

Entwicklerhandbuch SCHA.T.S.I.

In dieser Datei wird der Aufbau und Umgang mit SCHA.T.S.I, der Stand der Entwicklung bei Übergabe des Projektes, sowie ein Einstieg und eine Arbeitsroutine mit Git/Github zur Codeverwaltung und Docker/Dockerhub als Deployment-Technologie erklärt.

Github-Repository: <https://github.com/LSServiceOperationsHRO>

Dockerhub-Repository: <https://hub.docker.com> - hlast96

1. Aufbau des SCHA.T.S.I-Projektes

1.1 Übersicht über die Gliederung des Projektes

Dieses Kapitel gibt einen kurzen Überblick über die Gliederung von SCHA.T.S.I.

Dabei ist festzuhalten, dass SCHATSI bis einschließlich zur Version 1.4.1 in zwei Komponenten geteilt wurde, die jeweils unabhängig von einander gewartet werden können und unabhängig arbeiten.

Alle nachfolgenden Versionen sind hingegen dreigeteilt. Der Grund hierfür war die bessere Wartbarkeit und erzielte Verschlinkung der ersten Komponente.

Die beiden nachfolgenden Bilder stellen diese Aufteilungen jeweils dar.

Dabei werden auch generalisiert die Aufgaben, welche in der Komponente abgedeckt werden visualisiert.

SCHA.T.S.I vor Version 1.4.2

SCHATSI

**SCHATSI
Machine Learning R**

Laufzeit überwachen
Text extrahieren
Wörter zählen
Bigrams und Trigrams
Ausdrücke filtern
Metadaten
Referenzen
Ranking erstellen

Word-Cloud
Topic-Modelling

Dabei ist ersichtlich, dass einige Funktionalitäten ab Version 1.4.2 von *SCHATSI_DataCleanser* in den *SCHATSI_Ranker* verschoben wurden.

SCHA.T.S.I ab Version 1.4.2

**SCHATSI
DataCleanser**

**SCHATSI
Ranker**

**SCHATSI
Machine Learning R**

Laufzeit überwachen
Text extrahieren
Wörter zählen
Bigrams und Trigrams
Ausdrücke filtern
Metadaten
Referenzen

Laufzeit überwachen
Negatives Filtering
Ranking erstellen

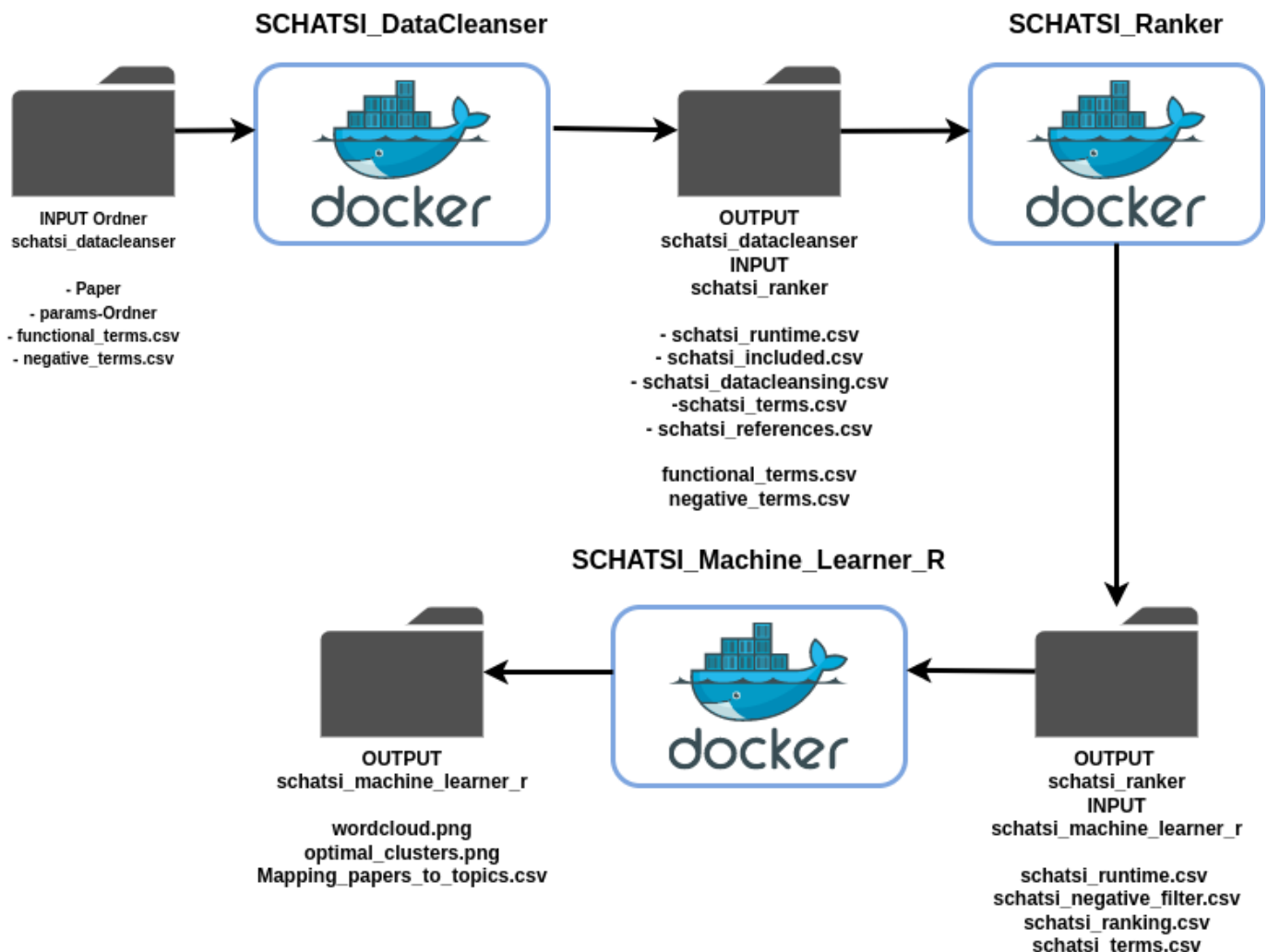
Word-Cloud
Topic-Modelling

1.2 Kommunikation und Datenaustausch zwischen den Komponenten von SCHA.T.S.I

Da SCHA.T.S.I in drei einzelnen Komponenten aufgeteilt ist, die zwar unabhängig voneinander verwaltet und eingesetzt werden können, jedoch eine gemeinsame Aufgabe erfüllen bedarf es einer Form der Kommunikation zwischen den einzelnen Komponenten.

Deshalb ist es wichtig zu verstehen was die jeweiligen Container als Input benötigen und als Output produzieren, **denn der Output des Vorausgegangenen Containers dient als Input für die Nachfolger**. Dies stellt den einzigen Weg dar, wie zwischen den Komponenten kommuniziert werden kann. Vorteilhaft ist dabei, dasss die Inputs und Outputs jeweils fixiert vorgegeben sind und der Nutzer lediglich für einen Container den entsprechenden Ordner mit den Papern oder dem Output der vorausgegangenen Komponente zu wählen hat damit die nächste Komponente starten kann.

Dieser Ablauf ist noch einmal im nachfolgenden Bild dargestellt.



1.3 Beschreibung der Funktionen von SCHA.T.S.I

Hier werden entsprechend der Aufteilung in Komponenten, die einzelnen Funktionen beschrieben.

Dabei werden ihre Aufgabe, Input, Output und ihre Funktionsweise erläutert. Dies stellt eine Übersicht über alle bisher entwickelten und bereits durchgeplanten Funktionen wieder, jedoch besteht hier kein Anspruch auf Vollständigkeit, da zukünftig weitere Funktionen von Nöten sein können.

Funktionen mit einem Sternchen müssen neu implementiert oder überarbeitet werden.

1.3.1 SCHATS.I_DataCleanser

In SCHATS.I004.py:

terms:

- INPUT: string (Text der jeweiligen Paper)

- OUTPUT: python list (gefundene single Terme)
- Funktion iteriert über den Text und hängt chars aneinander, solange bis ein Sonderzeichen oder ein Trennzeichen kommt
- Diese Zeichenkette wird in die Liste eingefügt, anschließend wird mit einer neuen Zeichenkette begonnen

bigrams:

- INPUT: python list (Output der Funktion "terms")
- OUTPUT: python list mit jeweils einem Tupel zweier Wörter als Elemente
- Funktion iteriert über die Python Liste und fügt immer das aktuelle Element und den Nachfolger zu einem Tupel zusammen
- Diese Tupel werden an eine Liste angehängt, die am Ende ausgegeben wird

trigrams:

- INPUT: python list (Output der Funktion "terms")
- OUTPUT: python list mit jeweils einem Tupel dreier Wörter als Elemente
- Funktion iteriert über die Python Liste und fügt immer das aktuelle Element, das Nächste und Übernächste Element zu einem Tupel zusammen
- Diese Tupel werden an eine Liste angehängt, die am Ende ausgegeben wird

terms_filtering:

- INPUT: python list (Output der Funktion "terms"), set mit stopwords aus stopwords.csv
- OUTPUT: python list mit den gefilterten Termen, python list mit den Häufigkeiten der jeweiligen gefilterten Terme
- Diese Funktion entfernt Dopplungen, indem es jeweils nur ein Exemplar eines jeden Terms in eine Liste anhängt wird, insofern das Element nicht in den stopwords vorkommt
- Anschließend wird die Häufigkeit eines jeden Terms, welches in der Liste der gefilterten Terme auftaucht bestimmt

bigram_filtering:

- INPUT: python list (Output der Funktion "bigrams"), set mit stopwords aus stopwords.csv

- OUTPUT: python list mit den gefilterten Bigrams, python list mit den Häufigkeiten der jeweiligen gefilterten Bigrams
- Diese Funktion entfernt Dopplungen, indem jeweils nur ein Exemplar eines jeden Bigrams in eine Liste anhängt wird, unter der Bedingung, dass weder das vordere, noch das hintere Wort ein stopword ist (KeinStopword KeinStopword)
- Anschließend wird die Häufigkeit eines jeden Bigrams, welches in der Liste der gefilterten Bigrams auftaucht bestimmt

trigram_filtering:

- INPUT: python list (Output der Funktion "trigrams"), set mit stopwords aus stopwords.csv
- OUTPUT: python list mit den gefilterten Trigrams, python list mit den Häufigkeiten der jeweiligen gefilterten Trigrams
- Diese Funktion entfernt Dopplungen, indem jeweils nur ein Exemplar eines jeden Trigrams in eine Liste anhängt wird, unter der Bedingung, dass weder das vordere, noch das hintere Wort ein stopword ist, das Wort in der Mitte jedoch ein stopword darstellt (KeinStopword Stopword KeinStopword)
- Anschließend wird die Häufigkeit eines jeden Trigrams, welches in der Liste der gefilterten Trigrams auftaucht bestimmt

In SCHATSI003.py:

string_preparation:

- INPUT: String (gesamter extrahierter Text eines Papers)
- OUTPUT: String mit dem Text, String mit den Referenzen
- Wandelt alle Groß geschriebenen Wörter in klein geschriebene Wörter um, damit kein Unterschied zwischen Groß und Kleinschreibung mehr beachtet werden muss
- Anschließend wird der Text von den Referenzen getrennt, indem das letzte Vorkommen des Wortes "reference" im gesamten Text gesucht wird und anschließend der gesamte String am Ort des letzten Vorkommens von "reference" in zwei Substrings getrennt wird

count_words:

- INPUT: String (Text der jeweiligen Paper)
- OUTPUT: Int (Gesamtanzahl der Wörter)

- Diese Funktion ermittelt die Gesamtanzahl der Wörter im Text, indem über den Text iteriert wird und jedes mal, wenn ein Trennzeichen erreicht wird, aber sich daran noch Buchstaben anschließen die Anzahl der Wörter um 1 erhöht wird

reference_data_cutting: IN WARTUNG ★

- INPUT: String (enthält jeweils den Text einer vollständigen Referenz)
- OUTPUT: 4 Strings: Autor, Erscheinungsjahr, Titel, Herkunft (also das medium, in dem es veröffentlicht wurde)
- Versuch mittels regulärer Ausdrücke die Jahreszahl der Veröffentlichung zu finden
- Versuch die Referenz in Substrings zu zerlegen, sodass diese jeweils den Autor, den Titel und die Herkunft enthalten

references: IN WARTUNG ★

- INPUT: String (Gesamter Text der Referenzen aus Funktion "string_preparation")
- OUTPUT: python list mit jeweils einem String, welcher jeweils eine Referenz enthält
- Versuch 4 verschiedene Stile der Referenzangabe zu erkennen ([1], 1., 1, " ")
- Die Referenz-Strings sollen jeweils an diesen Trennzeichen geteilt werden und an eine Liste angefügt werden.

metadata_author: TO-DO ★

- INPUT: String (Text eines Papers)
- OUTPUT: String (Autoren)

metadata_year: TO-DO ★

- INPUT: String (Text eines Papers)
- OUTPUT: String (Jahr)

metadata_title: TO-DO ★

- INPUT: String (Text eines Papers)
- OUTPUT: String (Titel des Papers)

metadata_origin: TO-DO ★

- INPUT: String (Text eines Papers)
- OUTPUT: String (Herkunft, also Medium, in dem Das Paper veröffentlicht wurde)

In main.py:

main.py:

- INPUT: PDF-Dateien der Paper, stopwords.csv, functional_terms.csv
- OUTPUT: csv-Files (included.csv, runtime.csv, datacleansing.csv, references.csv, terms.csv)
- Diese Funktion ist die eigentlich ausgeführte Funktion, die alle anderen Funktionen, auch aus den anderen Dateien aufruft.
- In dieser Funktion wird die gesamte Zeit, sowie die Zeit der einzelnen Teilaufgaben gestoppt und in *output_runtime* geschrieben
- Zu diesen einzelnen Teilaufgaben zählen:
 - Prüfung, ob Text aus Datei extrahiert werden kann -> Wenn ja, dann Eintrag in *output_included*
 - Textextraktion aus der Datei mittels des Moduls *pdftotext*
 - Diesen Text zerlegen in reinen Text und referenzen
 - Wörter eines jeden Textes zählen -> Eintrag in *output_datacleansing*
 - Anschließend Finden und filtern der einzelnen Terme, Bigrams und Trigrams, sowie herausfiltern von stopwords und nicht zulässigen Bigrams und trigrams -> Jeweils gefilterte Terme, Bigrams und Trigrams, sowie deren Häufigkeit als Eintrag in *output_terms*
 - Extrahieren der Referenzen, Eintrag jeweils in *output_references*; IN WARTUNG, deshalb nur Speicherung des gesamten Referenztextes ★
 - TO-DO: Metadaten extrahieren, diese in *output_datacleansing* schreiben und mit ausgeben ★
- Anschließend werden wird aus jeden der *outputs* eine eigene csv-Datei generiert und das Programm speichert diese Dateien im Output-Ordner

1.3.2 SCHATSI_Ranker

ranking:

- INPUT: python list mit funktionalen Termen (aus "functional_terms.csv") und python list mit den gefundenen, gefilterten Termen und deren Häufigkeiten aus den Papern
- OUTPUT: Pandas Dataframe mit den Namen der Paper, der Wortanzahl der Paper, den gefunden funktionalen Termen der Paper und der Summe der gefunden funktionalen Terme geteilt durch die Gesamtanzahl der Terme im Paper
- Diese Funktion berechnet das Ranking der einzelnen Paper
- Dazu werden für jedes Paper die Summe der gefundenen funktionalen Terme ermittelt und diese durch die Gesamtanzahl der Wörter geteilt.
- Dieser Wert stellt die Häufigkeit des Auftretens besonders nützlicher Ausdrücke im Paper dar und es gilt, je höher der Wert desto besser.
- TO-DO: Zukünftig, soll auch die Anzahl der gefundenen negativen Terme in das Ranking einfließen, indem Paper, die zwar eine hohe Anzahl funktionaler Terme aufweisen, jedoch auch negative Terme besitzen entsprechend tiefer gerankt werden ★

negative_filtering: TO-DO ★

- INPUT: python list (aus "negative_terms.csv"), python list (aus "terms.csv")
- OUTPUT: python list mit dem Namen des Papers, den vorhandenen negativen Termen und deren Anzahl
- Diese Funktion soll eine Art "negative Filterung" durchführen. Die Idee dahinter ist, dass es Terme gibt, die die Attraktivität eines Papers auch wieder senken können, selbst wenn ansonsten viele der gesuchten Terme darin vorkommen. Dies deutet nämlich auf ein Paper hin, dass zwar viele der gesuchten Begriffe nutzt, jedoch nicht das Thema behandelt, welches für das eigene Projekt bedeutsam ist.
- Deshalb soll nun aus der Liste der gefilterten Terme und Ihrer Häufigkeiten und einer Liste mit Termen, welche einen negativen Einfluss auf die Nützlichkeit des Papers haben, die Anzahl der vorhandenen negativen Terme pro Paper zu bestimmen.
- Diese Übersicht soll in eine überarbeitete Version des Rankings einfließen, um die Qualität des Rankings zu verbessern

main.py:

- INPUT: Dateien: functional_terms.csv, terms.csv

- OUTPUT: Dateien: ranking.csv, negative_filtering.csv
- Diese Funktion ist die eigentlich ausgeführte Funktion, die alle anderen Funktionen, auch aus den anderen Dateien aufruft.
- In dieser Funktion wird die gesamte Zeit, sowie die Zeit der einzelnen Teilaufgaben gestoppt und in *output_runtime* geschrieben
- Zu diesen einzelnen Teilaufgaben zählen:
 - Das Einlesen der csv-Dateien "functional_terms.csv" und "terms.csv"
 - Der Aufruf der Funktion *ranking* und eine Eintragung in ein Pandas Dataframe nach dem Ablauf
 - Der Aufruf der Funktion *negative_filtering* und eine Eintragung in ein Pandas Dataframe nach dem Ablauf
 - Anschließend werden aus den beiden Dataframes und dem *output_runtime* jeweils eine csv-Datei generiert

2. Stand der Entwicklung

In diesem Kapitel wird der aktuelle Stand der Entwicklung dargestellt. Dies stellt eine Momentaufgabe bei Übergabe des Projektes (31.08.22) dar. Dabei müssen die Funktionalitäten in der ersten Spalte von Grund auf entwickelt und diejenigen in der zweiten Spalte überarbeitet werden.

Übersicht des Entwicklungsstandes bei Übergabe des Projektes

TO DO	In Wartung/Überarbeitung	DONE
Metadaten extrahieren - Autor - Erscheinungsjahr - Titel - Herkunft	Referenzen schneiden - Reference_datacutting	Text extrahieren <i>Completed</i>
Metadaten in datacleansing.csv sichern - output_datacleansing anpassen	Referenzdaten extrahieren - references	Text und Referenzen trennen <i>Completed</i>
Machine Learning Python - Clustering aus Terms und Ranking	Ranking überarbeiten - Negatives Filtering in Ranking einbauen	Wörter zählen <i>Completed</i>
Negatives Filtering - aus negative_filter.csv die Terme finden, zählen und für Ranking vorbereiten		Terme, Bigrams, Trigrams <i>Completed</i>
		Terme/Bigrams/Trigrams filtern <i>Completed</i>
		Terme/Bigrams/Trigrams filtern <i>Completed</i>
		Funktionale Terme finden und zählen <i>Completed</i>
		Ranking aus Terms und gefundenen functional terms bilden <i>Completed</i>

3. Umgang mit Git und Github

Eine umfangreiche Anleitung und Dokumentation zur Versionsverwaltung mit Git ist unter folgendem Link zu finden: <https://git-scm.com/doc>

In diesem Kapitel werden hingegen nur die Beschreibung des Projekt-Repositories, sowie die Arbeit mit Git erläutert

3.1 Das SCHA.T.S.I Projekt auf Github

Das gesamte SCHA.T.S.I-Projekt wird in den 3 einzelnen Repositories verwaltet.

Jedes dieser Repositories enthält eine der 3 Komponenten von SCHA.T.S.I.

- Den **SCHATSI_DataCleanser** / oder auch einfach nur SCHATSI
- Den **SCHATSI_Ranker**
- und den **SCHATSI_Machine_Learner_R**

Öffnet man eines dieser Repositories so findet man zum einen die Codebasis und die gesamten Files der Komponente ,und zum anderen auf der rechten Seite unter dem Punkt *Releases* die veröffentlichten Versionen für Windows, MacOS und Linux. Dort können Sie auch neue Versionen von SCHA.T.S.I zum Download packen und bereitstellen.

Jede der Komponente ist prinzipiell unabhängig voneinander und dementsprechend werden auch deren Versionen einzeln verwaltet.

Von besonderer Bedeutung sind der Ordner `src` , da hier die Quelldateien der jeweiligen Komponente enthalten sind, sowie die Dateien `functional_terms.csv` , `negative_terms.csv` , und der Ordner `params` , da diese den Input für den Docker darstellen. Weiterhin sind die Dateien `Dockerfile` und `docker-compose.yml` bedeutsam, da diese den Aufbau des Dockers, der die Komponente enthält beschreibt.

3.2 Arbeit mit Git

Um mit dem Code von SCHA.T.S.I arbeiten zu können muss ein lokaler Ordner unter Versionsverwaltung gestellt und mit dem Github-Repository verbunden werden. Eine sehr gute und detaillierte Anleitung ist unter folgendem Link zu finden: <https://legacy.thomas-leister.de/github-fuer-anfaenger-repository-anlegen-und-code-hochladen/>

Ist eine Verbindung zum jeweiligen Repository hergestellt, dessen Code bearbeitet werden soll so sollte zuerst mit `git status` überprüft werden, ob die eigene lokale Version aktuell ist. Falls

nicht sollte mit `git pull` eine Aktualisierung der lokalen Dateien vorgenommen werden.

Anschließend können Sie die Dateien bearbeiten.

```
# NAVIGIERE IN DEN ORDNER MIT DEN UNTER VERSIONIERUNG STEHENDEN DATEIEN

# Falls Status des lokalen gits betrachtet werden will
# zeigt an, ob Dateien noch nicht versioniert sind, Ob Änderungen vorhanden
sind und ob Änderungen vorgemerkert sind
git status

# Falls Status anzeigt, dass die eigene lokale Version nicht aktuell ist
# Dies zieht alle Änderungen im globalen Repo in die lokale Version
git pull

# Alle Dateien unter Versionsverwaltung stellen und Änderungen zum commit
vormerken
git add .
# Änderungen auf den Stapel senden, welcher alle Inhalte zur Veröffentlichung
im Repository enthält
git commit
# Vorgemerkte Änderungen des commit-Stapels in Repository pushen
git push #branch, falls Spezifizierung gewünscht
```

4. Umgang mit Docker und Dockerhub

In diesem Kapitel werden die Grundbegriffe, Funktionsweise und die Arbeit mit Docker erläutert.

4.1 Funktionsweise und Anwendung von Docker

Docker dient zur Isolierung und isolierten Ausführung von Software in sogenannten `Containern`. Die Idee dahinter ist, die Software nur einmalig schreiben zu müssen und diese dann für verschiedene Betriebssysteme mit der gleichen Codebasis bereitzustellen.

Die einzige Grundlage für diesen Ansatz ist, dass auf dem Rechner selbst Docker installiert sein muss (Siehe dazu im Nutzerhandbuch `Installation` für die jeweiligen Betriebssysteme).

4.1.1 Wichtige Begriffe

1. Container

Ausgeführte Instanz eines `Images`. Nach vollendeter Ausführung wird der Container beendet. D.h. SCHA.T.S.I. wird jeweils in einem solchen Container gestartet, ausgeführt und

anschließend wieder beendet.

2. Image

Abbild/ Bauplan eines Containers. Diese bestehen aus einzelnen `Layern`, welche jeweils einen einzelnen Befehl oder eine einzelne Datei enthalten. Diese Images können gebaut werden, sind portabel und können in Repositories gespeichert werden.

In diesem Projekt wird `Dockerhub` zur Verwaltung unserer Repositories genutzt.

4.2 Dateien für die Nutzung von Docker

In diesem Projekt gibt es drei Dateien, welche für die Erstellung und Nutzung unserer Docker Container von Bedeutung sind.

- Dockerfile
- docker-compose.yml
- SCHATSI_RUN (spezifisch für alle drei Betriebssysteme)

4.2.1 Dockerfile

Dieses File dient dem Bau eines Images und stellt sozusagen den Bauplan des Images dar.

Absteigend werden nacheinander durch die Befehlsblöcke festgelegt:

- Auf welcher Softwarebasis SCHATSI aufsetzt (`FROM ...`) -> Debian Linux 10 Bullseye und Python 3.10
- welche notwendigen Installationen von uns festgelegt worden sind im requirements.txt (`COPY requirements.txt .`)
- zusätzliche Installationen, meist von Hilfsprogrammen für unsere gewünschten Installationen (`RUN apt-get update ...`)
- Durchführung der Installation der angegebenen Software (`RUN pip install ...`)
- anschließend die Quelldateien aus src-Ordner und Parameter-Dateien params-Ordner in den Container kopieren (`COPY ...`)
- zuletzt, welche Befehle beim Start eines Containers nach Plan des Images ausgeführt werden sollen `CMD ["python", "./main.py"]`

In unserem Fall wird also nach Initialisierung Python gestartet und anschließend die `main.py` ausgeführt.

4.2.2 `docker-compose.yml`

Dieses File sorgt beim Start der Software dafür, dass das gewünschte Images aus dem Dockerhub bezogen wird, indem es über den Punkt `image` den Nutzer, das Repository und den Tag festlegt.

Zusätzlich legt dieses File beim Start des Dockers zwei Partitionen an, welche für den Input in den Container hinein, sowie den Output vom Container auf das ausführende System dienen (`volume`). Diese Partitionen werden vom Nutzer beim Start des Dockers festgelegt. Dies geschieht durch Starten der Datei `SCHATSI_RUN`.

4.2.3 `SCHATSI_RUN`

Diese Datei ist der Startpunkt für den Nutzer. Mit dieser Datei wird ein Docker-Image geladen, ein Container initialisiert und in diesem die Software für SCHATSI ausgeführt. Die Datei öffnet ein Fenster, in welchem der Nutzer zu seinem Input-Ordner navigieren und diesen auswählen und bestätigen kann.

Anschließend wird diese Wahl an die Datei `docker-compose.yml` übergeben, sodass ein Container entsprechend der Wahl des Nutzers aufgebaut werden kann.

Dieses File sollte nicht verändert werden, damit es für alle Versionen von SCHATSI einsetzbar ist. Einzig das File für MacOS sollte überprüft und ggf. überarbeitet werden, da sich die Datei noch im BETA Status befindet.

4.3 Dockerhub

Unsere erstellten Images werden für jede der 3 Komponenten von SCHA.T.S.I in einem eigenen Repository auf Dockerhub verwaltet. Von dort können Sie beim Start der `SCHATSI_RUN` heruntergeladen und anschließend ausgeführt werden.

Nach der Anmeldung auf Dockerhub erscheint ein Dashboard mit Übersicht über alle Repositories des SCHA.T.S.I-Projektes.

The screenshot shows the Docker Hub profile for user **hlast96**. At the top, there is a search bar with the text "Search for great content (e.g., mysql)". Below the search bar, there is a dropdown menu showing "hlast96" and a "Create Repository" button. The main content area lists three repositories:

Repository Name	Last pushed	Not Scanned	Stars	Downloads	Visibility
hlast96 / schatsi_machine_learning_r	2 months ago	Not Scanned	0	0	Public
hlast96 / schatsi_ranker	2 months ago	Not Scanned	0	0	Public
hlast96 / schatsi	2 months ago	Not Scanned	0	117	Public


At the bottom, there is a tip: "Tip: Not finding your repository? Try switching namespace via the top left dropdown."


Gut zu erkennen sind die einzelnen Repositories, jeweils ein Repo für eine Komponente:


- `schatsi_datadleaner` - hier einfach *schatsi*
- `schatsi_ranker`
- `schatsi_machine_learning_r`

Jedes Repository besitzt eine Detailansicht.

In dieser Ansicht werden chronologisch absteigend die aktuellsten veröffentlichten Images gelistet. Die Images können jeweils gezogen werden, wenn diese in der *docker-compose.yml* als image genannt sind.

 **hlast96 / schatsi**

Description
This repository does not have a description 

 Last pushed: 2 months ago

Docker commands

Public View






To push a new tag to this repository,

```
docker push hlast96/schatsi:tagname
```

Tags and Scans

VULNERABILITY SCANNING - DISABLED [Enable](#)

This repository contains 16 tag(s).

TAG	OS	PULLED	PUSHED
20220601		---	2 months ago
20220335		---	2 months ago
20220334		---	3 months ago
20220333		---	3 months ago
20220332		---	3 months ago

[See all](#)

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions.

[Upgrade to Pro](#) [Learn more](#)

Jede der Veröffentlichungen kann wiederum genauer untersucht werden, und es wird eine Detailansicht geöffnet, welche die Historie des Images darstellt, also in welchen Schritten dieses Image erstellt wird. Sehr Hilfreich bei Fehlermeldungen, die das Image direkt und nicht den ausgeführten Code betreffen.

4.4 Verbindung zum Dockerhub im Terminal herstellen

Um nun ein `Image` zu erstellen und dieses hochzuladen müssen Sie sich mit ihrem Account von Dockerhub anmelden, damit Dockerhub die erstellten Images in die Repositories speichert.

Dazu geben Sie einfach folgenden Befehl in ihr Terminal ein:

```
docker login
```

Diesen Befehl bestätigen Sie mit `Enter`. Es wird erscheint die Aufforderung ihren Dockerhub-Usernamen und ihr Passwort einzugeben und dieses zu bestätigen.

Eine Meldung in dieser Art (Versionsabhängig) erscheint:

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, **head** over to <https://hub.docker.com> to create one.

Username: DOCKERHUBUSERNAME

Password: DOCKERHUBPASSWORD

WARNING! Your password will be stored unencrypted [in](#) /home/someuser/.docker/config.json.

Configure a credential helper to remove this warning. See

<https://docs.docker.com/engine/reference/commandline/login/#credentials-store>

Login Succeeded

War diese Anmeldung erfolgreich können Sie nun ein Image erstellen und ins Dockerhub hochladen.

4.5 Erstellen und Veröffentlichen eines neuen Docker-Images

WICHTIG: Überschreiben Sie funktionierende Images nicht, es sei denn der Code liegt gesichert im GITHUB. Ändern Sie lieber den TAG auf eine höhere Version und erstellen Sie dann ihr Image. Wenn es keine Fehler mehr gibt und dies eine Version zur Veröffentlichung ist können Sie sie später noch veröffentlichen. (Siehe 5. *Veröffentlichung eines neuen Releases*)

Die folgende Routine erstellt ein Image aus ihren lokalen Dateien, passt die Tags an die genutzten Repositories an und pusht das Image in das Repository bei Dockerhub.

```
# In Dateipfad navigieren, wo die Dateien von SCHA.T.S.I liegen

# VORHER: tag in docker-compose.yml anpassen an neue Version

# Image auf lokalem Rechner aus Dateien im angegebenen Ordner bauen
# ACHTUNG: ERSTMALES BAUEN KANN SEINE ZEIT DAUERN
sudo docker build -t REPOSITORYNAME:TAG /PFAD/ZU/DEN/DATEIEN
# Bsp: sudo docker build -t schatsi/20220434 /home/h/Github

# Tag anpassen an Repository-Name im Dockerhub
sudo docker tag REPOSITORYNAME:TAG hlast96/REPOSITORYNAME:TAG

# Anmelden mit Anmeldedaten für das Dockerhub
sudo docker login
# Passwort eingeben und mit ENTER bestätigen

# Erstelltes Image in das angegebene Repository hochladen

sudo docker push hlast96/REPOSITORYNAME:TAG
```

Im Anschluss daran sollte das neue Image in der Übersicht im entsprechenden Repository auf Dockerhub zu finden sein.

5. Veröffentlichung eines neuen Releases

In einem Release müssen enthalten sein:

- SCHATSI_RUN, **jeweils spezifisch für ein Betriebssystem** (SCHATSI_RUN: Linux, SCHATSI_RUN.exe: Windows, SCHATSI_RUN_MacOS: MacOS)
- DockerFile
- docker-compose.yml **Mit angepasstem image**, z.B. hlast96/REPO/TAG (siehe 4. Umgang mit Docker und Dockerhub)
- functional_terms.csv
- params-Ordner (mit stopwords.csv)
- ReadMe-Datei
- negative_terms.csv (ab Version 1.4.2)

Diese Dateien müssen in einen Ordner gepackt und anschließend ge.zippt werden.

Möchten Sie ein neues Release veröffentlichen, können Sie die folgende Vorgehensweise durchführen:

1. Alle nötigen Dateien in einen Ordner, zusammen mit der SCHATSI_RUN-Datei für das jeweilige Betriebssystem zippen. Da jeweils Starter für Windows, Linux und MacOS (Beta) vorhanden sind erhalten Sie insgesamt 3 Ordner.
2. Gehen Sie in das Repository der jeweiligen Komponente, von der Sie ein neues Release veröffentlichen möchten und klicken Sie rechts auf `Releases`
3. Klicken sie auf `Draft new Release`
4. Geben Sie der Veröffentlichung einen entsprechenden Tag (>1.4.1) und einen Titel
5. Schreiben Sie wichtigsten Neuerungen in das große Textfeld um Übersicht zu geben was neues erreicht wurde.
6. Geben Sie die 3 gezippten Ordner in das Anhangfeld: **Achten Sie Darauf die Ordner eindeutig zu benennen. Es muss erkennbar sein, welcher Ordner für welches Betriebssystem geeignet ist.**
7. Drücken Sie auf `Publish release` um die Dateien unter dem neuen Tag zu veröffentlichen.

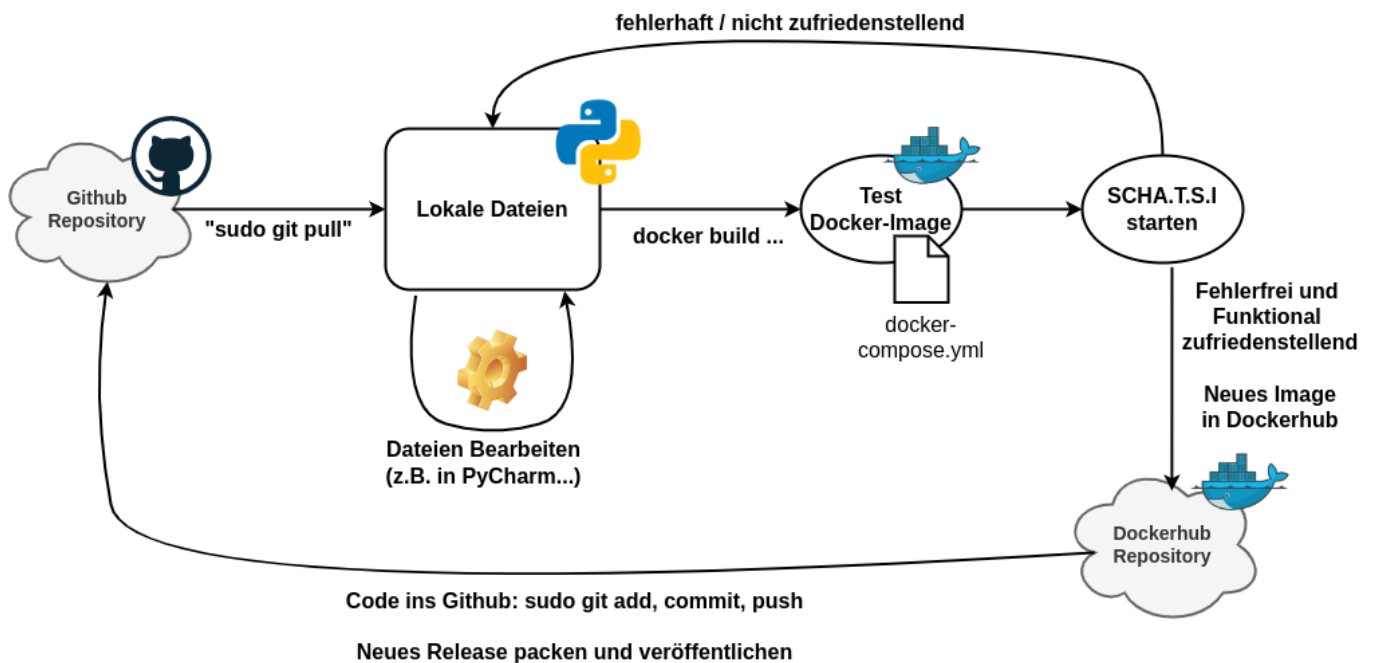
6. Arbeitsroutine

- git pull
- Anschließend Dateien bearbeiten
- Zum Ausführen und Testen: DockerRoutine mit einem "Probe-"Dockertag und angepasster Docker-Compose.yml
- SCHA.T.S.I starten mit "SCHATSI_RUN"
 - Routine läuft durch
 - Fehler werden angezeigt
- Wenn Änderungen in Ordnung sind : git Routine -> Code ins Repo
- Anschließend: Dockerroutine mit Arbeits-Dockertag

- Anschließend Neuen Release veröffentlichen: PACK-Routine und Veröffentlichen

In diesem Kapitel soll nun einmal eine gute Arbeitsroutine erläutert werden mit der bei der Entwicklung der Software in der Vergangenheit gut gearbeitet werden konnte und welche alle Schritte von der Entwicklung, Testung über Deployment und Veröffentlichung abdeckt. Dies ist natürlich nur eine Empfehlung, jedoch kann die Routine insbesondere zur Einarbeitung genutzt werden, um die Prozesse besser zu verstehen.

Der Ablauf ist im Bild unten verdeutlicht.



1. Zuerst sollte der lokal vorhandene Code auf den Stand des Codes im Github gebracht werden.

Dazu muss im Terminal in den Ordner der Dateien navigiert werden und folgendes Kommando ausgeführt werden, um alle Änderungen in die lokalen Dateien zu ziehen.

```
sudo git pull
```

2. Anschließend können die Dateien mit dem Tool der Wahl bearbeitet werden , wie z.B. PyCharm.
3. Sollen die Software getestet werden, so müssen Sie nun einmal die Routine zur Erstellung eines Dockerimages (Siehe 4.5 Erstellen und Veröffentlichen eines Docker Images) durchführen, um ein Image mit den Neuerungen zu erhalten, welche Sie testen wollen.

Wichtig: Überschreiben Sie keine funktionierende Version, sondern ändern Sie den docker-tag in der docker-compose.yml in eine neue Version, auf der Sie dann nach

belieben testen können - das erspart Ihnen später viel Arbeit bei der Wiederherstellung des alten Images.

4. War das Erstellen des Images erfolgreich so können Sie SCHA.T.S.I mit der angepassten docker-compose.yml starten. Die Software läuft durch und es werden etwaige Fehler angezeigt.

Sie können nun die lokalen Dateien erneut bearbeiten, sollte Ihnen das Ergebnis nicht gefallen. Im Anschluss bauen Sie das Docker-Image einfach nochmal und testen so stets die Neuerungen.

5. Sind Sie mit dem Ergebnis zufrieden, so können Sie das Image ins Dockerhub pushen (Siehe 4.5)
6. Des Weiteren sollten Sie dann den Code im Github updaten. (*Siehe 3.2. Arbeit mit Git*)
7. Sind die Ergebnisse bei SCHA.T.S.I soweit fortgeschritten und die Version durch Tests als stabil empfunden, so können Sie eine neues Release im Github erstellen, sodass die Nutzer die neuen Features testen können. Führen Sie dazu die Routine aus 5. *Veröffentlichung eines neuen Releases* aus.