) or being able to apply different preprocessing and feature extraction strategies to the input data, the foundation is readily available. As outlined in chapter 7.2, the application is scheduled to be extended and to reach a more complete level by Q2 2018.

5.1 One-click Solution for Researchers

Experimenting with different variations of preprocessing, feature extractions, and algorithms in general is time-consuming in more than one way:

1. **Coding time:** The time to write the code to (either by manual or automatic selection) run the algorithm(s) with different settings.
2. **Computation time:** Machine and deep learning algorithms can take, depending on the available hardware, anywhere from hours to weeks [55].
3. **Analysis time:** Once the results are available, they have to be compared and evaluated for suitability.

While the application cannot improve computation time per-se (see chapter 5.2 for how it might improve runtime indirectly), it is able to reduce the time spent coding the algorithms to zero (not accounting for time required to convert the input into the required format (see chapter 3.1.2) as there is no need to write code to perform any kind of analysis. As a byproduct, this also decreases the need for so-called “quick and dirty”³⁰ solutions (which often incur technical debt [56]) as there is a tool ready to use.

Furthermore, by providing visual guidance on how well the algorithms scored, a user is able to quickly see which algorithm scored better than others³¹. When parsing console output, which is the default for algorithms implemented using the popular *scikit-learn* package [57] (see Figure 35), there is significant cognitive overhead to discern relevant information and thereafter to compare the data extracted from the output of different processes.

As an example, below is the console output from a Naïve Bayes algorithm ran on the “job ads” input:

```
C:\Users\Linus\Dropbox\ZHAW\5. Semester\PA\Code\classification\sklearn\sentiment.analysis-master\venv\lib\site-packages\gensim\utils.py:860: UserWarning: detected Windows; aliasing chunkize to chunkize_serial
  warnings.warn("detected Windows; aliasing chunkize to chunkize_serial")
[Parallel(n_jobs=1)]: Done 1 out of 1 | elapsed: 0.0s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 2 out of 2 | elapsed: 0.0s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 3 out of 3 | elapsed: 0.0s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 4 out of 4 | elapsed: 0.1s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 5 out of 5 | elapsed: 0.1s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 6 out of 6 | elapsed: 0.2s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 7 out of 7 | elapsed: 0.2s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 8 out of 8 | elapsed: 0.2s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 9 out of 9 | elapsed: 0.2s remaining: 0.0s
[Parallel(n_jobs=1)]: Done 10 out of 10 | elapsed: 0.2s finished
Vectorizing input texts
Performing feature selection based on chi2 independence test
[CV] .....
[CV] ..... , score=0.5714285714285714, total= 0.0s
[CV] .....
```

³⁰ This term, while it may have a negative connotation, is used as a neutral term in this context, not least since factors other than knowledge and experience can influence code quality [74].

³¹ We also observed this during our user test (see chapter 6).

```
[CV] ..... , score=0.5428571428571428, total= 0.0s
[CV] .....
[CV] ..... , score=0.6470588235294118, total= 0.0s
[CV] .....
[CV] ..... , score=0.7058823529411765, total= 0.0s
[CV] .....
[CV] ..... , score=0.6764705882352942, total= 0.0s
[CV] .....
[CV] ..... , score=0.5, total= 0.0s
[CV] .....
[CV] ..... , score=0.7272727272727273, total= 0.0s
[CV] .....
[CV] ..... , score=0.48484848484848486, total= 0.0s
[CV] .....
[CV] ..... , score=0.42424242424242425, total= 0.0s
[CV] .....
[CV] ..... , score=0.59375, total= 0.0s
Training Naive Bayes
CV Score = 0.587381111536
Predicted  1  2  3  4  __all__
Actual
1          14  2  1  5      22
2          16  2  1  1      20
3           4  2  5  8      19
4           7  1  1 15      24
__all__   41  7  8 29      85
```

Figure 35: stdout and stderr from scikit-learn

In comparison, the information shown in the application’s overview of the same process:



Figure 36: The overview of the process

And detailed view of the process:

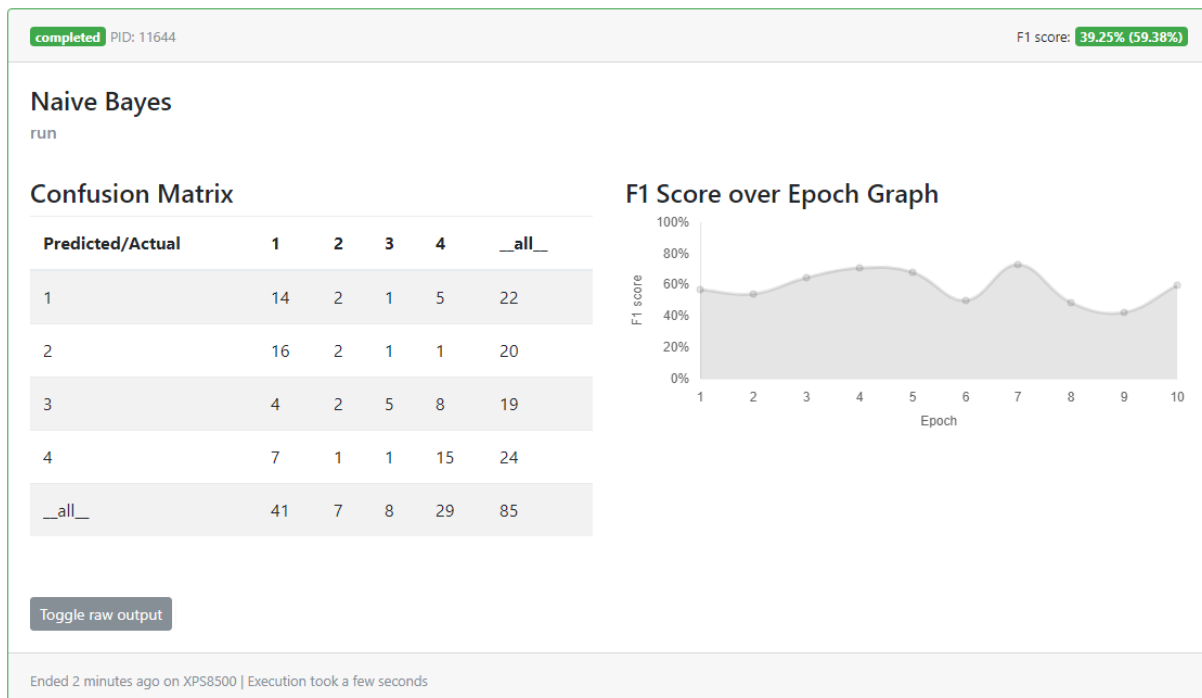


Figure 37: Detailed view of the process

By extracting the relevant data from the output and presenting (Figure 36 and Figure 37) the output from different algorithms in the same way, this cognitive overhead is reduced to the one-time task of familiarizing oneself with the interface – which, as shown in see chapter 6, takes less than fifteen minutes.

5.2 Understanding Algorithm Suitability and Solving Text Classification

While text classification is by no means a new problem [58, pp. 253-287], [59], it is still relevant today and actively researched as it has the potential to solve or simplify a wide range of problems – from phishing and spam [60] to bug reports [61] and reactions to drugs [62].

These applications require developing and evaluating different approaches for the problem at hand while only having limited knowledge which combination of preprocessing, feature extraction, and algorithm is suitable for this type of input – and in text classification, there exists a huge range of different approaches and accompanying algorithms [63, pp. 165 - 166].

Choosing the correct combination is in no small part based on experience and validated by trial-and-error [64] and often it is not apparent which algorithm is most suitable for a problem (see also chapter 4).

It would be desirable to have access to a tool which furthers understanding given an input as to which algorithms are suited for this task. This would drastically reduce the time spent experimenting with different preprocessing and feature extraction pipelines and would bring the scientific community closer to not only understanding algorithm suitability in text classification but to possibly solve text classification as a research field.

Since the application presented in this paper allows for comparison of several algorithms on one input data set, it may be used to gain insights which algorithm(s) are optimal for a given problem and to thereby contribute to solving text classification.

6 User Test

Towards the end of this project, we asked two research assistants, Fernando Benites and Jan Deriu, at the institute we worked with, to try out our application during a user test. They both interacted with the application for fifteen minutes during which they explored virtually all the features it offers. We asked them to think out loud and – combined with our observations of their interaction with the application and a short individual chat after the test – we gained valuable feedback on what works well, what could be optimized, and which features they wish were offered.

Both testers knew virtually nothing about our application and were not part of any prior meetings. They were told the application is “an AutoML tool with a web GUI”. As such they had very little clue as to what to expect and what the application’s features and limitations are.

6.1 Instructions

We decided against using a set of very specific instructions and instead went with a task description which would invite them to explore the interface. Using the application always consists of the same three (or two, if one is not interested in evaluating the trained models) steps:

1. Starting a job
2. Monitoring a job and waiting until it has completed
3. Evaluating the job using string or file input

Consequently, the instructions were based on these steps, which closely resemble the envisioned real-world usage.

These are the instructions Jan and Fernando were given:

Our application offers training and evaluation of text classification algorithms. We have provided you with two sets of data. One contains English tweets for sentiment analysis and the other German job advertisements that are rated between 1 and 4 according to the entry level required. Feel free to use either one of them.

During this test, we'd like you to do the following:

0. *Please always think out loud and tell us why and what you are doing*
1. *Start a job with >2 algorithms*
2. *Interpret the information shown in the monitor. What information can you see about a process? How do the algorithms compare to each other? What results/output do you get?*
3. *Kill a process you think is not worth waiting for until it has completed.*
4. *Evaluate the job*
 - *With text input → interpret the result*
 - *With file input → interpret the result*

6.2 Setup

During the test, the tester sat in front of a laptop with us sitting around the tester. The laptop served as a client for a remote connection to a machine (Windows 10 desktop) where the application was running. The remote machine was used for performance reasons as that machine was more powerful and capable of easily running several algorithms in parallel. The file upload dialog of the browser we used, Chrome 64, was set to a folder containing four files: *jobads_test.tsv*, *jobads_train.tsv*, *tweets_test.tsv*, *tweets_train.tsv*.

While we attempted to reduce any frictions and issues not relevant for the test (such as aforementioned file upload dialog), we only handed the tester the instructions (see chapter 6.1) and neither gave an introductory talk nor answered any questions during the test.

Both testers spent fifteen minutes interacting with the application while the other person was in a different room and as such both tests were performed independently from one another.

6.3 Results

Just as the application has three main screens (start job, monitor, evaluate) we divide this section into three parts and present the results from the user test. This section is supplemented by an additional fourth part about further comments we received from the testers.

6.3.1 Starting a Job

- Both testers had no major issues navigating the interface and quickly figured out where they could start a job.
- They selected the correct files for “evaluate tweets” for the two file inputs (train and test) and chose to run all algorithms.
- One tester, Fernando, wondered what the parameters were. He again commented on the lack of inspecting the parameters when in the “Monitoring” phase.
- The other tester, Jan, wondered what would happen if he mixed the two datasets e.g. use tweets for training and job ads for testing.
- Both testers named their jobs – whether they wanted us to be able to identify their job later on or they did not notice the job name is optional, is unclear.
- Only one tester, Fernando, also started a job with the job ads whereas we skipped this step with Jan for time reasons. When Fernando was starting this job, he clicked on “Show more information”³² again commented he would like to get more information on the algorithms’ parameters.

6.3.2 Monitoring a Job

- Neither tester has noticed the trophy icon awarded to highest scoring process nor the baseline indicated above each job’s score bar in the overview.
- Both testers were initially confused when nothing was visible on the screen since the processes were yet to generate any output.
- After noticing a few key metrics and features (runtime, different processes running simultaneously, raw output), Fernando spent a considerable amount of time getting into the details of the algorithms and interpreting the epochs in the Naïve Bayes and its confusion matrix, noting how it was not able to detect negative tweets well. When monitoring the second job (with the job ads), he interpreted the Random Forest’s matrix.
- While exploring the screen, Fernando was not able to tell which algorithms were still running and at some point (when he intended to switch to the “Evaluate” screen), accidentally killed a few algorithms.
- Both testers successfully (and intentionally) killed one or more processes using the red button in a process box.
- One of Jan’s first questions was what the color green indicated which was quickly followed by whether the bars in the overview were progress bars. Later on, he noticed the final score on the bar and read the accompanying tooltip.
- Just like Fernando, Jan also inspected the confusion matrix, but he did not explore it in such a detailed way.

³² This is a debugging feature which shows the arguments etc. as they are stored in the database. However, it does not show more information about the underlying algorithm or its parameters.

- Furthermore, Jan was wondering how many epochs have elapsed (when he was looking at the overview) and noted how the graphs were being updated in real time.

6.3.3 Evaluating a Job

- Whereas Fernando chose to run the evaluation against three generated models, Jan clicked the button to only activate the best model.
- Both testers first performed one or more evaluations with string input and then successfully (without reloading) switched to file input.
- Fernando was interested in the results and tried to get the unsigned SVM to predict the “neutral” label for an input.
- Jan wondered whether he could switch between the F1 score and other metrics (e.g. accuracy) for choosing the “best” model.
- Both testers looked at the downloaded file after the file evaluation and were delighted to see the predicted labels. Fernando compared the predicted and the actual labels.
- Jan asked whether a model is saved for a process he killed (which had already produced output) and which model (in general) was saved. We answered these questions after the test³³.

6.3.4 Additional Comments

- Jan would like to see more info about the epochs (e.g. loss) and to be able to retrieve more information (in a machine-readable format) for further logging in an external environment.
- Additionally, Jan would like to see more figures about the input such as the number of texts.
- Fernando wished he would like to have confidence figures for the evaluation with file input. However, he mentioned the confidence matrix was useful and the overview was “very good”.
- Both testers expressed positive feedback about the user interface and were pleased with it.

6.4 Conclusion

From the user test we gained valuable feedback from both testers and insights from how they interacted with the application which we added to the list of planned features (see chapter 7.2). We purposefully performed this test late in the project as we did not want the testers to be bothered with bugs and incomplete features (alpha stage) but rather with the near-final work from this project (beta stage). The test was also rewarding –both from what we observed and heard from the testers, the application is intuitive and may be useful in the every-day work of a researcher (see chapter 5.1).

³³ For the curious: the answers are: yes, a model is saved (and kept) for killed processes, too, assuming they had already generated a model; the best model (according to the F1 score, higher being better) is saved.

7 Conclusion

In this chapter we would like to take the bird's eye perspective on our project and on one hand reflect on the results achieved (chapter 7.1) and on the other hand look at the features which are scheduled for the follow-up project in form of a bachelor's thesis (chapter 7.2).

7.1 Achieved Results

In this project we have successfully (see chapter 6) built an application to run and compare several machine and deep learning algorithms. By doing so we have not only satisfied the Definition of Done (see chapter 1) but also laid the foundation for continued development of the application.

In its current state, the application contains five algorithms, which we have described (see chapter 4.1) and evaluated their performance (see chapter 4.2) on two different data sets.

The GUI (see chapter 3.3) and the backend server (see chapter 3.2) were built using state-of-the-art technologies and frameworks and the two repositories have high cohesion with one another yet as little coupling as necessary.

We have performed a user test (see chapter 6) where our application was used by two researchers for fifteen minutes each and virtually all the application's features (from a user's perspective) worked without any major problems. Additionally, we gained valuable feedback and insights on the behavioral patterns of users.

As for possible use cases (see chapter 5), there is the obvious one to use the application to perform text classification and compare different algorithms (see chapter 5.1). Yet the application also has the potential to contribute to the deeper understanding of algorithm suitability in the field of text classification (see chapter 5.2) where it could contribute to solving the problem of text classification.

7.2 Features Planned for the Bachelor's Thesis

While we have built a first working version of our application in the scope of this work, there is still ample room for more features, some of which are planned to be implemented in the scope of a follow-up bachelor's thesis.

Each feature is given an ID³⁴, a priority between 1 (highest) and 3 (lowest), and is part of a package. The packages are:

- **algorithms:** optimizing and implementing algorithms as well as parameter search
- **stats & figures:** display statistical figures about input data and results
- **preprocessing:** tweak preprocessing and feature extraction of input data
- **performance:** make the system faster and more responsive
- **(unclassified):** other features that cannot be assigned to another package

ID	Feature	Package	Priority
1	Add more deep learning algorithms to the application	algorithms	1
2	Make use of ensemble learning	algorithms	3
3	Add the ability to run in the individual algorithms in Docker containers	algorithms	3
4	Add support for cluster environments such as AWS EC2 or Microsoft Azure	performance	2

³⁴ The order of features, and thus the ID, has no meaning besides providing a unique number to reference to a feature later.

ID	Feature	Package	Priority
5	Provide an estimated time to completion for each process	stats & figures	1
6	Allow the user to choose whether to execute the processes in parallel or sequentially	performance	2
7	Allow the preprocessing step to be configured by the user	preprocessing	1
8	Allow the feature extraction to be configured by the user	preprocessing	1
9	Provide an export of any computed model in the form of a ready-to-use playground	evaluation	2
10	Add the option to set a maximum execution time for a process	performance	3
11	Implement heuristics to automatically choose algorithm(s) for a given input	algorithms	2
12	Provide general estimates for the running time of algorithms independent from the input (minutes, hours, days)	stats & figures	3
13	Automatic termination of low-scoring processes	performance	3
14	Allow for different input formats	(unclassified)	1
15	Support running one algorithm with different preprocessing configurations	algorithms	1
16	Support running one preprocessing configuration with different algorithms	algorithms	1
17	Provide sample input for the different supported file formats	(unclassified)	2
18	Generate per-class word clouds	evaluation	3
19	Analyze input data and provide statistical figures about it	stats & figures	3
20	Evaluation with file input: provide statistical figures about the label/class distribution	stats & figures	2
21	Evaluation with file input: output the number of matches	stats & figures	2
22	Evaluation with text input: display confidence of assigned class/label	stats & figures	2
23	Allow the natural language of the input data to be chosen	preprocessing	3
24	Output F1 scores per class/label	stats & figures	3
25	Run a few dozen data sets and compare with state-of-the-art results	(unclassified)	2
26	Add the option to clean up generated files on the server through the GUI	(unclassified)	2
28	Make the filename of a file downloaded from the evaluation with file input more human-friendly	(unclassified)	2

Table 6: Prioritized feature list for the bachelor's thesis

Implementing these features will enhance the application's functionality and ease of use such as more flexible input formats, distributed execution of individual processes, and configurable variations of preprocessing and feature extraction.

8 References

8.1 Bibliography

- [1] D. Panchal, "What is Definition of Done (DoD)?," Scrum Alliance, 03 09 2008. [Online]. Available: [https://www.scrumalliance.org/community/articles/2008/september/what-is-definition-of-done-\(dod\)](https://www.scrumalliance.org/community/articles/2008/september/what-is-definition-of-done-(dod)). [Accessed 05 12 2017].
- [2] IMDb, "The Good, the Bad and the Ugly (1966)," [Online]. Available: <http://www.imdb.com/title/tt0060196/>. [Accessed 16 12 2017].
- [3] Sklearn, "Sklearn," [Online]. Available: <http://scikit-learn.org>. [Accessed 01 12 2017].
- [4] M. Feurer, K. Aaron, E. Katharina, J. T. Springberg, M. Blum and F. Hutter, "Efficient and Robust Automated Machine Learning," *Advances in Neural Information Processing Systems*, no. 28, pp. 2962-2970, 2015.
- [5] Prodigy, "Prodigy," 2017. [Online]. Available: <https://prodi.gy/>. [Accessed 01 12 2017].
- [6] L. Quoc and Z. Barret, "Using Machine Learning to Explore Neural Network Architecture," Google, 17 05 2017. [Online]. Available: <https://research.googleblog.com/2017/05/using-machine-learning-to-explore.html>. [Accessed 01 12 2017].
- [7] GATE, "GATE: a full-lifecycle open source solution for text processing," The University of Sheffield, 2017. [Online]. Available: <https://gate.ac.uk/overview.html>. [Accessed 01 12 2017].
- [8] IBM, "About IBM SPSS Modeler," IBM, 2017. [Online]. Available: https://www.ibm.com/support/knowledgecenter/en/SS3RA7_18.0.0/modeler_mainhelp_client_ddita/clementine/entities/clem_family_overview.html. [Accessed 01 12 2017].
- [9] P. Li, "sentiment.analysis," 06 05 2015. [Online]. Available: <https://github.com/Poyuli/sentiment.analysis>. [Accessed 01 12 2017].
- [10] J. Deriu, A. Lucchi, V. De Luca, A. Severyn, S. Müller, M. Cieliebak, T. Hofmann and M. Jaggi, "Leveraging Large Amounts of Weakly Supervised Data for Multi-Language Sentiment Classification," in *WW 2017 - International World Wide Web Conference*, Perth, Australia, 2017.
- [11] Keras, "Keras," [Online]. Available: <https://keras.io/>. [Accessed 01 12 2017].
- [12] SQLite, "SQLite Is Serverless," [Online]. Available: <https://www.sqlite.org/serverless.html>. [Accessed 01 12 2017].
- [13] SQLite, "Full-Featured SQL," [Online]. Available: <https://www.sqlite.org/fullsql.html>. [Accessed 01 12 2017].
- [14] SQLite, "How SQLite Is tested," [Online]. Available: <https://www.sqlite.org/testing.html>. [Accessed 01 12 2017].

- [15] SQLite, "High Reliability," [Online]. Available: <https://www.sqlite.org/hirely.html>. [Accessed 01 12 2017].
- [16] P. Leach, M. Mealing and R. Salz, "A Universally Unique Identifier (UUID) URN Namespace," 07 2005. [Online]. Available: <https://tools.ietf.org/html/rfc4122>. [Accessed 01 12 2017].
- [17] expressjs.com contributors, "Express - Node.js web application framework," [Online]. Available: <https://expressjs.com/>. [Accessed 01 12 2017].
- [18] moment.js, "Moment.js," [Online]. Available: <http://momentjs.com/>. [Accessed 01 12 2017].
- [19] @veksenn and @zthall, "Lodash," [Online]. Available: <https://lodash.com/>. [Accessed 01 12 2017].
- [20] Node.js Foundation, "Node.js," [Online]. Available: <https://nodejs.org/en/>. [Accessed 01 12 2017].
- [21] Yarn, "Yarn," [Online]. Available: <https://yarnpkg.com/en/>. [Accessed 01 12 2017].
- [22] JS Foundation and contributors, "ESLint - PLuggable JavaScript linter," [Online]. Available: <https://eslint.org/>. [Accessed 01 12 2017].
- [23] M. Ogden, "Callback Hell," [Online]. Available: <http://callbackhell.com/>. [Accessed 01 12 2017].
- [24] I. A. Casas, "Node v7.6.0 (Current)," Node.js Foundation, 22 02 2017. [Online]. Available: <https://nodejs.org/en/blog/release/v7.6.0/>. [Accessed 16 12 2017].
- [25] T. Kadlecik, "Mastering Async Await in Node.js," RisingStack, 05 07 2017. [Online]. Available: <https://blog.risingstack.com/mastering-async-await-in-nodejs/>. [Accessed 01 12 2017].
- [26] MDN web docs, "Cross-Origin Resource Sharing (CORS)," Mozilla, 15 11 2017. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>. [Accessed 01 12 2017].
- [27] Node.js, "child_process.spawn," [Online]. Available: https://nodejs.org/api/child_process.html#child_process_child_process_spawn_command_args_options. [Accessed 01 12 2017].
- [28] Ian Bicking, The Open Planning Project, PyPA, "Virtualenv," [Online]. Available: <https://virtualenv.pypa.io/en/stable/>. [Accessed 03 12 2017].
- [29] Node.js, "Path: Windows vs. POSIX," [Online]. Available: https://nodejs.org/api/path.html#path_windows_vs_posix. [Accessed 09 12 2017].
- [30] Node.js, "options.detached," [Online]. Available: https://nodejs.org/api/child_process.html#child_process_options_detached. [Accessed 01 12 2017].

- [31] @mdo and @fat, "Bootstrap · The most popular HTML, CSS, and JS library in the world.," [Online]. Available: <http://getbootstrap.com/>. [Accessed 01 12 2017].
- [32] E. You, "Vue.js," [Online]. Available: <https://vuejs.org/>. [Accessed 01 12 2017].
- [33] Bootstrap Vue core team, "Bootstrap Vue," [Online]. Available: <https://bootstrap-vue.js.org/>. [Accessed 01 12 2017].
- [34] MDN web docs, "XMLHttpRequest," Mozilla, 02 12 2017. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest>. [Accessed 03 12 2017].
- [35] Vue.js, "Style Guide," [Online]. Available: <https://vuejs.org/v2/style-guide/>. [Accessed 03 12 2017].
- [36] BootstrapVue, "bootstrap-vue/webpack," [Online]. Available: <https://github.com/bootstrap-vue/webpack>. [Accessed 03 12 2017].
- [37] webpack, "webpack," [Online]. Available: <https://webpack.js.org/>. [Accessed 03 12 2017].
- [38] Babel, "Babel," [Online]. Available: <https://babeljs.io/>. [Accessed 03 12 2017].
- [39] R. Seddon, "Introduction to JavaScript Source Maps," HTML5 Rocks, 21 03 2012. [Online]. Available: <https://www.html5rocks.com/en/tutorials/developertools/sourcemaps/>. [Accessed 03 12 2017].
- [40] Google Developers, "Minify Resources (HTML, CSS, and JavaScript)," 26 04 2016. [Online]. Available: <https://developers.google.com/speed/docs/insights/MinifyResources>. [Accessed 03 12 2017].
- [41] MDN web docs, "JavaScript language resources," Mozilla, 10 10 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Language_Resources. [Accessed 03 12 2017].
- [42] Twitter Bootstrap, "Cards," [Online]. Available: <https://getbootstrap.com/docs/4.0/components/card/>. [Accessed 15 12 2017].
- [43] MDN web docs, "Date.prototype.toLocaleString()," Mozilla, 03 09 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toLocaleString. [Accessed 15 12 2017].
- [44] MDN web docs, "HTTP response status codes," Mozilla, 18 07 2017. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>. [Accessed 15 12 2017].
- [45] Moment.js, "Docs: Durations: Humanize," [Online]. Available: <http://momentjs.com/docs/#/durations/humanize/>. [Accessed 15 12 2017].
- [46] Twitter Bootstrap, "Popover," [Online]. Available: <https://getbootstrap.com/docs/4.0/components/popovers/>. [Accessed 15 12 2017].

- [47] The Segue Creative Team, "Why Whitespace is so Important in Web Design," Segue Technologies, 10 09 2015. [Online]. Available: <https://www.seguetech.com/whitespace-web-design/>. [Accessed 15 12 2017].
- [48] BootstrapVue, "Card: Header and footer," [Online]. Available: <https://bootstrap-vue.js.org/docs/components/card#header-and-footer>. [Accessed 15 12 2017].
- [49] Wikipedia, "Process identifier," 25 10 2017. [Online]. Available: https://en.wikipedia.org/wiki/Process_identifier. [Accessed 15 12 2017].
- [50] Wikipedia, "Confusion matrix," 07 08 2017. [Online]. Available: https://en.wikipedia.org/wiki/Confusion_matrix. [Accessed 15 12 2017].
- [51] Wikipedia, "Standard streams," 10 12 2017. [Online]. Available: https://en.wikipedia.org/wiki/Standard_streams. [Accessed 15 12 2017].
- [52] S. Singh, "A Practical Guide to Tree Based Learning Algorithms," 22 07 2017. [Online]. Available: <https://sadanand-singh.github.io/posts/treebasedmodels/>. [Accessed 01 12 2017].
- [53] A. McCallum and K. Nigam, "A Comparison of Event Models for Naive Bayes Text Classification," Pittsburgh, 1998.
- [54] S. Patel, "Chapter 2 : SVM (Support Vector Machine)—Theory," 03 05 2017. [Online]. Available: <https://medium.com/machine-learning-101/chapter-2-svm-support-vector-machine-theory-f0812effc72>. [Accessed 01 12 2017].
- [55] R. Shukla, "How to train your Deep Neural Network," 05 01 2017. [Online]. Available: <http://rishy.github.io/ml/2017/01/05/how-to-train-your-dnn/>. [Accessed 14 12 2017].
- [56] M. Fowler, "TechnicalDebt," 01 10 1003. [Online]. Available: <https://martinfowler.com/bliki/TechnicalDebt.html>. [Accessed 14 12 2017].
- [57] scikit-learn, "Working With Text Data," [Online]. Available: http://scikit-learn.org/stable/tutorial/text_analytics/working_with_text_data.html. [Accessed 14 12 2017].
- [58] C. D. Manning, P. Raghavan and H. Schütze, Text classification and Naive Bayes, Cambridge: Cambridge University Press, 2009.
- [59] F. Sebastiani, "Machine learning in automated text categorization," *ACM Computing Surveys (CSUR)*, vol. 34, no. 1, pp. 1-47, 2002.
- [60] N. K. Nagwani and A. Sharaff, "SMS spam filtering and thread identification using bi-level text classification and clustering techniques," *Journal of Information Science*, vol. 43, no. 1, pp. 75-87, 2017.
- [61] J. Xuan, H. Jiang, Z. Ren, J. Yan and Z. Luo, "Automatic Bug Triage using Semi-Supervised Text Classification," 15 04 2017. [Online]. Available: <https://arxiv.org/ftp/arxiv/papers/1704/1704.04769.pdf>. [Accessed 15 12 2017].
- [62] R. Ginn, P. Pimpalkhute, A. Nikfarjam, A. Patki, K. O'Connor, A. Sarker, K. Smith and C. Gonzalez, "Mining Twitter for Adverse Drug Reaction Mentions: A Corpus and

- Classification Benchmark," 05 2014. [Online]. Available: https://www.researchgate.net/profile/Abeed_Sarker/publication/280301158_Mining_Twitter_for_adverse_drug_reaction_mentions_a_corpus_and_classification_benchmark/links/56d205b608ae85c8234ae39d.pdf. [Accessed 15 12 2017].
- [63] C. C. Aggarwal and C. Zhai, "A Survey of Text Classification Algorithms," 2012. [Online]. Available: <https://pdfs.semanticscholar.org/5c89/852b90a1e9e506d237749c745bf42ac0f737.pdf>. [Accessed 14 12 2017].
- [64] D. Effrosynidis, S. Symeonidis and A. Arampatzis, "A Comparison of Pre-processing Techniques for Twitter Sentiment Analysis," 02 09 2017. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-67008-9_31. [Accessed 14 12 2017].
- [65] Docker Inc., "Get Docker CE for Ubuntu," [Online]. Available: <https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/>. [Accessed 01 12 2017].
- [66] Keymetrics, "PM2 - Advanced Node.js Process Manager," [Online]. Available: <http://pm2.keymetrics.io/>. [Accessed 03 12 2017].
- [67] Docker Inc., "docker load," [Online]. Available: <https://docs.docker.com/engine/reference/commandline/load/#extended-description>. [Accessed 03 12 2017].
- [68] Node.js, "Downloads," [Online]. Available: <https://nodejs.org/en/download/>. [Accessed 03 12 2017].
- [69] Yarn, "Installation," [Online]. Available: <https://yarnpkg.com/en/docs/install>. [Accessed 03 12 2017].
- [70] Python Software Foundation, [Online]. Available: <https://www.python.org/downloads/>. [Accessed 03 12 2017].
- [71] Twitter Bootstrap, "Badges: Contextual variations," [Online]. Available: <https://getbootstrap.com/docs/4.0/components/badge/#contextual-variations>. [Accessed 15 12 2017].
- [72] BootstrapVue, "Badges: Contextual variations," [Online]. Available: <https://bootstrap-vue.js.org/docs/components/badge/#contextual-variations>. [Accessed 15 12 2017].
- [73] Wikipedia, "Exit status: Semantics," 07 05 2017. [Online]. Available: https://en.wikipedia.org/wiki/Exit_status#Semantics. [Accessed 15 12 2017].
- [74] M. Lavallée and P. N. Robillard, "Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality," 16-24 05 2015. [Online]. Available: http://www.upedu.org/papers/ICSE2015_OrganizationalFactors/LavalleeRobillard_ICSE2015_WhyGoodDevelopersWriteBadCode.pdf. [Accessed 14 12 2017].

8.2 Figures

Figure 1: The application's navigation bar	17
Figure 2: The "Start a New Job" screen.....	18
Figure 3: Red indicates not all required inputs have been filled	19
Figure 4: Green indicates all required inputs have been filled.....	19
Figure 5: Blue indicates there are non-required inputs that are empty.....	19
Figure 6: The submit button has a yellow background and is disabled since not all required fields have been filled.....	19
Figure 7: Once all required fields have been filled, the button is active and has a blue (primary color) background.....	19
Figure 8: Choosing the data sets when starting a job.....	19
Figure 9: Choosing the algorithms when starting a job.....	20
Figure 10: The input field where a job can be named before starting it.....	20
Figure 11: The grey created label	21
Figure 12: The green completed label	21
Figure 13: The blue running label	21
Figure 14: The cyan killed label.....	21
Figure 15: The red errored label.....	21
Figure 16: The screen where one can choose which job to load and monitor.....	22
Figure 17: The "Monitor" screen for a job which ran with all algorithms on the tweets input (excerpt, for space reasons).....	23
Figure 18: The top part of the "Monitor" screen	24
Figure 19: The "Tools" menu for a job.....	24
Figure 20: The overview card in the "Monitor" screen.....	25
Figure 21: A bar chart where the current score is less than the maximum score.....	25
Figure 22: A bar chart where the maximum score is below the baseline	25
Figure 23: The detail view of a process in the overview (in this case: Naïve Bayes for Tweets) with expanded raw output.....	27
Figure 24: A process which is still running and thus can be terminated	27
Figure 25: The "Evaluate" screen for a job with no models selected and all inputs empty.....	28
Figure 26: The header section of the "Evaluate" screen.....	29
Figure 27: Choosing models where one process is still running	29
Figure 28: The two different input fields for the "Evaluate" screen	29
Figure 29: A string was provided as input.....	30
Figure 30: A file was uploaded to be evaluated	30
Figure 31: A blue (or green) button indicates the evaluation is ready to be started	30
Figure 32: The results of an evaluation with string input	31
Figure 33: The "Download" buttons are shown when a file served as input for the evaluation ..	31
Figure 34: F1 Score/Epoch for SVM with default scaling	34
Figure 35: stdout and stderr from scikit-learn.....	36
Figure 36: The overview of the process.....	36
Figure 37: Detailed view of the process.....	36

8.3 Tables

Table 1: Changed files.....	10
Table 2: Command line paramters	11
Table 3: An overview of the differently colored labels used and their meanings.....	21
Table 4: Data Overview.....	33
Table 5: Performance of algorithms	33
Table 6: Prioritized feature list for the bachelor's thesis.....	42
Table 7: How to change the application ports.....	54

9 Appendix

9.1 Official description of the task

Ist ein Tweet positiv oder negativ? Hat jemand Selbstmord-Gedanken? Welche Note sollte ein Prüfungstext erhalten?

Diese Fragen so unterschiedlich sie klingen haben eines gemeinsam: Man muss einen Text in vorgegebene Kategorien einteilen, z.B. positiv/negativ oder Selbstmord-gefährdet ja/nein. Dies wird heute typischerweise mit einem Machine-Learning-System gelöst, das für die jeweilige Aufgabenstellung konfiguriert und auf geeigneten Trainingsdaten trainiert wird. Dabei werden z.B. Feature-basierte Systeme eingesetzt, aber auch Neuronale Netze wie CNNs oder RNNs.

Die manuelle Auswahl des geeigneten Systems für eine konkrete Klassifikationsaufgabe ist sehr viel Aufwand. Deswegen möchten wir gern ein Framework entwickeln, das verschiedene Basis-Systeme für Textklassifikation kennt und für einen neuen Task automatisch ein möglichst gutes Klassifikationssystem aufbaut. Damit könnte man quasi auf Knopfdruck jedes beliebige Klassifikationsproblem lösen. Das Framework soll jedes Basis-System für den Task trainieren und optimieren, und anschließend das beste System daraus entwickeln.

Das Ziel dieser Arbeit ist die Grundfunktionen des Frameworks zu entwickeln:

- *Einlesen in das Thema automatische Text-Klassifikation*
- *Konzeption und Implementierung des Frameworks, das aus gegebenen Trainingsdaten ein Klassifikationssystem generiert. Focus ist hierbei eine ausbaufähige Architektur*
- *Integration von verschiedenen Basis-Systemen. Dabei kann auf bestehende Systeme zurückgegriffen werden, die am InIT entwickelt wurden*
- *Anwendung des Frameworks auf Klassifikationstasks und Vergleich mit dem State of the Art in der Wissenschaft.*

Je nach Qualität der Resultate können die Ergebnisse als Open-Source-Tool oder sogar in einem wissenschaftlichen Paper publiziert werden.

9.2 Installation Instructions

These installation instructions assume you have a local clone of the Git repositories *classification*, *backend*, *gui*, and *docker-app-image* which can be found at <https://github.engineering.zhaw.ch/good-bad-ugly/>. This does **not** apply if you only wish to run the docker image.

The *deep-mlsa* algorithm requires some special attention as it uses files we cannot³⁵ check in to version control. We have, however, created the necessary skeleton in folder structure to make this process easier for you. Please perform these steps in *classification/deep-mlsa-master/code*.

1. Download the “Word Embeddings trained with Word2Vec on 200 million English Tweets using 200 dimensions” from <https://spinningbytes.com/resources/embeddings/> and extract the results to *embeddings/en_embeddings_200M_200d/*. Rename *bigram* and *trigram* to *0gram* and *1gram* (respectively).
2. Download “Supervised Phase” in the chapter of “WWW-2017: Leveraging Large Amounts of Weakly Supervised Data for Multi-Language Sentiment Classification” from <https://spinningbytes.com/resources/suppmaterialpub/> and extract the contents to the *models* folder.

9.2.1 Using Docker

While the setup is not exceptionally difficult, there are quite a few steps involved which can be cumbersome yet can easily be automated. For this reason, we have prepared a *Dockerfile* which generates (using *docker-app-image/build.sh* and *docker-app-image/save-image.sh*) an archive called *app.tar*.

We do not recommend using Docker for development, as the build, due to the large size of pre-trained models and embeddings, takes a few minutes to execute in the best-case scenario (when all layers are unmodified and can be loaded from the cache) which is impractical.

You can use a pre-built *app.tar* or build the image yourself. In either case, you need a working Docker environment. While installation is possible on all major operating systems (Windows, macOS, Linux), we tested it only using Linux Ubuntu 16.04. For setup instructions, we refer to the official Docker CE installation instructions [65].

9.2.1.1 Building the Image

Please ensure you have the directory structure set up as described above and Docker installed and working. There is a convenience script located at *docker-app-image/build.sh* which you can execute³⁶ as root³⁷. The build script will perform a series of tasks such as installing Python and Node.js in the container, installing the necessary packages/libraries, and finally starting the PM2 process manager [66] to run the front- and backend. Furthermore, it will tag the image as *good-bad-ugly/app*.

To run the image, please refer to the appropriate section (9.2.1.2).

³⁵ For both legal/copyright reasons and file size issues.

³⁶ It might be necessary to give the script the necessary executable permissions which can be done with `chmod +x docker-app-image/*.sh` which will also grant that permission to the other convenience scripts related around Docker tasks.

³⁷ This is due to Docker requiring root permissions. Should you have added your user to the *docker* group (which is less secure), you still need to run the script(s) as root since the scripts only run if the user is root to cover all cases.

9.2.1.2 Running the Image

If you received the image in form of a tar archive (called *app.tar* by default), you will first have to load it into your local Docker environment. To do so, please consult the CLI reference [67].

To run the image, you can either use *docker-app-image/run.sh* (for details about the scripts, please refer to the previous section, 9.2.1.1) or run the command *docker run --rm -it -p 8080:8080 -p 3000:3000 -p 5000:5000 good-bad-ugly/app*.

The port mapping of the port 5000 is not required, it exposes PM2's health endpoint to the host which can be used for monitoring purposes. We refer to PM2's documentation for further details on how to make use of that endpoint.

You can then start the application by pointing your modern web browser to <http://localhost:8080>.

9.2.2 Manual Setup

Should you decide to perform the setup manually, you basically execute the aforementioned *Dockerfile* by hand. This can be done either on Windows or Linux. It should be noted, however, we developed the application under Windows 10. This is the recommended setup for development.

Prerequisites to run our application are Node.js 8 LTS [68], Yarn [69], and Python 3 [70]. After installing these tools and having verified their binaries are included in the PATH variable of your system, please install *virtualenv* globally by running *pip3 install virtualenv* and *nodemon* globally by running *yarn global add nodemon*³⁸.

After installing these additional packages, please setup the virtual environments for Python in the classification repo as described in the *Dockerfile*. Please note, the command to activate the corresponding *venv* needs to be executed only once per *venv* (contrary to what one reads in the *Dockerfile*) and the command is different yet similar for Windows³⁹. Also, do not forget to copy/overwrite the patched *sklearn* file as described in the *Dockerfile*.

Once you have successfully set up the Python tools, you may continue with setting up the JavaScript parts of the application. First copy or rename *backend/.env.example* to *backend/.env* and adjust if need be. Afterwards you may execute *yarn install* in both the *gui* and *backend* directory in the shell of your choice followed by *yarn dev* to start the respective services. The services started will be auto-reloading, i.e. as soon as you save/modify a source file, the respective server will be restarted automatically. You can also use *yarn prod* instead which serves production-optimized files and does not automatically restart the servers.

9.3 Port Mappings

Should the need arise to use other ports than the ones documented (3000 for the backend and 8080 for the GUI), please consult the following table.

Component, default port	Where to change the port	Further changes necessary
Backend, 3000	<i>backend/.env</i> at "PORT="	<i>docker-app-image/Dockerfile</i> : the line which reads "EXPOSE 3000"

³⁸ The attentive reader will note our *Dockerfile* also installs the *pm2* package. For local development, *pm2* is not needed and we use *nodemon* instead. They are both similar tools whereas *pm2* focusses more on production setups while *nodemon*'s focus is development environments.

³⁹ Since it also differs whether you use *cmd.exe* or PowerShell, we ask you to please familiarize yourself with *virtualenv* as described in <https://virtualenv.pypa.io/en/stable/userguide/#usage>.

		<i>gui/src/main.js</i> : change “Vue.http.options.root”
GUI, 8080	<i>gui/config/index.js</i> in “module.exports.dev.port”	<i>docker-app-image/Dockerfile</i> : the line which reads “EXPOSE 8080”

Table 7: How to change the application ports

Please also adjust the *docker run* (and/or *docker-app-image/run.sh*) command accordingly.

9.4 Meeting Notes

9.4.1 2017-09-19

- old school approach: SVM, feature-based; cheap; Python and Java; 20 - 30 algorithms
- new approach: mentioned in the ETH paper we read, based on deep learning; expensive; 10 algorithms; mostly Python

Ziel Wettbewerb - effizienter sein, als, was es bis jetzt gibt

Data Robot als Inspiration: number crunching, similar to what we'll be doing, just for a different field; PWC's solution also goes in that direction PWC: GUI as inspiration - do we want access and/or integration?

currently the deep learning algorithms are chosen based on gutt feeling

we should interface between the many different libraries and compare/evaluate them (automatically); NOT improve any of them or implement any algorithms-> unify the current process

an important step for feature-based approaches is preprocessing

what we need to do: create a pipeline à la:

```

-----> data >-----> preprocessing >-----> choose algorithms >----->
output
      csv           configure the libs           language-independent interface           GUI
              json                                     xml

```

the GUI should allow:

- monitoring
- graphs to compare the algorithms
- changing + tweaking the settings

additionally, one should be able to use clusters and keep licensing in mind

tools for feature-based approach:

- [nltk](#)
- [Stanford CoreNLP](#)
- [spacy.io](#)
- [textblob](#)
- ZHAW's own work on sentiment analysis
- [scikit-learn](#)

tools for deep learning approach:

- [keras](#) -> implement standard algos from literature; usually one-liners
- [glove](#), [word2vec](#) etc. -> need / can be trained with Wikipedia, news, Twitter

goal for PA: text in, config by hand, result out goal for BA: comparison, GUI etc.

schriftlicher Teil: 30 - 40 pages, > 20de or en? overview of literature-> already start taking notes and summarize reading Zitierstandard ist egal

DEADLINE DECEMBER 21

regular meetings

after trying out something for 30 mins, get in touch with Don (tuge@zhaw.ch), he's in the office Mon - Thu

beginning of May, take ~ 20 competitions and try to beat 'em all :)

read papers on Mark's website, <http://dreamboxx.com/mark>

9.4.2 2017-09-27

To Do / where to start

ein Klassifikationssystem zum Laufen bringen (open source oder ZHAW) -> sentiment analysis in English

deep learning (Python, TensorFlow, ready to use; github) -> need word embeddings (ready to use from Mark), train, toy around, draw learning curve etc.

scikit-learn (simple, feature-based)

usually, preprocessing is the most work

afterwards, 2 - 3 systems with completely different technologies

=> first do for sentiment analysis, afterwards e.g. suicide prevention with *same* data

=> try to find similarities (understand the concept, get the feel for it)

also try out <https://github.com/facebookresearch/fastText>

data from ZHAW (semeval 2016)

first only for English, but also account for multi-language system (e.g. Swiss German)

get in touch with [Remo Maurer \(murm\)](#) (GPU cluster; not needed for sentiment analysis, but for word embeddings)

read [semeval16 paper](#)

read [Swiss Chocolate paper](#)

book about old-school sentiment analysis: "Sentiment Analysis" by Bing Lu, ISBN 978-1-107-01789-4, Cambridge

4th week: hand in Zeitplan

<https://spinningbytes.com/>

9.4.3 2017-10-04

this semester: classic ML ansprechen + Task analysieren lassen

next semester: deep learning

ensemble: verschiedene Klassifikationen nochmals neu lernen -> better score

challenge framework: parallelization (how?)

To-Do

In Grundarchitektur einlesen:

- CNN, neuronales Netz, Standard
- SVM / random forest (semeval 2014) -- Gegensatz zu CNN

Zeitplan nächste Woche

Verschiedene Algorithmen mit den gleichen Daten ausprobieren

POS etc. in Python -> spacy.io

Python 3!

9.4.4 2017-10-10

Questions (pre-meeting)

- GUI web-based (probably Node.js)
- GUI with 3 pre-defined presets (quick, average, in-depth analysis/comparison) per algo
- settings: use preset and a selection of options (5 - 10) are configurable with a "Settings" dialog
- A bunch of progress bars, one per each algo
- Backend/preprocessing with Python
- process algos sequentially or in parallel (option, GUI)
- configurable preprocessing

Meeting

- Nadina: create issues for deep-mlsa <https://github.com/spinningbytes/deep-mlsa/issues>, cc Mark when done, tensorflow-gpu
- Linus: scikit-learn: use http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
- Linus: fastText better split train/test
- Linus: scikit-learn: not multiple files
- Linus: scikit-learn - use all classifiers in a loop; pipeline class (see mail from Don)
- At some point: use Docker
- Inspiration: <https://github.com/tensorflow/tensorboard> (visualization, not control)
- GUI should automatically do everything, maybe a "full or simple" toggle
- GUI: setting for timeout / time allotment per algo
- GUI runs on a server, decoupled from algos
- Heuristics for choosing (and not-restarting etc.) algos
- no pwc!
- graph F1-score for different parameters for each algo
- for the time being: input not a csv, but numeric feature vector; later stage
- (scikit-learn has a feature selection class)
- feature extraction: again, separate; later stage
- POS-tagging: one-hot encoding / Einheitsvektor
- semeval: emoji prediction (do tweet and emoji align) <-- our tool has to be functional by end of the year; adapt deep-mlsa

ToDo

- 5 Features by hand, test which ones are worth it (unigrams, contains positive/negative word (hard-coded list))
- read slides from Mark
- look at semeval 2018
- write-up a vision for PA / week 11 ("use case", use as Aufgabenstellung for report)

Post-meeting research

<http://gearman.org/> / <https://stackoverflow.com/questions/32974791/handle-long-running-processes-in-nodejs> / https://nodejs.org/api/child_process.html

9.4.5 2017-10-24

Questions (pre-meeting)

- discuss Vision document
- deep-mlsa: what does distant do?
- BA

Meeting

- word embeddings map similar words close (cat/dog vs rocket fuel)
- word embeddings are terrible / unable to detect sentiment ("good" and "bad" are mapped closely)
- deep-mlsa in distant phase accounts for above problem with e.g. smiley detection (3 - 4 %% difference)
- BA: good
- semeval has plenty of annotated sets of tweets
- use exact train data to compare with papers!
- for sentiment analysis: use avg of positive and negative labels (not including neutral) (<- de-facto standard for semeval)
- use DB for backend; store algo, output, params, progress, alive/dead etc.
- which tool do we use to evaluate
- for the time being: ignore fastText, focus on sklearn

ToDo

- Linus
 - GUI graph
 - GUI job uuid hash
 - GUI job <input>
 - backend DB
- Nadina
 - parallelize scikit (n_jobs=2, pickling of pipelimize, pipelimize_feature fails)
 - scikit: better output possible?
 - convert stdout (fastText etc.) to JSON
- Ask Jan about h5 problems
- DB Modell
- (graphical) overview of architecture
- dummy implementation of all arch. parts
- wireframe sketch of GUI

9.4.6 2017-11-01

Questions (pre-meeting)

- sklearn intermediate output?

Meeting

- wireframe
 - aggregated graph of all processes (scatter plot)
 - "The current winner is ... with an F1-score of ..."
 - leaderboard of algos with settings

- group leaderboard by algo
- terminate low-scoring algos / preemptive killing

ToDo

- DB
- MVP
- no OS-level details

9.4.7 2017-11-08

ToDo

- Linus
 - ✓ read Nadina's JSONs and graph them
 - ✓ file upload
 - ✓ friendly name for job
 - ✓ optional: custom job name
 - monitor: above process details: show summary charts of F scores + overview of proc status + graph of F scores over time
 - ✓ "horse race graph" in monitor
 - mini-chart per proc
 - ✓ per proc: F over t
 - ✓ summary: F per proc
 - monitor: show ETA
 - ✓ kill algorithms (and not just algorithm configs)
 - ✓ more descriptive texts for "start job"
 - ✓ job.ended_at -> momentjs buggy
- ✓ "test case": a few algos (shortened) just for show
- outlook: automated parameter selection
- wireframe of ~5 main screens / balsamiq
- ✓ standardize format for confusion matrix?
- ✓ parametrize "sentiment.py" -> choose algos, features etc. via CLI args
 - read up on "sensible defaults"
- report: intro, goal, etc.
 - mention: not about optimization, but UI/UX, framework etc.
- update Zeitplan
- ✓ definition of done
- ✓ BA title

9.4.8 2017-11-15

Questions (pre-meeting)

- BA title: classification horserace
- we are "done" / DoD
- F scores as 0.42 or 42%?
- talk about venv detection
- download? what data?

Meeting

- no download of results (csv)

- download docker/zip to run result
- export- and importability of models (per framework)
- input formats?
- provide sample inputs / structure of input
- BA: options for preprocessing (tweets, hashtags etc.)
- BA: compare different preprocessings for one algo
- BA: dimensions to change: preprocessing (lowercasing etc.) + feature extraction (count exclamation signs etc.), algorithm, parameter/settings of algo
- display ETAs for algos ("seconds", "hours", "days")
- analyze training data (stats, Eckpunkte, baselines)
- horserace chart: display baseline
- BA: all about text classification
- word clouds per class

ToDo

- Linus
 - ✓ backend cleanup input+output
 - ✓ display current AND max F1 score per algo in overview
 - ✓ F1 score overview: minichart of epochgraph -> sparklines
 - ✓ display best F1 score explicitly
 - ✓ F1 scores in %
 - ✓ job: store best process_id
 - ✓ GUI for evaluate text + file
 - ✓ display file sizes
 - ✓ display confusion matrix (replace description by conf. matrix -> hide desc behind info button)
 - ✓ docker image
 - ✓ serve gui from static
 - ✓ log level dependent on NODE_ENV=production
- export model
- export parameter settings
- CLI tool for REPL evaluation of best model + code skeleton
- Nadina
 - pickle classifier
 - store best model (using a parameter)
- BA title
 - ML/AI/text analysis/auto ML
 - buzzwords "unglaublich impressive"
- change evaluationsmass

9.4.9 2017-11-22

Questions (pre-meeting)

- done!
- code docs?
 - licensing (GPL, MIT, BSD etc.) of libs
 - own code
 - lib code
- contents of report?

- current state of the art / prior work
- outlook (-> BA)
- tech overview (gui, backend, classification)
- methodology
- docker image

Meeting

- evaluate: display stats
 - file: about label/class distribution + number of matches
 - text input: confidence, score of machine
- check licenses...
- BA packages:
 - preprocessing
 - algo optimization/implementation, horserace, parameter search
 - evaluate
 - code cleanup

ToDo

- BA title
- second set of data (not sentiment analysis)
 - e.g. age + gender; job ads (entry-level or not)
- Linus
 - Vue styleguide
 - ✓docker repo + build file (contexts...)
- confusion matrix: F1 score per class
- report
 - installation docs (-> Dockerfile)
 - state of the art:
 - pwc
 - AutoML
 - tensorboard
 - overview of tools used
 - 2 use cases
- what we did:
 - created a base architecture
 - frontend design
 - plumbing
- user test / user acceptance test
 - Jan, Fernando
 - concrete task
 - e.g. here's the data, give algorithm which is the best after 5 minutes
 - have user think out loud
- sensible code docs
 - backend/
- have a look at "Domänenübergreifende Sentiment-Analyse mit Deep Convolutional Neural Networks" as inspiration

9.4.10 2017-12-06

Questions (pre-meeting)

- we removed the diff bar once a process is dead
- email addresses of Jan, Fernando (user test)
- Show other data set
- BA feature list
- BA title:
 - Framework for automatic text classification using classical machine and deep learning algorithms
 - automatic detection/identification of suitable algo

Meeting

- BA title Mark: Automatic [selection and] optimization of machine learning and deep learning algorithms for text classification

ToDo

- BA title
- PA drucken
 - schwarzes Teil am Rücken
- 2-sided

9.4.11 2017-12-13

Questions (pre-meeting)

- report: walk-through?
- BA title (Mark: Automatic optimization of machine and deep learning algorithms for text classification)
- Fachbegriff für $currentScore < maxScore$
- Report / User Test: results only or also step-by-step?
- PA title??? Developing a tool to train and evaluate machine and deep learning algorithms for text classification through a simple user interface
- Mark: A framework to optimize machine learning algorithms for text classification through an intuitive user interface

Meeting

- when comparing algos -> look at best scoring algorithm, epoch score over time (shape of curve) -> first insight, not highly scientific, auf extreme Ausreisser eingehen, runtime
- BA:
 - given a text collection: explore/analyze collection (grep, sed, awk, wc-style), word clouds, distribution, mining, "Text Analyse Workbench", preprocessing/feature extraction -> input text, display the tokenized results
 - explore the results of n tokenizers on a small subset of the input texts (for visual inspection)

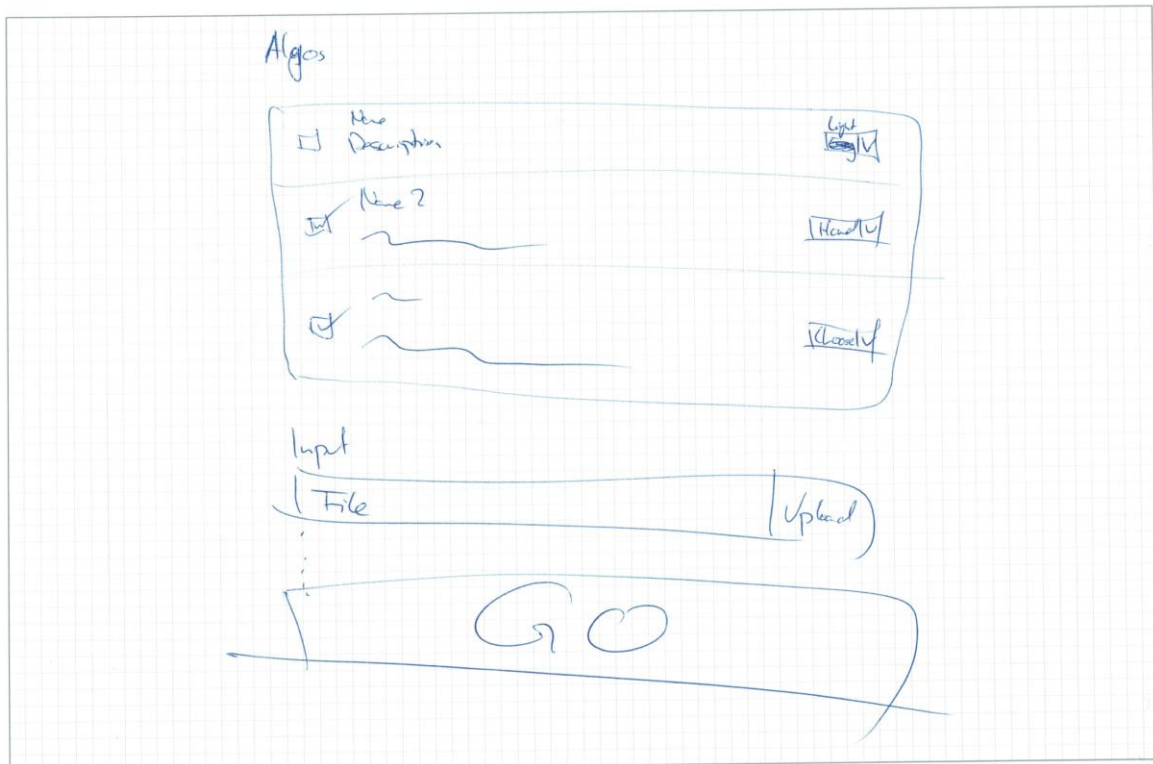
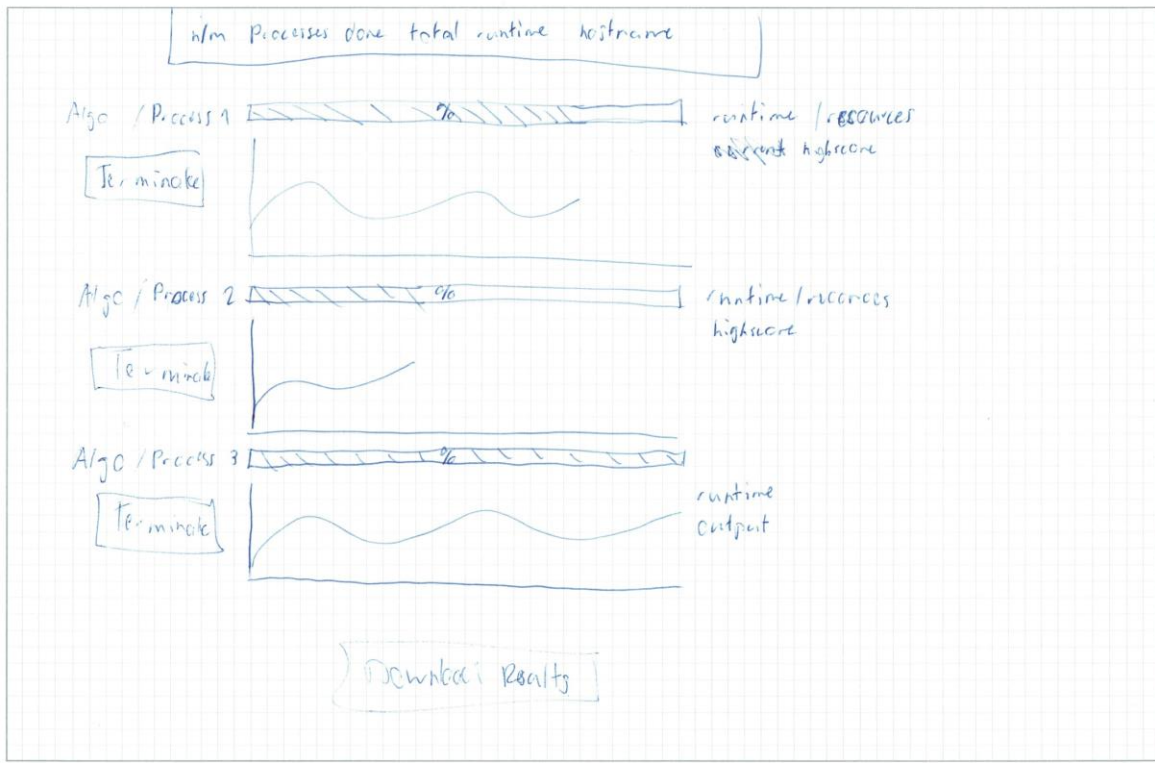
ToDo

- ask Don if he wants a print-out, too
- Linus:
 - three main screens
 - change "overfitting" to "current (max)"
 - start screen: Wikipedia source
- put user test logs in attachment
- prepare thumb drive

- **LOOK AT ZHAW GUIDELINES**
 - Plagiatserklärung
- prior work: mention AutoML (+ reference papers), AutoML for text classification; tools for text classification (GATE <https://gate.ac.uk/>, IBM SPSS https://www.ibm.com/support/knowledgecenter/en/SS3RA7_15.0.0/com.ibm.spss.ta.help/tmfc_intro.htm)
- outlook:
 - Stolpersteine / Grenzen
- use case
 - simplify work for researchers (one-click)
 - beat the problem text classification by running all and everything
 - understand which algo is optimal for problem
- good-bad-ugly <--> positive-negative-neutral

9.5 GUI Sketches

These were mentioned in chapter 9.4.6.



9.6 Timetable

Week (semester calendar)	To-Do
1 38	Kick-off
2 39	Read literature
3 40	Get familiar with tools
4 41	Test tools with same data, set up timetable
5 42	Test tools with different data
6 43	Test tools with different data
7 44	GUI
8 45	GUI + Python algorithms
9 46	GUI + Python algorithms
10 47	Testing + refactoring + fine tuning
11 48	Write report
12 49	Write report
13 50	Proofreading
14 51	Hand-in

9.7 License Information

Due to the nature of the programming languages involved (JavaScript and Python) we made heavy use of open source packages. As this application is not intended neither for commercial usage nor redistribution, no special considerations were made regarding whether a package might not be usable due to its license. Please see the file *LICENSES.txt* in the corresponding repositories/folders. We would like to thank the authors and contributors of these packages for their work.